AirbnbRevenueForecast

May 2, 2019

1 Machine Learning and Data Analysis for Business and Finance

2 Final Project: Boston Airbnb Revenue Prediction

Yuzhou Liu, Hang Xu, Leiyuxiang Wu, Jiaren Ma

2.1 Introduction

The goal of this project is to identify factors contributing to popular Airbnb listings in Boston and provide hosts with guidance on how to lift revenue by manipulating key contributing factors. We combine external data like text reviews, distance to Boston attractions and crime rate by neighborhood with Boston Airbnb listing data, applying typical models such as KNN, Ridge/Lasso,Kernel Ridge, Boosting Tree and Random Forest to find which of those variables best interpret the popularity of the given properties. Then we can use relationships between factors and properties popularity to help hosts boost their revenue.

2.2 Package Used

```
In [1]: import pandas as pd
        import numpy as np
        from sklearn.model_selection import train_test_split
        from sklearn.kernel_ridge import KernelRidge
        from sklearn.preprocessing import StandardScaler
        from sklearn.ensemble import RandomForestRegressor
        from sklearn.model_selection import GridSearchCV
        from sklearn.model_selection import ShuffleSplit
        from sklearn.ensemble import GradientBoostingRegressor
        from sklearn.model_selection import cross_val_score
        from sklearn.kernel_ridge import KernelRidge
        from sklearn.neighbors import KNeighborsRegressor
        from sklearn.linear_model import Ridge
        from sklearn.linear_model import Lasso
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [2]: def plot_feature_importances(model):
            n_features = Features.shape[1]
```

```
plt.barh(range(n_features), model.feature_importances_, align='center')
plt.yticks(np.arange(n_features), Features_df.columns)
plt.xlabel("Feature importance")
plt.ylabel("Feature")
plt.ylim(-1, n_features)
```

2.3 Datasets & Preprocesing

The main dataset investigated here is Boston airbnb listing data from **Inside Airbnb** webiste plus text reviews data, distance to Boston Attraction and crime rate by neighborhood. The airbnb lisitng data include features of about 6000 current airbnb listings around great boston area on 09/14/2018. The raw dataset contains daily price, cleaning fee, security deposite and other 27 variables.

The target variable of our interests is availability_90, which is the historical average 90-day availability (number of days). As mentioned before, one of our aims is to perdict monthly revenues for airbnb owners, and the availability_90 data contains the average monthly occupation, which can convert to monthly revenue multiplying by daily price.

For **availability_90**, we converted it to average occupancy in 90 (OCC) days for revenue prediction by simply deducting from 90. Thus, as mentioned before, it can be easily converted to monthly revenue multiplying by daily price/3.

Moreover, we did some modification and transformation such as log transformation and drop N/A records to clean the dataset. We also converted categorical variables to dummy variables and scale all variables so as to satisfy some model assumptions.

```
In [3]: airbnb = pd.read_csv('../Data/Airbnb.csv', index_col = 0)
        ResponseTimeDummy= pd.get_dummies(airbnb['host_response_time'],drop_first=True)
        RoomTypeDummy = pd.get_dummies(airbnb['room_type'],drop_first=True)
        PropTypeDummy = pd.get_dummies(airbnb['property_type'],drop_first=True)
        frame = [airbnb,ResponseTimeDummy,RoomTypeDummy,PropTypeDummy]
        airbnb = pd.concat(frame, join="inner", axis=1)
        Target = airbnb.OCC
        Features_df = airbnb.drop(['OCC', 'host_response_time','room_type', 'property_type'], a
```

Scaling down

```
In [4]: scaler = StandardScaler()
        scaler.fit(Features_df)
        Features = scaler.transform(Features_df)
```

```
/anaconda3/lib/python3.7/site-packages/sklearn/preprocessing/data.py:625: DataConversionWarnin
  return self.partial_fit(X, y)
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3: DataConversionWarning: Data wi
  This is separate from the ipykernel package so we can avoid doing imports until
```

2.4 KNN as Benchmark

Determine Proper K value Since K-Nearest Neighbor has a special advantage of "no assumptions required" as well as "easy to understand", we decide to adopt this method in our research as Benchmark model.

The overall process for all our models are similar. First, we use a for loop to find the optimal parameter value in KNN cases is the number of "n". Judging from the plot output, we find that the number of around 8 will be a good fit since the accuracy of training set and test set are close while overfitting is got rid of.

