

Machine Learning Engineer Nanodegree

Capstone Project: [House Prices: Advanced Regression Techniques](#)

Rishabh Chopra
November 19, 2017

I. Definition:

Project Overview:

The task here is to build a model of housing prices in Ames, Iowa using the Ames Housing Dataset.



The author of this data, Dean De Cock grew unsatisfied with the Boston Housing Dataset which was used as an end semester regression project for his students. The Boston Housing dataset is from the 70's and the housing prices have become unrealistic for today's market. Even inflating the prices does not seem right as it'll make the data from "real to realistic", as said by him.

The "**SalePrice**" prediction can be used for multiple purposes by a real estate agent or even by a firm for investment analysis.

Without machine learning, the house prices are usually predicted by experts using complex rules. This is neither cost, nor time - effective. Furthermore, experts in the field cannot judge every aspect of the house. This is where Machine Learning comes in.

This is a *supervised multivariate regression* problem. *Supervised*, meaning we are given a set of *labeled* training examples. *Multivariate*, meaning we are given multiple features about the house. *Regression* refers to predicting a *target* numeric value, such as the price of the house, given a set of *features*, such as the age, area, etc. of the house.

Problem Statement:

Given are 79 explanatory describing the quality and quantity of many physical attributes of residential homes in Ames, Iowa. Our aim is to train a supervised multivariate regression model to predict accurately, the sale prices of new houses not seen before by the model.

Concretely, given the features, we want to predict a selling price for the house, such that the deviation between the predicted price (\hat{y}) and the actual price(y) is minimal,

i.e. $\min(y - \hat{y})$.

The dataset was retrieved from the Ames City Assessor's Office and further edited and curated by Dead De Cock. The train file has 1460 observations and the test file has 1459 observations.

The tasks involve:

1. Get the data via a direct [link](#) or through online platforms like [Kaggle](#).
2. Visualise data to gain insights:
 - Descriptive statistics of numerical features
 - Analysing the target variable.
 - Visualising and understanding correlated features.
 - Plotting Scatter Matrices to gain more insights about correlated variables.
 - Detect Outliers using Univariate and Bivariate analysis.
 - Visualising percentage of missing data, by feature.
3. Preprocessing:
 - Convert ordinal features to integer values.
 - Filling in missing data.
 - Removing outliers.
 - Creating new features.
 - Log transforming skewed features.
 - Making dummy variables for categorical columns.
4. Implementing:
 - Performing cross validation with many dirty models to see which ones perform best, in terms of train time and error.
 - Performing Feature Selection by using the top n features.
 - Fine tuning the top 3 models using the reduced set of features.
 - Stacking the tuned models with a Lasso meta-learner/blender.
5. Using the best model, predicting on the test set and reporting Kaggle Public Leader Score.

Metrics:

Logarithmic RMSE : The main metric used for this project (as given by [kaggle](#)) will be the logarithmic *RMSE* (Root Mean Squared Logarithmic Error) between the logarithm of the predicted value and logarithm of the observed sale price.

$$\sqrt{\frac{1}{m} \sum_{i=1}^m (\ln(y^i) - \ln(h_{\theta}(x^i)))^2}$$

Here:

- m is the number of instances in the dataset you are measuring the *RMSE* on.
- i refers to the index into the instances.
- y^i is the target label (the desired output value for that instance).
- h is known as the prediction function, also known as the *hypothesis*.
- θ refer to the parameters/weights of the prediction function, which decide how much weight to give each feature.
- x^i is the vector of all the feature values of the i^{th} instance in the dataset.
- $h_{\theta}(x^i) = \hat{y}$ is the predicted value.

Some points to note:

- *RMSE* is negatively oriented. This means that low RMSE is better.
- *RMSE* is indifferent to the direction of the error. Penalises both types of errors equally.
- Taking logs means that errors in predicting expensive houses and cheap houses will affect the result equally, i.e $\ln(5000) - \ln(10000) = \ln(50) - \ln(10)$.

Another metrics which is sometimes helpful is the **R2 Score**, also known as the *coefficient of determination*. It is measured as the proportion of the total variation in y (target label or price of the house, here) in the sample that can be attributed to the linear relationship with X (input variables or features). In other words, the proportion of variation in y , that can be explained by X is known as r^2 .

$$r^2 = \frac{\text{Variance of } \hat{y}}{\text{Variance of } y}$$

where y are the target labels, and \hat{y} are the predicted labels.

II. Analysis

Data Exploration:

Dataset:

The original dataset contained 112 explanatory variables describing 3970 property sales that had occurred in Ames, Iowa between 2006 and 2010 .

