house-prices-advanced-regression-techniques

January 11, 2020

1 House Prices: Advanced Regression Techniques

1.1 Outline:

1. Introduction

2. Data Wrangling

3. Data Story

4. Inferential Satistics

5. Macine Learning

1.2 1. Introduction

House Prices: Advanced Regression Techniques from Kaggle competitions is chosen to be first Capstone Project. Typically, a home buyer would not describe their dream house beginning with the height of the basement ceiling or the proximity to an east-west railroad. But this playground competition's dataset proves that much more influences price negotiations than the number of bedrooms or a white-picket fence. The project is to explore 79 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa to predict the final price of each home.

Real estate sales agents, sellers and buyers will be interesting in predicting the sales price as it will help them to set their listing price, offering price and final sales price.

The dataset from Kaggle is modified from the Ames Housing dataset which was compiled by Dean De Cock for use in data science education. It's an incredible alternative for scientists looking for a modernized and expanded version of the often-cited Boston Housing dataset. The data set is provided in csv files.

The work would include data wrangling, exploratory data analysis and machine learning.

Creative feature engineering and advanced regression techniques like random forest and gradient boosting skills are needed to complete this project.

The result will be submitted to Kaggle. It will be evaluated on Root-Mean-Squared-Error (RMSE) between the logarithm of the predicted value

1.3 2. Data Wrangling

1.3.1 2.1 Data Set Variables

The data set is provided by Kaggle. The data is loaded in to Pandas data frame without difficulty.

In order to understand our data, we should look at each variable and try to understand their meaning and relevance to this problem. The detail description of each variabl can be found in provided "data_description.txt". We can create a spreadsheet with the following columns: * Variable - Variable name. * Type - Identification of the variables' type. There are two possible values for this field: 'numerical' or 'categorical'. * Expectation - Our expectation about the variable influence in 'SalePrice'. We can use a categorical scale with 'High', 'Medium' and 'Low' as possible values. * Conclusion - Our conclusions about the importance of the variable, after we give a quick look at the data. We can keep with the same categorical scale as in 'Expectation'. * Comments - Any general comments that occurred to us.