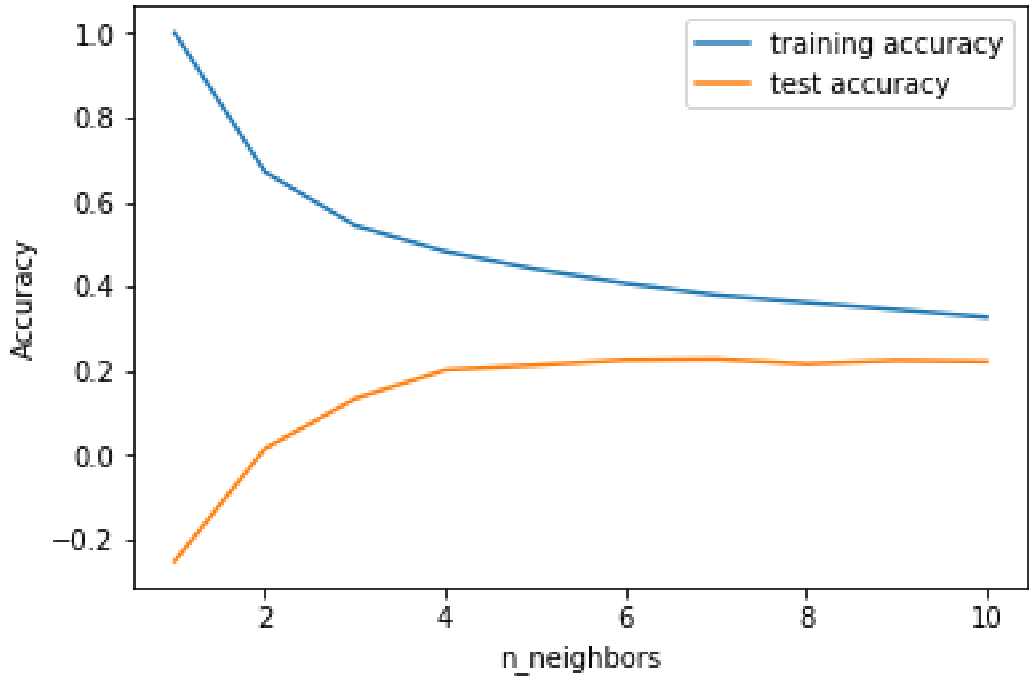
Then, we run the cross validation (different train/test data splits) 15 times to better wipe out the randomness of the regression result and get the score for modle performance.

```
In [12]: training_accuracy = []
         test_accuracy = []
         # try n_neighbors from 1 to 10
         neighbors_settings = range(1, 11)
         X_train, X_test, y_train, y_test = train_test_split(Features, Target, test_size = 0.2

         for n_neighbors in neighbors_settings:
             # build the model
             clf = KNeighborsRegressor(n_neighbors=n_neighbors)
             clf.fit(X_train, y_train)
             # record training set accuracy
             training_accuracy.append(clf.score(X_train, y_train))
             # record generalization accuracy
             test_accuracy.append(clf.score(X_test, y_test))

         plt.plot(neighbors_settings, training_accuracy, label="training accuracy")
         plt.plot(neighbors_settings, test_accuracy, label="test accuracy")
         plt.ylabel("Accuracy")
         plt.xlabel("n_neighbors")
         plt.legend()

Out[12]: <matplotlib.legend.Legend at 0x263b64d3048>
```



Cross Validation We see that the mean training score accuracy approximates 0.374 and the mean test score accuracy approaches 0.180.

```
In [11]: n = 15
         reg = KNeighborsRegressor(n_neighbors=8)
         KNNTrainScore = np.zeros(n)
         KNNTestScore = np.zeros(n)
         for i in range(n):
             X_train, X_test, y_train, y_test = train_test_split(Features, Target, test_size =
             reg.fit(X_train,y_train)
             KNNTrainScore = reg.score(X_train,y_train)
             KNNTestScore = reg.score(X_test,y_test)
         print('K-Nearest Neighbour Train Score is {}'.format(KNNTrainScore.mean()))
         print('K-Nearest Neighbour Test Score is {}'.format(KNNTestScore.mean()))
```

```
K-Nearest Neighbour Train Score is 0.3745826054526281
K-Nearest Neighbour Test Score is 0.18002205425136086
```

