Joint Project Report

Predicting Ames Housing Prices using Machine Learning with Decision Tree Modelling

---

A 24-Hour Project by

Oshton Tsen & Edmond Nemsingh

Submitted in fulfilment of the requirements for a feature engineering competition by Kaggle Inc.

December 2018

Table of Contents

1. Abstract

The objective of this briefing is to showcase common techniques in the field of Machine Learning using the Ames Housing dataset, compiled by Dean De Cock in 2011 as a modern replacement to the Boston Housing dataset. This briefing showcases the Gradient Boosting Regressor, Random Forest Regressor, and Decision Tree Regressor models and their ability to predict sale price with features kept constant.

Specifically, the aim was to minimize the Mean Absolute Error (MAE) of the predictions. In order to process the dataset, its spatial structure was exploited in order to optimize feature engineering efforts. Once the features were finalized, the machine learning algorithms were tuned to a training dataset and cross validated. The most notable results were achieved by the XGBRegressor model, which demonstrated that the approach used for the problem was largely successful. Most importantly, the model produces predictions that are competitive to other housing price prediction models.

2. Introduction: Preprocessing

The provided dataset is comprised of numerical and categorical values. During the analysis of the dataset, the fundamental principles of feature engineering were used in order to optimize the given dataset for Machine Learning. To better visualize the steps taken to develop the statistical model, the following work flow diagram depicts each stage of the process.

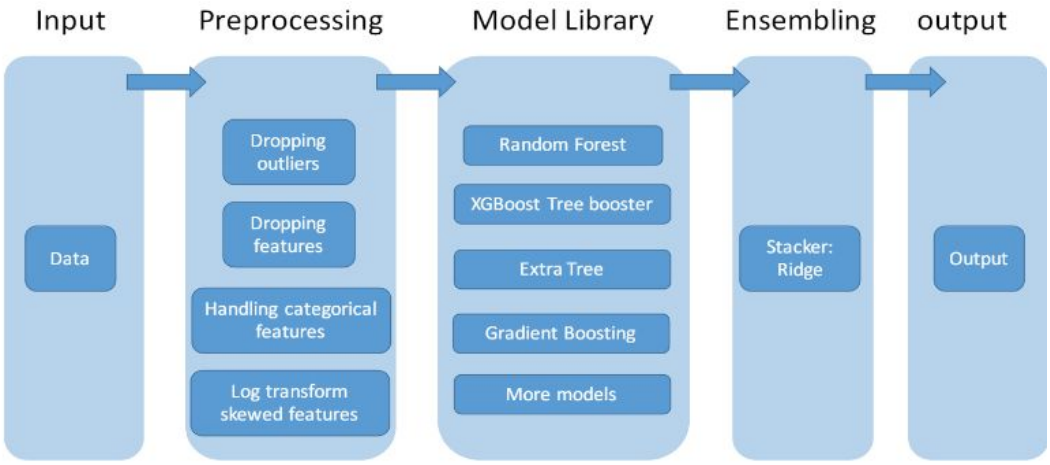


Figure 1. Feature Engineering Work Flow Diagram

After reading the .csv file containing the Ames Housing Dataset, the train and test data were merged and one-hot encoded together. One hot encoding is the process of turning categorically inputted data into boolean values so that a machine learning algorithm is better

able to decipher. For this process, singular columns were split into *n* different features, where *n* is equivalent to the amount of unique entries within the original feature. One complication that needed to be accounted for in this dataset was the discrepancy between the *n*-value for the train & test dataset. In other words, the categorical column values may exist in the training data set, but not in the test data set. To address this issue, joining the two datasets prevented data misalignment. To rectify this, we temporarily merged the datasets, one hot encoded the data to ensure the same n-value then re-separated the data at the same point using the Id column. Below is an example of how the one-hot encoding process was executed for one category:

SaleType: Type of sale
WD Warranty Deed - Conventional
CWD Warranty Deed - Cash
VWD Warranty Deed - VA Loan

WD Warranty Deed - Conventional	CWD Warranty Deed - Cash	VWD Warranty Deed - VA Loan
0	1	0
1	0	0
0	0	1
1	0	0