[3]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from scipy.stats import norm
from sklearn.preprocessing import StandardScaler
from scipy import stats
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
```

[4]:
```
#loan training data
df_train = pd.read_csv('../input/train.csv')
```

[5]:
```
#check Variable
df_train.columns
```

[5]:
```
Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
       'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
       'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
       'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
       'MasVnrArea', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
       'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1',
       'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating',
       'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
       'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
       'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
       'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'SalePrice'],
      dtype='object')
```

1.3.2 2.2 Missing data

Missing data can imply a reduction of the sample size. This can prevent us from proceeding with the analysis. We need to check prevalent of missing data and if missing data is random or have a pattern.

[6]:
```
#missing data
total = df_train.isnull().sum().sort_values(ascending=False)
percent = (df_train.isnull().sum()/df_train.isnull().count()).
 ↪sort_values(ascending=False)
missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])
missing_data.head(25)
```

[6]:

	Total	Percent
PoolQC	1453	0.995205
MiscFeature	1406	0.963014
Alley	1369	0.937671
Fence	1179	0.807534
FireplaceQu	690	0.472603
LotFrontage	259	0.177397
GarageCond	81	0.055479
GarageType	81	0.055479
GarageYrBlt	81	0.055479
GarageFinish	81	0.055479
GarageQual	81	0.055479
BsmtExposure	38	0.026027
BsmtFinType2	38	0.026027
BsmtFinType1	37	0.025342
BsmtCond	37	0.025342
BsmtQual	37	0.025342
MasVnrArea	8	0.005479
MasVnrType	8	0.005479
Electrical	1	0.000685
Utilities	0	0.000000
YearRemodAdd	0	0.000000
MSSubClass	0	0.000000
Foundation	0	0.000000
ExterCond	0	0.000000
ExterQual	0	0.000000

If more than 15% of the data is missing, we should exclude the corresponding variable in our analysis. Variables 'PoolQC', 'MiscFeature', 'Alley', 'Fence', 'FireplaceQu', 'LotFrontage' will be excluded.

'GarageCond', 'GarageType', 'GarageYrBlt', 'GarageFinish' and 'GarageQual' have the same percentage of missing data. These should be for the same set of obervation. Since the number of car space is the most important factor for garage, we will take 'GarageCars' into our analysis and exclude the other Garage* with missing data.

'MasVnrArea' 'MasVnrType', 'BsmtExposure', 'BsmtFinType2', 'BsmtFinType1', 'BsmtCond' and 'BsmtQualwe' are not essential variabls. Furthermore, they have a strong correlation with 'YearBuilt', which we will consider. Thus, we will not lose information if we exclude these.

Finally, we have one missing observation in 'Electrical'. We'll delete this observation and keep the variable.

In summary, to handle missing data, we'll delete all the variables with missing data, except the variable 'Electrical'. In 'Electrical' we'll just delete the observation with missing data.

[7]:
```
#handling with missing data
df_train = df_train.drop((missing_data[missing_data['Total'] > 1]).index,1)
df_train = df_train.drop(df_train.loc[df_train['Electrical'].isnull()].index)
df_train.isnull().sum().max()
```

[7]:
```
0
```

[8]:
```
type(total)
```

[8]:
```
pandas.core.series.Series
```

1.3.3 2.3 Outliers

Outliers can markedly affect our models and can be a valuable source of information, providing us insights about specific behaviors. We'll just do a quick analysis through the standard deviation of 'SalePrice' and a set of scatter plots.

2.3.1 Univariate analysis The primary concern here is to establish a threshold that defines an observation as an outlier. To do so, we'll standardize the sale price data by converting data values to have mean of 0 and a standard deviation of 1.

[9]:
```
#standardizing data
saleprice_scaled = StandardScaler().fit_transform(df_train['SalePrice'][:,np.
 ↪newaxis]);
low_range = saleprice_scaled[saleprice_scaled[:,0].argsort()][:5]
high_range= saleprice_scaled[saleprice_scaled[:,0].argsort()][-5:]
print('outer range (low) of the distribution:')
print(low_range)
print('\nouter range (high) of the distribution:')
print(high_range)
```

```
outer range (low) of the distribution:
[[-1.83820775]
 [-1.83303414]
 [-1.80044422]
 [-1.78282123]
 [-1.77400974]]

outer range (high) of the distribution:
[[5.06034585]
 [5.42191907]
 [5.58987866]
 [7.10041987]
 [7.22629831]]
```

Low range values are similar and not too far from 0. High range values are far from 0 and two values larger than 7 are really out of range. For now, we'll not consider any of these values as an outlier but we should be careful with those two values larger than 7.

2.3.2 Bivariate analysis

[10]:
```
#bivariate analysis saleprice/grlivarea
var = 'GrLivArea'
data = pd.concat([df_train['SalePrice'], df_train[var]], axis=1)
data.plot.scatter(x=var, y='SalePrice', ylim=(0,800000));
```

The two values with bigger 'GrLivArea' seem strange and they are not following the crowd. Therefore, we'll define them as outliers and delete them. The two observations in the top of the plot look like two special cases, however they seem to be following the trend. We will keep them.

[11]:
```
#deleting points
df_train.sort_values(by = 'GrLivArea', ascending = False)[:2]
df_train = df_train.drop(df_train[df_train['Id'] == 1299].index)
df_train = df_train.drop(df_train[df_train['Id'] == 524].index)
```

[12]:
```
#bivariate analysis saleprice/grlivarea
var = 'TotalBsmtSF'
data = pd.concat([df_train['SalePrice'], df_train[var]], axis=1)
data.plot.scatter(x=var, y='SalePrice', ylim=(0,800000));
```

The ones with TotalBsmtSF > 3000 look like outliers. But we can keep them for now.

1.4 3. Data Story

Sales price descriptive statistics summary

[13]:
```
df_train['SalePrice'].describe()
```

[13]:
```
count      1457.000000
mean     180942.138641
std       79521.569966
min       34900.000000
25%      129900.000000
50%      163000.000000
75%      214000.000000
max      755000.000000
Name: SalePrice, dtype: float64
```

[14]:
```
#skewness and kurtosis
print("Skewness: %f" % df_train['SalePrice'].skew())
print("Kurtosis: %f" % df_train['SalePrice'].kurt())
```