2.5 Ridge Regression

Grid Search

```
In [23]: score_used = 'r2'
         param_grid={'alpha':[0.001,0.01,0.1,1,10,50,100,200,300,400,500,600,700,800,900,1000]
```

```
shuffle_split = ShuffleSplit(test_size=0.2, train_size=0.8, n_splits=15)
grid_search=GridSearchCV(Ridge(),param_grid,cv=shuffle_split,scoring=score_used,
                         return_train_score=True)
grid_search.fit(Features,Target)
results = pd.DataFrame(grid_search.cv_results_)
print(results[['rank_test_score','mean_test_score','mean_train_score','param_alpha']]
```

```
    rank_test_score  mean_test_score  mean_train_score param_alpha
0                 8         0.192743          0.208192       0.001
1                 7         0.192743          0.208192        0.01
2                 6         0.192748          0.208192         0.1
3                 5         0.192797          0.208192           1
4                 4         0.193131          0.208171          10
5                 2         0.193662          0.207992          50
6                 1         0.193807          0.207641         100
7                 3         0.193439          0.206557         200
8                 9         0.192550          0.205110         300
9                10         0.191339          0.203432         400
10               11         0.189918          0.201608         500
11               12         0.188359          0.199693         600
12               13         0.186709          0.197724         700
13               14         0.185001          0.195728         800
14               15         0.183258          0.193721         900
15               16         0.181496          0.191717        1000
```

Cross Validation

```
In [22]: n = 15
         ridge = Ridge(alpha = 100)
         ridgeTrainScore = np.zeros(n)
         ridgeTestScore = np.zeros(n)
         for i in range(n):
             X_train, X_test, y_train, y_test = train_test_split(Features, Target, test_size =
             ridge.fit(X_train,y_train)
             ridgeTrainScore = ridge.score(X_train,y_train)
             ridgeTestScore = ridge.score(X_test,y_test)
         print('Ridge Train Score is {}'.format(ridgeTrainScore.mean()))
         print('Ridge Test Score is {}'.format(ridgeTestScore.mean()))
```

```
Ridge Train Score is 0.20928762949902466
Ridge Test Score is 0.18177406383409755
```

2.6 Lasso Regression

Grid Search

```
In [28]: score_used = 'r2'
         param_grid={'alpha':[0.001,0.01,0.05,0.08,0.1,1,10,100]}
         shuffle_split = ShuffleSplit(test_size=0.2, train_size=0.8, n_splits=15)
         grid_search=GridSearchCV(Lasso(),param_grid,cv=shuffle_split,scoring=score_used,
                                  return_train_score=True)
         grid_search.fit(Features,Target)
         results = pd.DataFrame(grid_search.cv_results_)
         print(results[['rank_test_score','mean_test_score','mean_train_score','param_alpha']]
```

```
   rank_test_score  mean_test_score  mean_train_score param_alpha
0                5         0.187493          0.209327       0.001
1                4         0.188004          0.209299        0.01
2                3         0.189236          0.209075        0.05
3                2         0.189815          0.208822        0.08
4                1         0.190035          0.208629         0.1
5                6         0.174666          0.182046           1
6                7         0.026774          0.028852          10
7                8        -0.001015          0.000000         100
```

Cross Validation

```
In [20]: n = 15
         lasso = Lasso(alpha = 0.1)
         lTrainScore = np.zeros(n)
         lTestScore = np.zeros(n)
         for i in range(n):
             X_train, X_test, y_train, y_test = train_test_split(Features, Target, test_size =
             lasso.fit(X_train,y_train)
             lTrainScore = ridge.score(X_train,y_train)
             lTestScore = ridge.score(X_test,y_test)
         print('Lasso Train Score is {}'.format(lTrainScore.mean()))
         print('Lasso Test Score is {}'.format(lTestScore.mean()))
```

```
Lasso Train Score is 0.20670583136969523
Lasso Test Score is 0.19954809391830128
```

The results from Ridge and Lasso regressions show little improvement from our benchmark KNN model, whihc indicates that the relationship of Features and airbnb occupancy is not linear. Therefore, we tried kernel ridge and tree based model latter.