Figure 2. One-Hot Encoding Process. Houses with the a specific type of warranty deed are indicated with a *1* for true, while houses that did not fall into a category are indicated with a *0* for false.

```
train_file_path = '.../train.csv'
test_file_path = '.../test.csv'
combined_file_path = '.../traintest.csv'
housing_data = pd.read_csv(combined_file_path)


#fill NA in the LotFrontage column with the average value
housing_data["LotFrontage"].fillna(housing_data.LotFrontage.mean(), inplace = True)
now = datetime.datetime.now()

coly = housing_data.columns.to_series().groupby(housing_data.dtypes).groups
n_vars = list(housing_data.dtypes[housing_data.dtypes!="object"].index)
to_remove = ['Id', 'MSSubClass', 'OverallQual', 'OverallCond', 'YearBuilt','YearRemodAdd',
             'GarageYrBlt', 'MoSold', 'YrSold']
for item in to_remove:
    n_vars.remove(item)
c_vars = list(housing_data.dtypes[housing_data.dtypes == "object"].index)
c_vars.extend(('OverallQual', 'OverallCond', #'YearBuilt','YearRemodAdd', 'GarageYrBlt'
                            'MoSold', 'YrSold'))
```

```
for variable in c_vars:
    dummies = pd.get_dummies(housing_data[variable], prefix=variable)
    housing_data = pd.concat([housing_data, dummies], axis = 1)
    housing_data.drop([variable], axis=1, inplace = True)

(...)

housing_data.to_csv('OneHotEncodedHousingData.csv', index = False)
#Split up OneHotEncoded File Manually then continue yourself.
```

Next, a statistical summary of the data set was generated in Pandas to better visualize the data.

	Id	MSSubClass	...	YrSold	SalePrice
count	1460.000000	1460.000000	...	1460.000000	1460.000000
mean	730.500000	56.897260	...	2007.815753	180921.195890
std	421.610009	42.300571	...	1.328095	79442.502883
min	1.000000	20.000000	...	2006.000000	34900.000000
25%	365.750000	20.000000	...	2007.000000	129975.000000
50%	730.500000	50.000000	...	2008.000000	163000.000000
75%	1095.250000	70.000000	...	2009.000000	214000.000000
max	1460.000000	190.000000	...	2010.000000	755000.000000

[8 rows x 38 columns]

Figure 3a. Statistical summary outputted to the console

Index	Id	MSSubClass	LotFrontage	LotArea
count	1460	1460	1201	1460
mean	730.5	56.8973	70.05	10516.8
std	421.61	42.3006	24.2848	9981.26
min	1	20	21	1300
25%	365.75	20	59	7553.5
50%	730.5	50	69	9478.5
75%	1095.25	70	80	11601.5
max	1460	190	313	215245

Figure 3b. Statistical diagram in table format

Based on the above information, the dataset is comprised of 1,460 houses that were sold between the years 2006 and 2010. The describes each house across 79 features including sales price.. Furthermore, the standard deviation of housing sales prices is approximately $79,443. According to the generated box plot and the statistical summary above, the data is skewed to the right since all the outliers are on the right side and the median is less than the mean. Additional box and whisker plots were constructed for each numerical feature. These can be found in the results section.

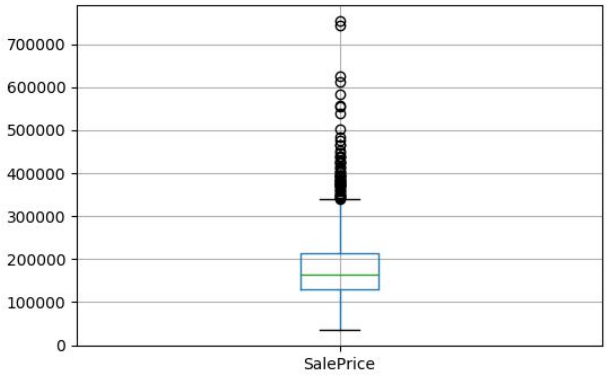


Figure 4. Box and Whisker Plot for the sales prices of homes

```
In [11]: correlationmatrixdata.LotFrontage.describe()
Out[11]:
count    1201.000000
mean       70.049958
std        24.284752
min        21.000000
25%        59.000000
50%        69.000000
75%        80.000000
max       313.000000
Name: LotFrontage, dtype: float64
```

From this data, it became noticeable that *LotFrontage* was a category that had a significant lack of data entries (1201/1460 data points). Since the variable *LotFrontage* represents the linear feet of street connected to property, it can be classified as quantitative data rather than categorical data. According to Dan Becker, Head of Kaggle Learning and previous Machine Learning professor at Johns Hopkins, complicated methodologies of imputation do not necessarily increase your predictive power especially when a majority of the data is given. In the case of *LotFrontage*, only 17% of the values are missing, thus imputing the missing values with the mean does not sacrifice much if any of the model's predictive power. Doing so, is likely to be more beneficial to the model accuracy instead of eliminating *LotFrontage* entirely from the dataset features. On the other hand, the MSSubClass category was excluded from the dataset because it was effectively an umbrella variable that combines other features in the dataset under one column. The features that *MSSubclass* is derived from include: Amount of stories, building style, building age based on initial build date, and attic presence. Accordingly, the one hot encoding process captured the information in this redundant variable. A broad brush approach was taken for preprocessing and as many variables were kept. For that reason, *MSSubClass* was the only singular feature of the dataset not utilized for reasons stated earlier.