```
Skewness: 1.880363
Kurtosis: 6.516048
```

The price is right Skewed. Distributions with kurtosis greater than 3 are said to be leptokurtic

[ ]:

[15]:
```
#histogram
sns.distplot(df_train['SalePrice']);
```

The histogram above shows an overview of the corresponding data set. The lowest sale price of a house is 34, 900 and the highest price is 755, 000. Majority of the sale prices are towards the cheaper side, with an average sale price of 180, 921. There are only a few sales prices that are over 500, 000.

[16]:
```
# Relationship with categorical features
#box plot overallqual/saleprice
var = 'OverallQual'
#data = pd.concat([df_train['SalePrice'], df_train[var]], axis=1)
print(data.head())
f, ax = plt.subplots(figsize=(8, 6))
fig = sns.boxplot(x=var, y="SalePrice", data=df_train)
fig.axis(ymin=0, ymax=800000);
```

```
   SalePrice  TotalBsmtSF
0     208500          856
1     181500         1262
2     223500          920
3     140000          756
4     250000         1145
```

The box plot above shows a plot of the individual sale price of a house versus the overall quality of each house. The gray horizontal line in each box plot represents the average sale price for each overall quality. On average, as the overall quality increases the sale price increases with it.

[17]:
```
data.head()
```

[17]:

	SalePrice	TotalBsmtSF
0	208500	856
1	181500	1262
2	223500	920
3	140000	756
4	250000	1145

[18]:
```
plt.rc('xtick', labelsize=10)
var = 'YearBuilt'
#data = pd.concat([df_train['SalePrice'], df_train[var]], axis=1)
f, ax = plt.subplots(figsize=(16, 8))
fig = sns.boxplot(x=var, y="SalePrice", data=df_train)
fig.axis(ymin=0, ymax=800000);
plt.xticks(rotation=90);
```

In summary Stories aside, we can conclude that: 'GrLivArea' and 'TotalBsmtSF' seem to be linearly related with 'SalePrice'. Both relationships are positive, which means that as one variable increases, the other also increases. In the case of 'TotalBsmtSF', we can see that the slope of the linear relationship is particularly high. 'OverallQual' and 'YearBuilt' also seem to be related with 'SalePrice'. The relationship seems to be stronger in the case of 'OverallQual', where the box plot shows how sales prices increase with the overall quality. We just analysed four variables, but there are many other that we should analyse. The trick here seems to be the choice of the right features (feature selection) and not the definition of complex relationships between them (feature engineering).

To explore the data, we will start with Correlation matrix,'SalePrice' correlation matrix, Scatter plots between the most correlated variables

[19]:
```
#### Correlation matrix (heatmap style)
```

[20]:
```
#correlation matrix
corrmat = df_train.corr()
f, ax = plt.subplots(figsize=(12, 9))
sns.heatmap(corrmat, vmax=.8, square=True);
```

This heatmap is a good way to get a quick overview.

At first sight, there are two red colored squares that get my attention. The first one refers to the 'TotalBsmtSF' and '1stFlrSF' variables, and the second one refers to the 'GarageX' variables. Both cases show how significant the correlation is between these variables. Actually, this correlation is so strong that it can indicate a situation of multicollinearity. If we think about these variables, we can conclude that they give almost the same information so multicollinearity really occurs. Heatmaps are great to detect this kind of situations and in problems dominated by feature selection, like ours, they are an essential tool.

Another thing that got my attention was the 'SalePrice' correlations. We can see our well-known 'GrLivArea', 'TotalBsmtSF', and 'OverallQual' , but we can also see many other variables that should be taken into account.

[21]:
```
#saleprice correlation matrix
k = 10 #number of variables for heatmap
cols = corrmat.nlargest(k, 'SalePrice')['SalePrice'].index
cm = np.corrcoef(df_train[cols].values.T)
sns.set(font_scale=1.25)
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f',␣
 ↪annot_kws={'size': 10}, yticklabels=cols.values, xticklabels=cols.values)
```