2.7 Kernel Ridge

Grid Search

```
In [ ]: score_used = 'r2'
        param_grid={"alpha": np.linspace(1,3,11),"gamma": np.linspace(0.01,0.1,11),'kernel':['
        shuffle_split = ShuffleSplit(test_size=0.2, train_size=0.8, n_splits=15)
```

```
grid_search=GridSearchCV(KernelRidge(),param_grid,cv=shuffle_split,scoring=score_used,
                         return_train_score=True)

kernel_Ridge=grid_search.fit(Features,Target)
KRcvResult = pd.DataFrame(kernel_Ridge.cv_results_)
```

In [5]:
```
KRcvResult = pd.read_excel('../Data/KRcvResult.xlsx', sheet_name = 'Grid_Search')
print(KRcvResult[['rank_test_score','mean_test_score','mean_train_score','param_alpha'
```

```
    rank_test_score  mean_test_score  mean_train_score  param_alpha  \
0                 3         0.227616          0.326363          1.0
1                 1         0.230205          0.391882          1.0
2                 7         0.223749          0.443405          1.0
3                26         0.209822          0.484340          1.0
4                39         0.189537          0.516697          1.0
5                50         0.163795          0.542055          1.0
6                61         0.133286          0.561636          1.0
7                72         0.098562          0.576389          1.0
8                83         0.060109          0.587075          1.0
9                92         0.018375          0.594312          1.0
10              102        -0.026229          0.598609          1.0
11                4         0.225752          0.317011          1.2
12                2         0.227832          0.377569          1.2
13               11         0.220452          0.424457          1.2
14               30         0.205504          0.461258          1.2
15               42         0.184109          0.489832          1.2
16               53         0.157154          0.511629          1.2
17               63         0.125321          0.527778          1.2
18               74         0.089168          0.539171          1.2
19               85         0.049188          0.546525          1.2
20               96         0.005839          0.550432          1.2
21              106        -0.040462          0.551380          1.2
22                6         0.223904          0.309244          1.4
23                5         0.225416          0.365666          1.4
24               15         0.217100          0.408575          1.4
25               33         0.201117          0.441711          1.4
26               44         0.178615          0.466855          1.4
27               55         0.150471          0.485362          1.4
28               66         0.117358          0.498296          1.4
29               77         0.079835          0.506503          1.4
..              ...              ...               ...          ...
91               45         0.176487          0.363062          2.6
92               56         0.147970          0.372689          2.6
93               67         0.113465          0.375827          2.6
94               78         0.073608          0.373351          2.6
95               89         0.028939          0.365972          2.6
96              101        -0.020053          0.354295          2.6
97              112        -0.072907          0.338845          2.6
```

98	119	-0.129192	0.320090	2.6
99	22	0.212549	0.275684	2.8
100	27	0.209600	0.313824	2.8
101	36	0.195285	0.338380	2.8
102	47	0.172761	0.353625	2.8
103	58	0.143355	0.361246	2.8
104	69	0.107913	0.362355	2.8
105	80	0.067066	0.357808	2.8
106	91	0.021348	0.348306	2.8
107	103	-0.028755	0.334447	2.8
108	114	-0.082784	0.316753	2.8
109	120	-0.140304	0.295697	2.8
110	24	0.211142	0.272436	3.0
111	28	0.207558	0.308763	3.0
112	38	0.192480	0.331462	3.0
113	48	0.169139	0.344832	3.0
114	59	0.138872	0.350563	3.0
115	70	0.102524	0.349752	3.0
116	82	0.060718	0.343242	3.0
117	94	0.013983	0.331722	3.0
118	105	-0.037198	0.315783	3.0
119	115	-0.092369	0.295947	3.0
120	121	-0.151086	0.272693	3.0