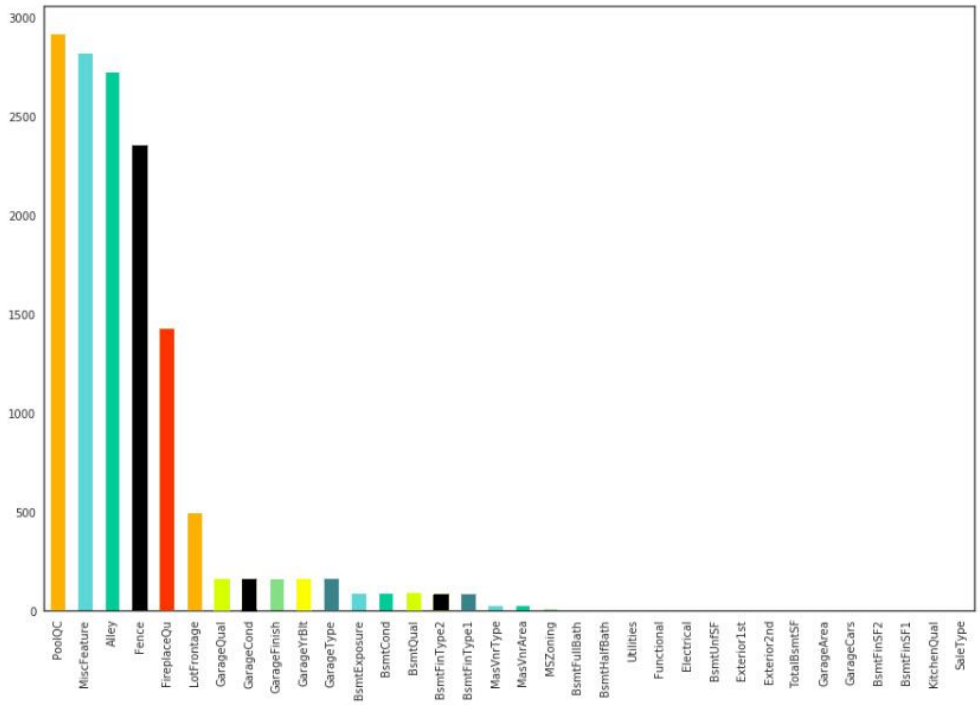


Figure 5. Bar graph revealing the number of missing values for each column (including *NA*)

Although datasets often use *NA* to indicate a missing value, further analysis of the Boston Housing dataset revealed that many of the *NA* values are legitimate. Many of the features in the dataset, using *NA*, do so to indicate whether or not the feature was part of the home. For example, the *PoolQuality* feature contains several *NA*'s. Further inspection revealed that these homes also have no *PoolSize* indicating that *NA*'s did not indicate that the data was unknown or missing, but rather it indicates that the homes did not have pools. A similar logic could be applied to other columns consisting of *NA* values.

In order to verify the model quality, the mean absolute error (*MAE*) was calculated. This evaluation metric can be evaluated with the following equation:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i| \qquad \textbf{(Eqn. 1)}$$

Where *n* is the total number of datapoints and (X_i, y_i) is the coordinate for a point i. In the context of the Ames Housing Dataset, $\hat{y}_i$ is defined to be a predicted housing price and y_i is the actual housing price.

3. Models and Methods

Statistical models offer an efficient method of making predictive decisions using a big set of data. This section will explore three different machine learning methodologies: decision tree, random forest, and XGBoost.

3.1. Decision Tree

The decision tree model is a series of if/else questions, in which the goal of the model is to get to the correct answers in as few stages as possible or minimum depth. It makes it choices by minimizing the SSE (Sum of Squared Errors) at each split/stage. This simple approach doesn't generate as much predictive power as some other models but decision trees are easily to visualize & are easy to explain to nonexperts.

3.2. Random Forest

The Random Forest Regressor is one that is widely used in the field of Machine Learning mainly due to its reliability and high predictive power. The model takes the Decision Tree model a step further. Known as an ensemble model, Random Forest combine a collection of decision trees to reduce the overfitting present in the single Decision Tree model. By creating different decision trees which each overfit in their own way, a model that is much more robust is created.

3.3. XGBoost

A relatively new implementation of the Gradient Boosting model (December 2017), the model has taken the Machine Learning world by storm due to its higher predictive power when compared to the Random Forest Model and ease of tuning. Note that this model was also the primary model used in the project. Gradient boosted machine learning combines multiple decision trees to create a superior model for standard tabular data. The core of gradient boosting utilizes shallow decision trees which provide good predictions on a small part of the data, more and more trees are then iteratively added to improve performance.