```
plt.show()
```

These are the variables most correlated with 'SalePrice':

- 'OverallQual', 'GrLivArea' and 'TotalBsmtSF' are strongly correlated with 'SalePrice'.
- 'GarageCars' and 'GarageArea' are also some of the most strongly correlated variables. However, as we discussed in the last sub-point, the number of cars that fit into the garage is a consequence of the garage area. 'GarageCars' and 'GarageArea' are like twin brothers. You'll never be able to distinguish them. Therefore, we just need one of these variables in our analysis and we can keep 'GarageCars' since its correlation with 'SalePrice' is higher.
- 'TotalBsmtSF' and '1stFloor' also seem to be twin brothers. We can keep 'TotalBsmtSF'.
- 'FullBath'
- 'TotRmsAbvGrd' and 'GrLivArea', twin brothers again.
- 'YearBuilt' is slightly correlated with 'SalePrice'.

Let's proceed to the scatter plots.
Scatter plots between 'SalePrice' and correlated variables

[22]:
```
#scatterplot
sns.set()
```

```
cols = ['SalePrice', 'OverallQual', 'GrLivArea', 'GarageCars',␣
 ↪'FullBath','TotalBsmtSF',  'YearBuilt']
sns.pairplot(df_train[cols], size = 2.5)
plt.show();
```

The pairplot above shows correlation between the sale price of each house, the overall quality (OverallQual), above grade living area in square feet (GrLivArea), total square feet of basement area (TotalBsmtSF), full bathrooms above grade (FullBath) and the original construction date (YearBuilt). In the first row it shows the correlation of the sales price towards the other features. Visually there is a linear increase between each feature and the sales price. Several graphs in the pairplot above show linear relationships.

[23]:
```
plt.figure(figsize= (15,15))
plt.xticks(rotation=45,fontsize=14)
```

```
plt.yticks(fontsize=14)
plt.xlabel('Neighborhood',fontsize=14)
plt.ylabel('SalePrice',fontsize=14)
my_order = df_train.groupby(by=["Neighborhood"])["SalePrice"].median().
 ↪sort_values(ascending=False).index
ax = sns.boxplot(x="Neighborhood", y="SalePrice", data=df_train, order=my_order)
```

The above box plot graph shows the sales prices for each of the neighborhoods in the given data. The neighborhood NridgHt has the highest average sales price. That said, the neighborhood NoRidge has three of the most expensive homes.

1.5 4. Inferential Statistics

[24]:
```
df_train[['SalePrice', 'BedroomAbvGr']].describe()
```

[24]:
```
            SalePrice  BedroomAbvGr
count     1457.000000   1457.000000
mean    180942.138641      2.866163
std      79521.569966      0.816595
min      34900.000000      0.000000
25%     129900.000000      2.000000
50%     163000.000000      3.000000
75%     214000.000000      3.000000
max     755000.000000      8.000000
```

[27]:
```
# Split BedroomAbvGr into two different groups. The mean number of bedrooms is␣
 ↪2.87 so it is best to group homes that have 2 bedrooms and 3 bedrooms above␣
 ↪grade.
two_bedroom = df_train.SalePrice.loc[df_train.BedroomAbvGr == 2]
three_bedroom = df_train.SalePrice.loc[df_train.BedroomAbvGr == 3]
# Calculate T-Test for the two independent variables
import scipy
scipy.stats.ttest_ind(two_bedroom, three_bedroom)
```

[27]:
```
Ttest_indResult(statistic=-5.247681020072384, pvalue=1.8309779204141817e-07)
```

1.6 5. Machine Learning

[28]:
```
from sklearn import datasets, linear_model
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt
```

[31]:
```
df = df_train
# Convert y to log scale and drop SalePrice from the data set
y = np.log(df_train.SalePrice)
df.drop(['SalePrice'], axis = 1, inplace = True)

# Convert data into dummy variables
final_features = pd.get_dummies(df).reset_index(drop=True)
final_features
final_features.shape
# Split into train and test subsets
X_train, X_test, y_train, y_test = train_test_split(final_features, y,␣
 ↪test_size=0.2, random_state=42)
# Scale features using robust scaler
from sklearn.preprocessing import RobustScaler
transformaer = RobustScaler().fit(X_train)
```

[32]:
```
# Linear Regression Model
from sklearn import linear_model
```