	param_gamma	param_kernel
0	0.010	rbf
1	0.019	rbf
2	0.028	rbf
3	0.037	rbf
4	0.046	rbf
5	0.055	rbf
6	0.064	rbf
7	0.073	rbf
8	0.082	rbf
9	0.091	rbf
10	0.100	rbf
11	0.010	rbf
12	0.019	rbf
13	0.028	rbf
14	0.037	rbf
15	0.046	rbf
16	0.055	rbf
17	0.064	rbf
18	0.073	rbf
19	0.082	rbf
20	0.091	rbf
21	0.100	rbf
22	0.010	rbf

```
23          0.019          rbf
24          0.028          rbf
25          0.037          rbf
26          0.046          rbf
27          0.055          rbf
28          0.064          rbf
29          0.073          rbf
..            ...          ...
91          0.037          rbf
92          0.046          rbf
93          0.055          rbf
94          0.064          rbf
95          0.073          rbf
96          0.082          rbf
97          0.091          rbf
98          0.100          rbf
99          0.010          rbf
100         0.019          rbf
101         0.028          rbf
102         0.037          rbf
103         0.046          rbf
104         0.055          rbf
105         0.064          rbf
106         0.073          rbf
107         0.082          rbf
108         0.091          rbf
109         0.100          rbf
110         0.010          rbf
111         0.019          rbf
112         0.028          rbf
113         0.037          rbf
114         0.046          rbf
115         0.055          rbf
116         0.064          rbf
117         0.073          rbf
118         0.082          rbf
119         0.091          rbf
120         0.100          rbf

[121 rows x 6 columns]
```

```
In [7]: B_Paremeter=KRcvResult.loc[KRcvResult['rank_test_score']==1]
        B_Paremeter=B_Paremeter.iloc[:,[4,5,23,41]]
        print(B_Paremeter)
```

```
   param_alpha  param_gamma  mean_test_score  mean_train_score
1          1.0        0.019         0.230205          0.391882
```

```
In [24]: n = 15
         KR = KernelRidge(alpha=1.,gamma=0.019,kernel='rbf')
         KRTrainScore = np.zeros(n)
         KRTestScore = np.zeros(n)
         for i in range(n):
             X_train, X_test, y_train, y_test = train_test_split(Features, Target, test_size =
             KR.fit(X_train,y_train)
             KRTrainScore = KR.score(X_train,y_train)
             KRTestScore = KR.score(X_test,y_test)
         print('KernelRidge Train Score is {}'.format(KRTrainScore.mean()))
         print('KernelRidge Test Score is {}'.format(KRTestScore.mean()))
```

```
KernelRidge Train Score is 0.38444126523588773
KernelRidge Test Score is 0.23340335219362562
```

The best Kernel Ridge regression gives us 0.23 mean_test_score, which is slightly higher than the Linear Regression method but still doesn't meet our expectation.The reason may be that we have many categorical predictors in our dataset, which may be poor indicators as the distance between two pointswhich are calculated in Kernel Function, doesn't represent any meaningful result. Therefore, we think the tree-type model,combining with bagging technique, would be the appropriate one to our dataset.

2.8 Boost Tree

Grid Search

```
In [30]: score_used = 'r2'
         param_grid={'learning_rate':[0.05,0.1,0.15,0.2],'max_depth':[5,10,15,20]}
         shuffle_split = ShuffleSplit(test_size=0.2, train_size=0.8, n_splits=15)
         grid_search=GridSearchCV(GradientBoostingRegressor(n_estimators=100),param_grid,cv=shu
                                 return_train_score=True)
         grid_search.fit(Features,Target)
         BTcvResult = pd.DataFrame(grid_search.cv_results_)
```

```
In [6]: BTcvResult = pd.read_csv('../Data/BTcvResult.csv',index_col=0)
        print(BTcvResult[['rank_test_score','mean_test_score','mean_train_score','param_learni
```

```
    rank_test_score  mean_test_score  mean_train_score  param_learning_rate  \
0                 6         0.321315          0.543580                 0.05
1                 1         0.338401          0.948304                 0.05
2                 9         0.255545          0.999445                 0.05
3                16         0.070316          0.999951                 0.05
4                 3         0.323943          0.655171                 0.10
5                 2         0.333588          0.988345                 0.10
6                10         0.251383          0.999998                 0.10
7                14         0.081453          1.000000                 0.10
8                 4         0.322206          0.741140                 0.15
9                 5         0.321561          0.997487                 0.15
```

```
10            11     0.245835      1.000000            0.15
11            15     0.075492      1.000000            0.15
12             7     0.311311      0.803302            0.20
13             8     0.304468      0.999400            0.20
14            12     0.241292      1.000000            0.20
15            13     0.087489      1.000000            0.20

    param_max_depth
0                 5
1                10
2                15
3                20
4                 5
5                10
6                15
7                20
8                 5
9                10
10               15
11               20
12                5
13               10
14               15
15               20
```

From the grid search result, the best paramettter combination is 5 max_depth and 0.05 learning_rate, whihc gives a relative better test secore and not very large test and train gaps. We believe that the combination offers the best prediction performance and has less possibility of overfitting issues.