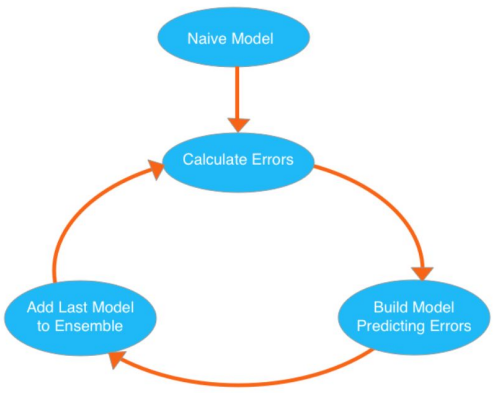


Figure 6. Parameters used for XGBoost modeling

As indicated in **Figure 6**, an initial under performing model is built and the associated errors are calculated. A second model is built to discover the correlations between the features and error. From each iteration of the cycle, the model gains new predictions about the dataset, which are then used to build newer models with additional insight on the data. All predictions are accumulated in what is called an ensemble of models. Before exhibiting these cycles, however, several parameters were specified for XGBoost. Each parameter controls a particular facet of the model's performance. The parameters used for this model are listed out below.

XGBoost Parameters	
Parameter/Value Used	**Definition**
n_estimators = 1000	Specifies the number of times to run through the modeling cycle
early_stopping_rounds = 5	Specifies how many duplicate values must be produced sequentially for the program to stop before n_estimators is reached.
learning_rate = 0.05	Scales the values that are used to train the model, so that the results produced in later model cycles have just as much impact to the model as the earlier cycle results.
n_jobs = -1	Determines the number of cores the machine uses to run the program. (-1 indicates the maximum number of cores)

Figure 7. Parameters used for XGBoost modeling

A *for* loop was created to try various values for each parameter. The loop iterates through a specified range and return the parameter values that produced the highest accuracy. These values are found in the left column of **Figure 7**.

3.4. Cross Validation

"Cross-Validation is a statistical method of evaluating generalization performance that is more stable and thorough than using a split into a training and test set." Under the cross-validation method, the data is split into k-folds (usually 5 or 10) and trained and tested repeatedly. This provides *k* accuracy values which are then aggregated into an average Mean Absolute Error.

This methodology provides us with a more robust training process and allows us to make more of the data that we have. The main interest is how the model performs against new, previously unseen data. While a model's ability to adapt to the training data is interesting, it's ability to predict new, previously unseen data is the main focus.

With cross-validation the data set can effectively be expanded for usage multiple times. This provides a way to more accurately assess generalization performance that go beyond the default measures of accuracy provided by the *score* method. Cross validation naturally implements a process that inherently includes multiple experiments without a requirement for more data unlike the *train-test split* method which is effectively a single experiment.

For this demonstration, we use a k-value of 5 for all models utilized.

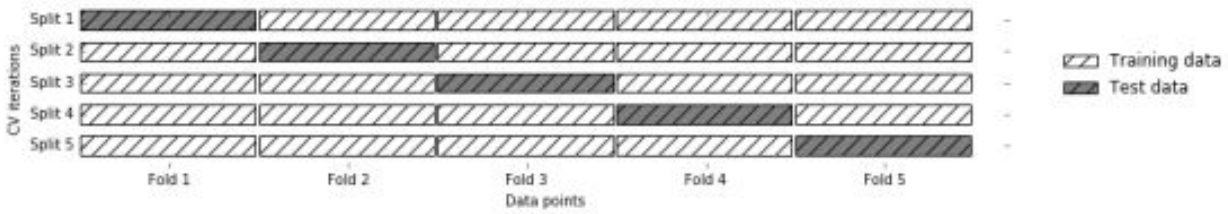
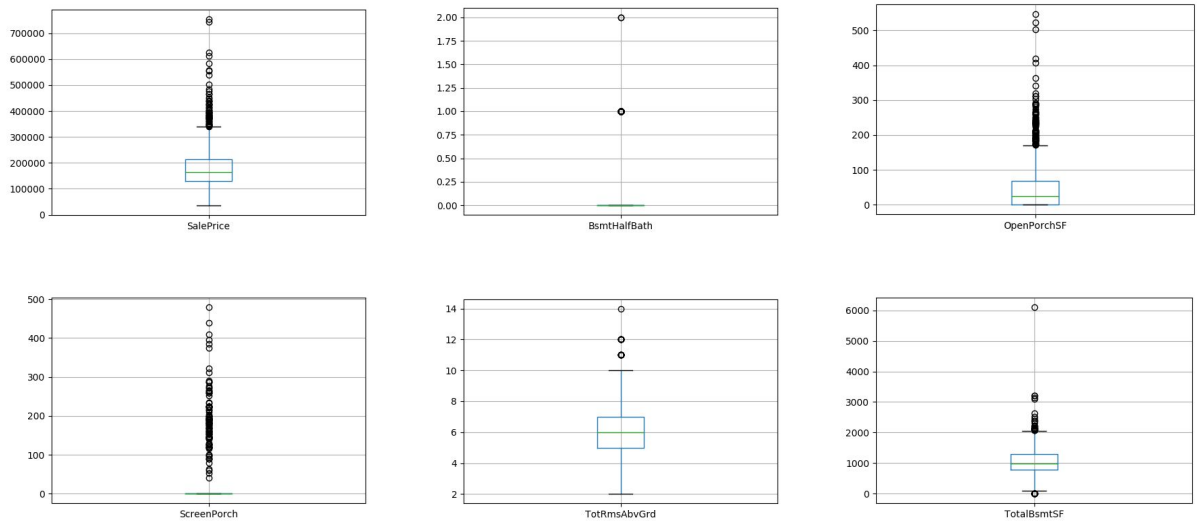
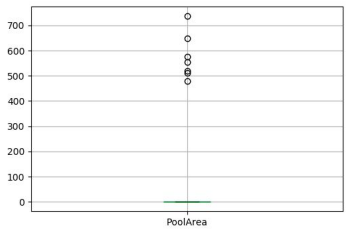
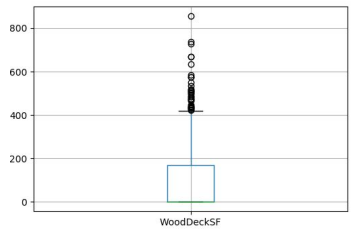