The final dataset has 79 explanatory variables describing 2930 residential property sales. Any explanatory variables requiring special knowledge were deleted along with records of property sales that were not residential sales, such as sale records of stand-alone garages, condos, and storage areas.

Inputs:

The 79 explanatory variables include:

- 20 continuous variables relating to various area dimensions for each observation. Such as the total dwelling square footage, lot size, etc. Area measurements of living area, basement, porches are broken down into individual categories based on quality and type.
- 14 discrete variables which quantify the number of items occurring within the house. Such as the number of bathrooms(full and half), bedrooms, kitchens, cars that can fit in the garage, etc. Date of construction and remodeling are also recorded.
- 23 ordinal variables which rate various items within the property such as the quality and condition of the heating/garage/exterior/fireplaces, etc.
- 22 nominal variables which describe the type of various physical aspects of the house along with the neighborhood, type of sale, and much more.
- There is a unique "Id" associated with each house.

The explanatory variables describe most, if not everything that a typical home buyer would want to know about a potential property. These 79 explanatory variables will be used to predict the price of a house.

Quick look at the data:

	1stFlrSF	2ndFlrSF	3SsnPorch	Alley	BedroomAbvGr	BldgType
0	856	854	0	NaN	3	1Fam
1	1262	0	0	NaN	3	1Fam
2	920	866	0	NaN	3	1Fam
3	961	756	0	NaN	3	1Fam
4	1145	1053	0	NaN	4	1Fam

Fig 2.1.1. Ames Housing Data

Descriptive Statistics about the features:

	1stFlrSF	2ndFlrSF	3SsnPorch	Alley	BedroomAbvGr
count	2919.000000	2919.000000	2919.000000	198.000000	2919.000000
mean	1159.581706	336.483727	2.602261	1.393939	2.860226
std	392.362079	428.701456	25.188169	0.489860	0.822693
min	334.000000	0.000000	0.000000	1.000000	0.000000
25%	876.000000	0.000000	0.000000	1.000000	2.000000
50%	1082.000000	0.000000	0.000000	1.000000	3.000000
75%	1387.500000	704.000000	0.000000	2.000000	3.000000
max	5095.000000	2065.000000	508.000000	2.000000	8.000000

Fig 2.1.2. Describing Numerical Features

Exploratory Visualizations:

Analysing the "SalePrice":

SalePrice	
count	1460.000000
mean	180921.195890
std	79442.502883
min	34900.000000
25%	129975.000000
50%	163000.000000
75%	214000.000000
max	755000.000000

Fig 2.2.1. Descriptive Statistics of target label

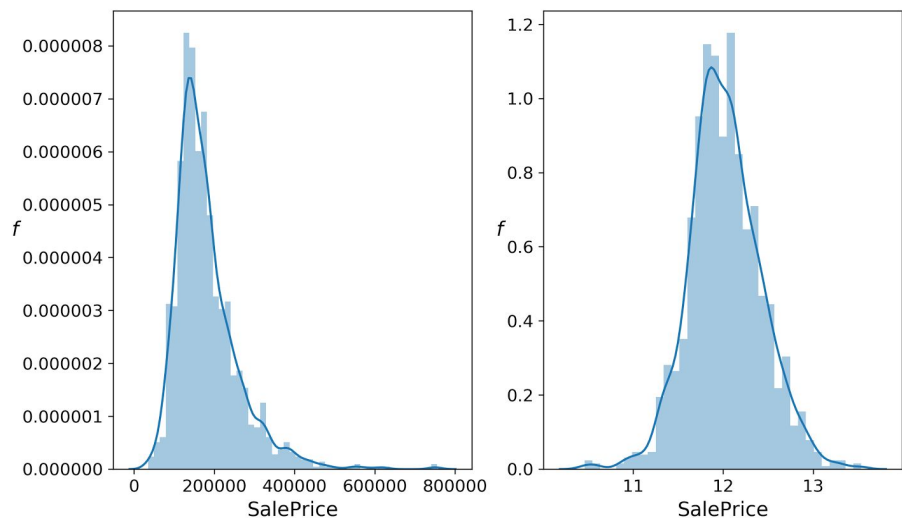


Fig 2.2.2. Log transforming the target label

As we can see, the target label is positively skewed. Most houses are priced around \$200,000 but there are some houses priced more than \$400,000, pulling the distribution to the right. Using log transformation on the 'SalePrice' will help in equally penalising errors in predicting expensive houses and cheap houses.