```
from sklearn import metrics
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_squared_log_error

model = linear_model.LinearRegression()

model.fit(X_train, y_train)

training_accuracy = model.score(X_train, y_train)


pred = model.predict(X_test)
lr_rmse = np.sqrt(metrics.mean_squared_error(y_test, pred))
r_squared = r2_score(y_test ,pred)
mean_squared = mean_squared_error(y_test, pred)
root_mean_squared_log_error = np.sqrt(mean_squared_log_error(y_test, pred))

print('Training accuracy:', str(training_accuracy))
print('Root-mean-squared error:', str(lr_rmse))
print('Root-mean-squared-log-error:', str(root_mean_squared_log_error))
print('Mean-squared-error:', str(mean_squared))
print('R-squared:', str(r_squared))
```

```
Training accuracy: 0.9468820305207895
Root-mean-squared error: 0.1312175298616099
Root-mean-squared-log-error: 0.010307182597766203
Mean-squared-error: 0.017218040142982487
R-squared: 0.9007681319041584
```

[33]:
```
# Random Forest Model
from sklearn.ensemble import RandomForestRegressor


random_forrest = RandomForestRegressor()
random_forrest.fit(X_train, y_train)

pred = random_forrest.predict(X_test)
rf_rmse = np.sqrt(metrics.mean_squared_error(y_test, pred))
r_squared = r2_score(y_test ,pred)
mean_squared = mean_squared_error(y_test, pred)
root_mean_squared_log_error = np.sqrt(mean_squared_log_error(y_test, pred))

print('Training accuracy:', str(training_accuracy))
print('Root-mean-squared error:', str(rf_rmse))
print('Root-mean-squared-log-error:', str(root_mean_squared_log_error))
print('Mean-squared-error:', str(mean_squared))
```

```
print('R-squared:', str(r_squared))
```

```
Training accuracy: 0.9468820305207895
Root-mean-squared error: 0.1524727128580114
Root-mean-squared-log-error: 0.011936371204325901
Mean-squared-error: 0.023247928166281594
R-squared: 0.8660163803696155
```

[34]:
```
# Decision Tree Regressor Model
from sklearn.tree import DecisionTreeRegressor


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

pred = decision_tree_regressor.predict(X_test)
dtr_rmse = np.sqrt(metrics.mean_squared_error(pred, y_test))
r_squared = r2_score(y_test ,pred)
mean_squared = mean_squared_error(y_test, pred)
root_mean_squared_log_error = np.sqrt(mean_squared_log_error(y_test, pred))

print('Training accuracy:', str(training_accuracy))
print('Root-mean-squared error:', str(dtr_rmse))
print('Root-mean-squared-log-error:', str(root_mean_squared_log_error))
print('Mean-squared-error:', str(mean_squared))
print('R-squared:', str(r_squared))
```

```
Training accuracy: 0.9468820305207895
Root-mean-squared error: 0.21201792894590274
Root-mean-squared-log-error: 0.01660835621457093
Mean-squared-error: 0.04495160219450986
R-squared: 0.740932683242677
```

[35]:
```
# Extra Trees Regressor Model
from sklearn.ensemble import ExtraTreesRegressor

extra_trees_regressor = ExtraTreesRegressor()
extra_trees_regressor.fit(X_train, y_train)

pred = extra_trees_regressor.predict(X_test)
xtr_rmse = np.sqrt(metrics.mean_squared_error(pred, y_test))
r_squared = r2_score(y_test ,pred)
mean_squared = mean_squared_error(y_test, pred)
root_mean_squared_log_error = np.sqrt(mean_squared_log_error(y_test, pred))

print('Training accuracy:', str(training_accuracy))
print('Root-mean-squared error:', str(xtr_rmse))
```

```
print('Root-mean-squared-log-error:', str(root_mean_squared_log_error))
print('Mean-squared-error:', str(mean_squared))
print('R-squared:', str(r_squared))
```