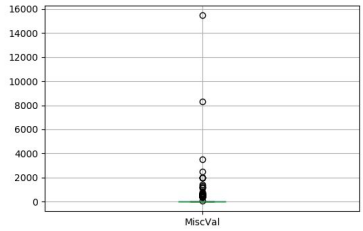
Figure 8. Cross validation diagram

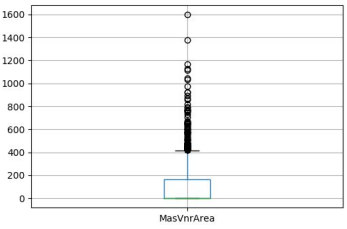
4. Results

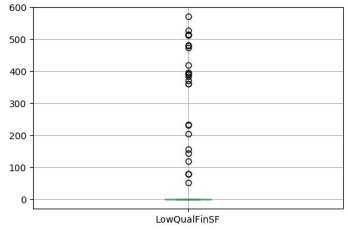


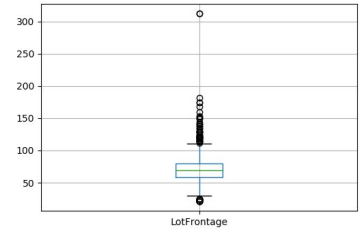

PoolArea

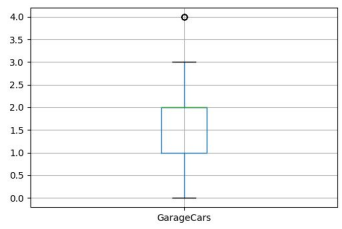

WoodDeckSF

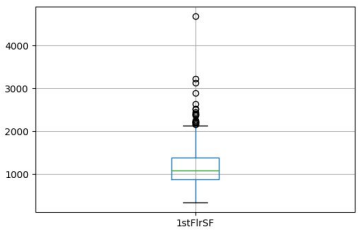

MiscVal

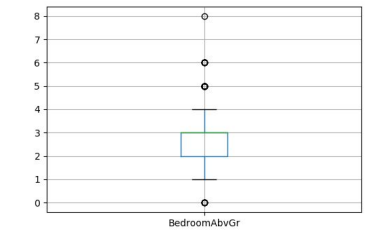

MasVnrArea

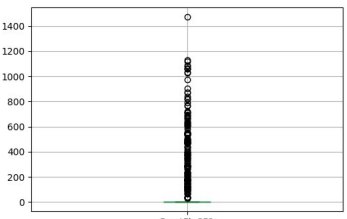

LowQualFinSF

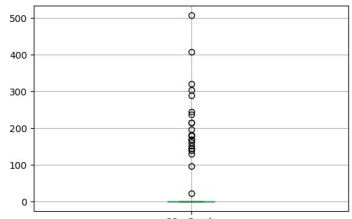

LotFrontage

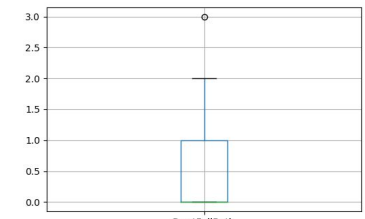

GarageCars

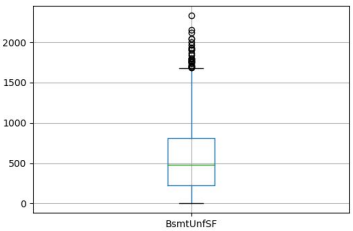

1stFlrSF

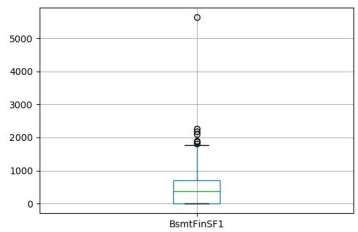

BedroomAbvGr

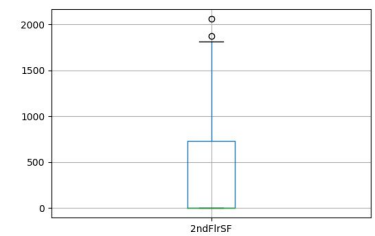

BsmtFinSF2


3SsnPorch


BsmtFullBath


BsmtUnfSF


BsmtFinSF1


2ndFlrSF

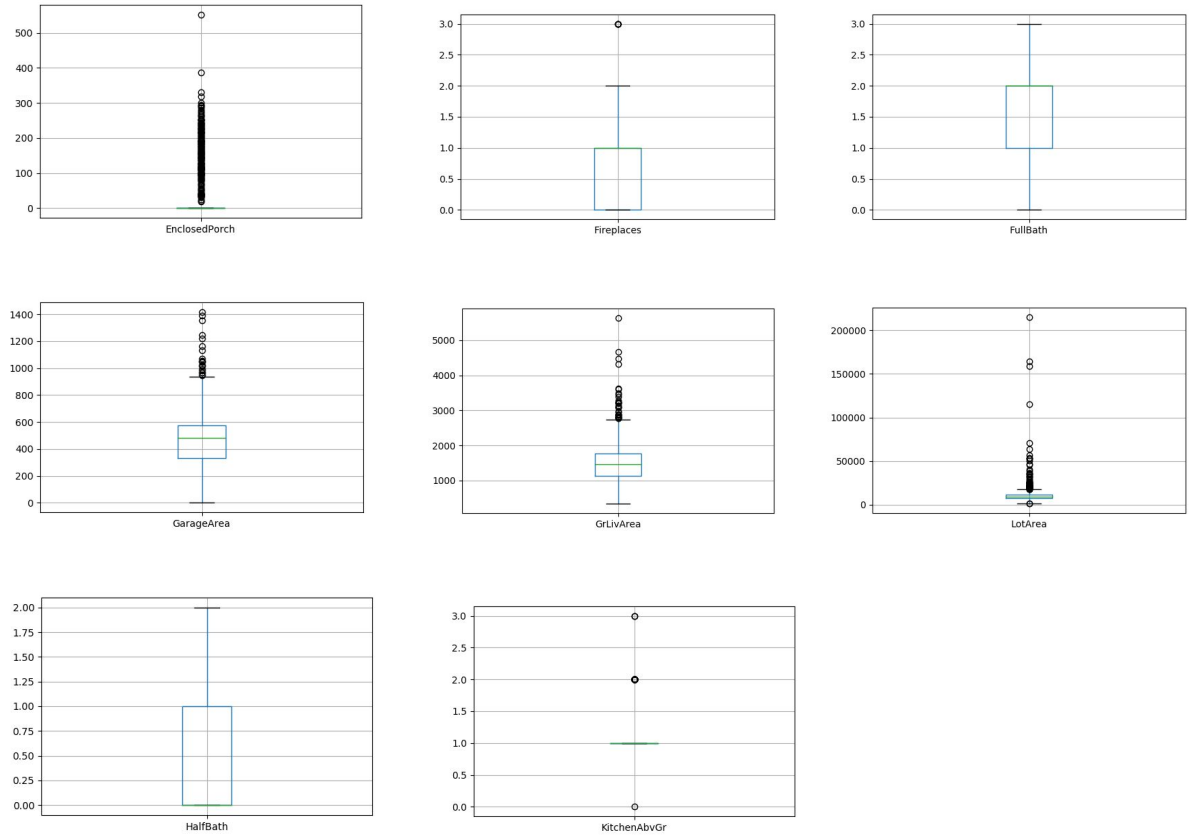


Figure 9. Box and Whisker plots for each numerical feature

4.1. Decision Tree

Using the Decision Tree model has an average Mean Absolute Error of 27,000.926. On average, this model made predictions that are 27,000 off from the true Sale Price value during the cross-validation process. This model's results are in-line with why this model has fallen out of favor within the Machine Learning community when compared to more complex ensemble models like Random Forest & Gradient Boosting.

4.2. Random Forest

Using the Random Forest Regressor model has an average Mean Absolute Error of 19663.795. On average, this model made predictions that are 19,663 off from the true Sale Price value during the cross-validation process. This model produces predictions that are 37.31% more accurate than the decision tree model on average using the same scoring methodology.

4.3. XGBoost

Using the Extreme Gradient Booster model has an average Mean Absolute Error of 15986.73. On average, this model made predictions that are 15,987 off from the true Sale Price value during the cross-validation process. On average, Extreme Gradient Boosting produces an average *MAE* that is 41.22% more accurate than the Decision Tree Model using the same scoring methodology.