Cross Validation

In [25]:
```python
n = 15
gbrt = GradientBoostingRegressor(n_estimators=100,max_depth=5,learning_rate=0.05)
gbrtTrainScore = np.zeros(n)
gbrtTestScore = np.zeros(n)
for i in range(n):
    X_train, X_test, y_train, y_test = train_test_split(Features, Target, test_size =
    gbrt.fit(X_train,y_train)
    gbrtTrainScore = gbrt.score(X_train,y_train)
    gbrtTestScore = gbrt.score(X_test,y_test)
print('GradientBoosting Train Score is {}'.format(gbrtTrainScore.mean()))
print('GradientBoosting Test Score is {}'.format(gbrtTestScore.mean()))
```

```
GradientBoosting Train Score is 0.5369947124235599
GradientBoosting Test Score is 0.35578740838117695
```

The Tree Based model here offers a lot better prediction performance compared to all above models. The main reason here we believe is that the Feature set here is more suitable to a tree based model. Tree based model is better when handling 0/1 dummies in feature sets. The Airbnb data has 3 categorical variables which were converted into 9 dummies. Therefore, 9 out of 28 X variables are dummies, which makes tree based model a better choice.

2.9 Random Forest

Gridsearch

```
In [29]: score_used = 'r2'
         param_grid={'max_features':[10,15,20,25],'max_depth':[5,10,15,20]}
         shuffle_split = ShuffleSplit(test_size=0.2, train_size=0.8, n_splits=15)
         grid_search=GridSearchCV(RandomForestRegressor(n_estimators=100),param_grid,cv=shuffl
                                  return_train_score=True)
         grid_search.fit(Features,Target)
         RFcvResult = pd.DataFrame(grid_search.cv_results_)
```

```
In [6]: RFcvResult = pd.read_csv('../Data/RFcvResult.csv',index_col=0)
        print(RFcvResult[['rank_test_score','mean_test_score','mean_train_score','param_max_fe
```

```
    rank_test_score  mean_test_score  mean_train_score  param_max_features  \
0                15         0.268981          0.331519                  10
1                13         0.272221          0.338160                  15
2                14         0.270218          0.336478                  20
3                16         0.266156          0.333034                  25
4                11         0.357217          0.657071                  10
5                 9         0.359831          0.666370                  15
6                10         0.357895          0.665381                  20
7                12         0.350703          0.656272                  25
8                 6         0.381922          0.856830                  10
9                 3         0.384479          0.859929                  15
10                5         0.383381          0.860619                  20
11                8         0.379319          0.857206                  25
12                2         0.386323          0.906603                  10
13                1         0.387667          0.908066                  15
14                4         0.384171          0.907470                  20
15                7         0.381014          0.907813                  25

    param_max_depth
0                 5
1                 5
2                 5
3                 5
4                10
5                10
6                10
7                10
8                15
```

```
9                 15
10                15
11                15
12                20
13                20
14                20
15                20
```

From the grid search result, the best parametter combination is 15 max_depth and 10 max_depth, whihc gives a relative better test secore and not very large test and train gaps. We believe that the combination offers the best prediction performance and has less possibility of overfitting issues.

Cross Validation

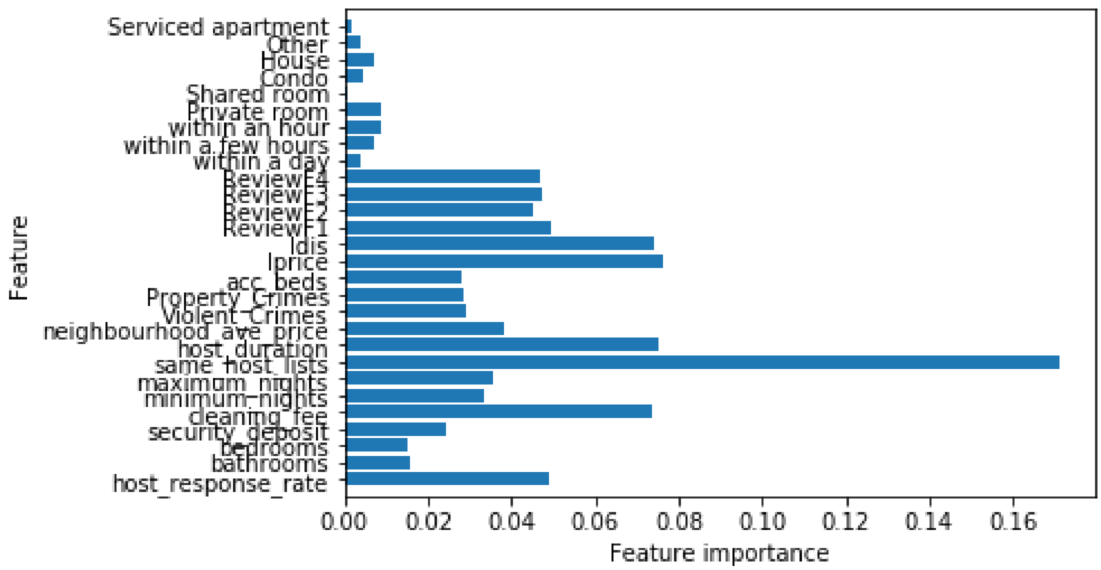
```
In [26]: n = 15
         forest = RandomForestRegressor(n_estimators=100,max_features = 10,max_depth = 15)
         forestTrainScore = np.zeros(n)
         forestTestScore = np.zeros(n)
         for i in range(n):
             X_train, X_test, y_train, y_test = train_test_split(Features, Target, test_size =
             forest.fit(X_train,y_train)
             forestTrainScore = forest.score(X_train,y_train)
             forestTestScore = forest.score(X_test,y_test)
         print('forestBoosting Train Score is {}'.format(forestTrainScore.mean()))
         print('forestBoosting Test Score is {}'.format(forestTestScore.mean()))
```

```
forestBoosting Train Score is 0.8546731650281809
forestBoosting Test Score is 0.3800719459845968
```