```
Training accuracy: 0.9468820305207895
Root-mean-squared error: 0.1451457451357008
Root-mean-squared-log-error: 0.011318550922367422
Mean-squared-error: 0.021067287330997816
R-squared: 0.8785839584408909
```

[36]:
```
# Gradient Boosting Regressor Model
from sklearn.ensemble import GradientBoostingRegressor

gradient_boosting_regressor = GradientBoostingRegressor()
gradient_boosting_regressor.fit(X_train, y_train)

pred = gradient_boosting_regressor.predict(X_test)
gbr_rmse = np.sqrt(metrics.mean_squared_error(pred, y_test))
r_squared = r2_score(y_test ,pred)
mean_squared = mean_squared_error(y_test, pred)
root_mean_squared_log_error = np.sqrt(mean_squared_log_error(y_test, pred))

print('Training accuracy:', str(training_accuracy))
print('Root-mean-squared error:', str(gbr_rmse))
print('Root-mean-squared-log-error:', str(root_mean_squared_log_error))
print('Mean-squared-error:', str(mean_squared))
print('R-squared:', str(r_squared))
```

```
Training accuracy: 0.9468820305207895
Root-mean-squared error: 0.12233737615650019
Root-mean-squared-log-error: 0.009638522393906218
Mean-squared-error: 0.01496643360485702
R-squared: 0.9137447030550895
```

[37]:
```
# Bar plot for each of the following root-mean-squared-log-error
df = pd.DataFrame({'Algorithm':['LR_RMSLE', 'RF_RMSLE', 'DTR_RMSLE',␣
 ↪'XTR_RMSLE', 'GBR_RMSLE'], 'Root-mean-squared-log-error':[0.0108, 0.0124, 0.
 ↪0161, 0.0120, 0.0100]})
ax = df.plot.bar(x='Algorithm', y='Root-mean-squared-log-error', rot=45)
```

The above bar plot shows the root-mean-squared-errors for the following algorithms. Gradient boosting gave the best result of 0.127 and the decision tree regressor gave the worst result of 0.191.

[ ]:
```
# Hyperparameter tuning for gradient boosting regressor
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
gbr=GradientBoostingRegressor()
search_grid=[{'n_estimators':[500,1000,2000],'learning_rate':[.001,0.01,.
 ↪1],'max_depth':[1,2,4],'subsample':[.5,.75,1]}]
search1=GridSearchCV(estimator=gbr,param_grid=search_grid,n_jobs=1,cv=5)
search1.fit(X_train, y_train)
```

[ ]:
```
# Gradient Boosting Regressor after hyperparameter tuning
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_log_error

pred = search1.predict(X_test)
gbr_rmse = np.sqrt(metrics.mean_squared_error(pred, y_test))
r_squared = r2_score(y_test ,pred)
mean_squared = mean_squared_error(y_test, pred)
root_mean_squared_log_error = np.sqrt(mean_squared_log_error(y_test, pred))
```

```
print('Training accuracy:', str(training_accuracy))
print('Root-mean-squared error:', str(gbr_rmse))
print('Root-mean-squared-log-error:', str(root_mean_squared_log_error))
print('Mean-squared-error:', str(mean_squared))
print('R-squared:', str(r_squared))
```

[ ]:
```
# Top 20 feature importances
feature_importances = pd.DataFrame(gradient_boosting_regressor.
 ↪feature_importances_,index = X_train.columns,columns=['importance']).
 ↪sort_values('importance',ascending=False)
feature_importances = feature_importances[0:20]
feature_importances
```

[ ]:
```
# Bar plot for feature importances
feature_importances = pd.DataFrame(gradient_boosting_regressor.
 ↪feature_importances_,index = X_train.columns,columns=['importance']).
 ↪sort_values('importance',ascending=False)

N = 20
ax = (feature_importances.iloc[0:N][::-1]
    .plot(kind='barh',
          title='Feature Importances',
          figsize=(10, 6)))
ax.grid(False, axis='y')
```

1.6.1 Conclusion

The best RMSLE (root-mean-squared-Logarithmic-error) was respectively 0.01002 which was obtained from the Gradient Boosting Regressor. After hyperparameter tuning the Gradient Boosting Regressor model, the RMSLE was 0.00936.

[ ]:
```
```