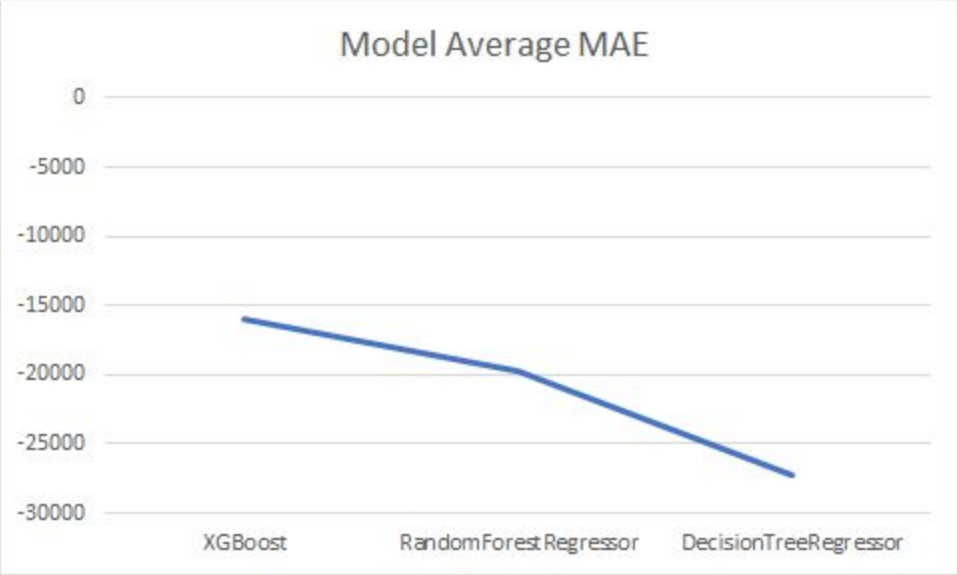


Figure 10. This graph depicts the average *MAE* of each model used.

5. Discussion

Judgin from all the results obtained from the 3 statistical models, the most accurate is Gradient Boost. Although theis is the best, it had a noticeably longer run time.

In performing this analysis, various statistical models were built, each offering a different perspective in making predictions on the Boston housing prices. The below sections explain the advantages and disadvantages to each model.

5.1. Decision Tree Model Pros & Cons

Basic decision trees still find much use within the Machine Learning community even if more modern ensemble models have overshadowed it. Decision Trees have the advantage of being very interpretable without external partial dependence plots being required since it only creates one tree that can be viewed when the model is finished fitting. Since this method of modelling is much simple a good amount of accuracy is sacrificed as a result.

5.2. Random Forest Tree Model Pros & Cons

Random Forest is an ensemble model that utilizes multiple decision trees to fit to a dataset. It's main advantage is that it is power & accurate with a small number of data points, it also performs very well on many prediction scenarios. Random Forest builds its decision trees independently using subsets of the feature list to develop individual trees fully.

It's main disadvantage are its black box nature, while you can get the feature importances it doesn't provide a natural relationship like a generalized linear model or a decision tree. Overfitting can easily occur in a Random Forest Model (what likely happened in our demonstration), the amount of trees needs to be tuned by the user in order to provide the maximum predictive power.

When the maximum amount of performance is required and a sacrifice to interpretability is acceptable, the Random Forest is what should be utilized.

5.3. Gradient Boosting Model Pros & Cons

Gradient Boosting is similar to the Random Forest model. Both the models are ensemble models utilizing multiple decision trees for outcome prediction. Gradient Boosting however build relatively shallow trees one at a time on the whole dataset, where each iterative tree improves the whole of the model.

One advantage of gradient boosting has been generally shown to be a more accurate model than the Random Forest model. This comes with a sacrifice to the run-time of the model since the model has to be built sequentially. Gradient Boosting has the same interpretability sacrifices that Random Forest makes, since all you can use for interpretation at the end are feature importances & partial dependence plots.

Gradient Boosting is also more prone to overfitting than the Random Forest model but there are methods to curtail this overfitting such as assigning a learning rate (forcing shrinkage) and maximizing tree depth.

6. Future Improvements

Due to the limited amount of time for the project, the dataset could not be preprocessed even more to further improve model quality. One of these improvements could be fitting the dataset to a correlation matrix. The correlation matrix is a heavily used method in the machine learning community because it summarizes scatterplots with the correlation. Each scatter plot should be describing the graphical behavior when one feature plotted against another. A line of best fit, via sum of least squares, is added to the plotted data points and the slope is calculated.

A positive trending slope is reflected by a positive correlation coefficient, and likewise a negative trending slope is reflected by a negative correlation coefficient. Asides from the sign of the correlation coefficient, the value is determined by measuring the distances of the data points from the line of best fit and comparing it with the degree of scattering in the y-axis. A large scatter in the y-axis and a small scatter around the fitted line leads to a strong correlation coefficient, while the opposite of this statement leads a weak correlation matrix. Note that a coefficient of 0 indicates that the plot has a stagnant behavior throughout. The closer to 1.0 the magnitude of the coefficient is, the stronger the correlation between the two features.

Using this data to locate codependent/redundant variables (confounding effect on models), could have helped for feature deletion. In the same vein, using correlation against the selling price gives a quick look at what the key variables are.