Like BoostTree Model, Random Forest model results are way better than all other models. In addition, the Forest gives us a slightly better prediction performance. We believe the performance enhancement comming from the feature diversification trees of random forest, which is controlled by the max_feature parameter.

Bussiness Insides from Feature Importance Since the Random Forest model is the Best Model we have, we plot the feature importance inside that model to find useful insides to predicting Boston airbnb revenue.

```
In [32]: plot_feature_importances(forest)
```



From the plot above, The most improtant variable for predicting airbnb revenues is **same host lists**, which indicates how many airbnb listings are the host has. We can say that whether a list has experienced (or we can say commercial) airbnb host are more likely to effect higher revenues. Besides that, other important variables are **host durations**, **log price**, **distance to boston attractions** and **Cleaning fee**, these are also important features which can bring more profitable airbnb listings. Then after all that, customers' reveiews also show some important on revenues, based on which topic it talks about.(The reveiw vectors are result from LDA topical analysis results)

2.10 Conclusion

```
In [31]: Result = {'KNN':[KNNTrainScore.mean(),KNNTestScore.mean()],
                   'Ridge':[ridgeTrainScore.mean(),ridgeTestScore.mean()],
                   'Lasso':[lTrainScore.mean(),lTestScore.mean()],
                   'KernelRidge':[KRTrainScore.mean(),KRTestScore.mean()],
                   'BoostTree':[gbrtTrainScore.mean(),gbrtTestScore.mean()],
                   'RandomForest':[forestTrainScore.mean(),forestTestScore.mean()]}
         pd.DataFrame(Result,index = ['Trainset Score','Testset Score'])
```

```
Out[31]:                      KNN     Ridge     Lasso  KernelRidge  BoostTree  \
         Trainset Score  0.374583  0.209288  0.206706     0.384441   0.536995
         Testset Score   0.180022  0.181774  0.199548     0.233403   0.355787

                         RandomForest
         Trainset Score      0.854673
         Testset Score       0.380072
```

Unfortunately, our models all fail to provide satisfied predictability, and we are also confused about the results. After discussion, we argue that the reason might be:

1. With only about 5000 records, it is still difficult for these models to accurately fit nonlinearity in these data. Besides, large number of dummy variables and predictors makes it even harder for regression models to function well.

2. The target variable OCC or availability_90 comes from original data, which is not perfectly the ideal target variable for us, because it is much better if we can get past occupancy information in 90 days from this variable, however, without a data dictionary (the website does not seem to provide it), we can only tell this variable is an average forecast based on past data. Therefore, our result is not satisfying.

Although the output of our models are not ideal and we are unable to offer a perfect model to predict average occupancy and revenue for Airbnb listers, our work still sheds light on what an Airbnb lister can do to improve the business value of their Airbnb listing property. Here is our advice:

1. Commercialization of airbnb(**some host list**) is a major potential revenue factor for Airbnb listers. Commercial airbnb and personal airbnb listings seem to be treated differently by the market.

2. Reviews are not so crucial as what we originally thought. Listers should pay more attention on how to improve their service rather than sit there worrying about a few negative reviews.

```
In [ ]:
```