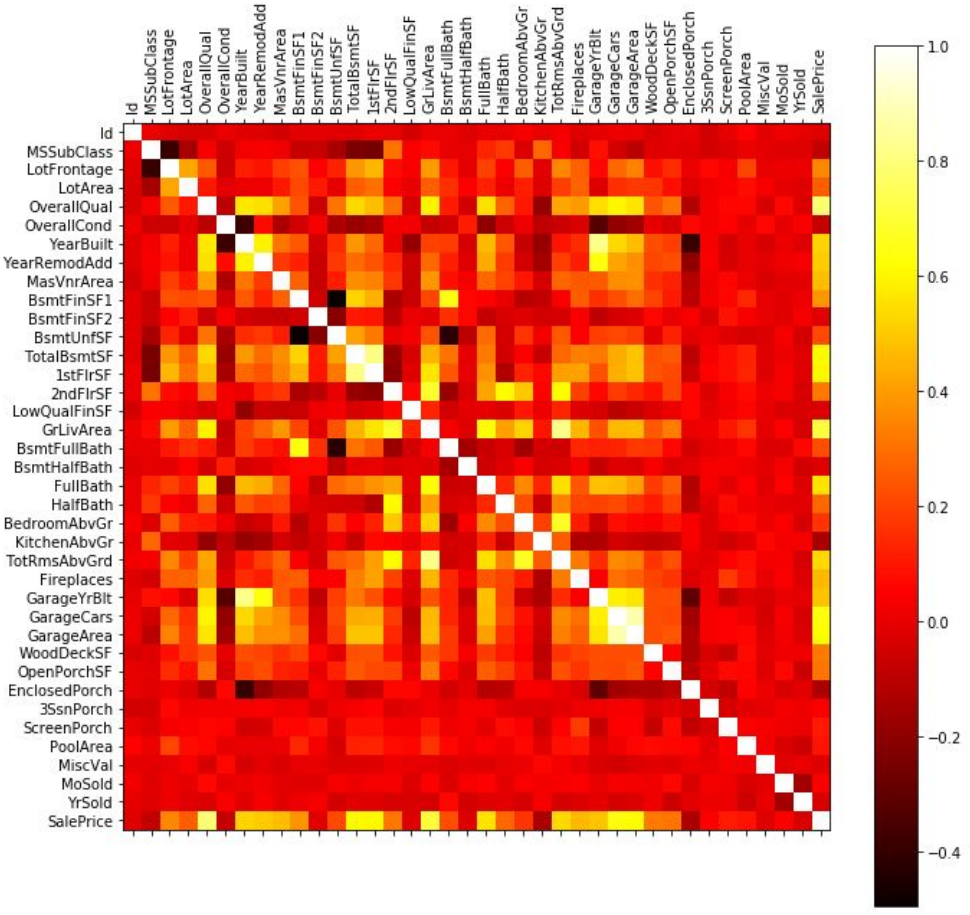


Figure 11. Correlation Map describing all 72 features

The following function was developed in order to produce the above correlation matrix:

```
İmport matplotlib as plt
import pandas as pd

#This function plots a graphical correlation matrix for each pair of columns in the dataframe.
```

```
#The inputs are df: pandas DataFrame and size: vertical & horizontal size of the plot
def plot_corr(data):
    corr = data.corr()
    fig, ax = plt.subplots(figsize=(len(correlationmatrixdata.columns), len(correlationmatrixdata.columns)))
    ax.matshow(corr)
    plt.xticks(range(len(corr.columns)), corr.columns, rotation = 90);
    plt.yticks(range(len(corr.columns)), corr.columns);
    plt.imshow(data.corr(), cmap='hot', interpolation = 'nearest')
    plt.colorbar()
    fig.set_size_inches(len(correlationmatrixdata.columns)/8,len(correlationmatrixdata.columns)/8)
    plt.savefig('correlationmatrix.png', dpi = 100)
```

Outliers & Training Dataset: Analysis

One mistake that was made when training the model, is the failure to account for outliers during the training segment of the process. In many documented demonstrations of this housing model, many participants elect to remove outliers during the preprocessing phase since the accuracy gained on outlier predictions was not worth the accuracy drop on the average element.

Additionally, the following algorithms can be incorporated in order to further improve the predictive capabilities of the model:

Boruta Algorithm for feature selection

The Boruta Algorithm is a powerful algorithm that can be used to select features. It works by comparing a feature to a cloned feature that randomly orders the elements. By comparing the correlation of the original X ordering to the randomly assigned order, you can see if the original column has more predictive power than random selection.

Stacking Algorithms for Higher Predictive Power

Many Kaggle Inc. winners utilize multiple models. By training and fitting multiple models one can then use the predictions that are generated as features for a new model . This tiered “brick-laying” approach is robust and highly predictive. When running our demonstration, we did it from a blind point of view taking in only what we knew how to do, and researching methods of improvement after the first submittal.