Correlation Matrix:

While exploring the dataset, I made a `high_corr` dictionary. Each key was a feature name and each value was a list of tuples with (correlated feature name, correlation coefficient), arranged in descending order.

An example would be:

'GrLivArea': [('TotRmsAbvGrd', 0.82), ('SalePrice', 0.70), ('2ndFlrSF', 0.68), ('FullBath', 0.63), ('OverallQual', 0.58), ('1stFlrSF', 0.56)]

This gave me good insights as to how one feature relates to others.

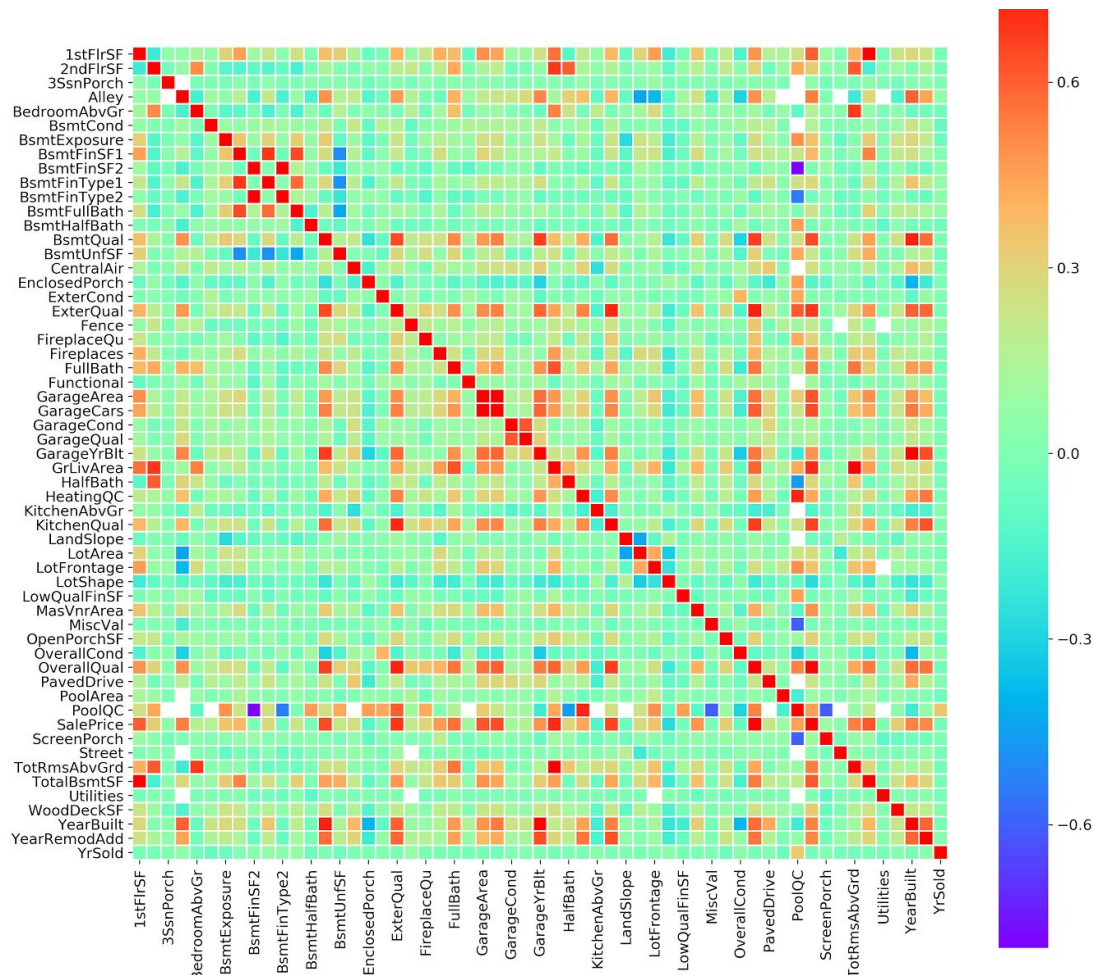


Fig 2.2.3. Correlation Matrix

Some interesting observations:

1. All features relating to the quality and condition of certain element in the house are mostly positively or weakly correlated with each other.
2. Similarly, space-related variables like 'GrLivArea', 'TotRmsAbvGr', 'TotalBsmSF' are all positively correlated with each other.
3. There are some *twins* (having a correlation coefficient of more than 0.68) features like: '1stFlrSF' and 'TotalBSmtSF', '2ndFlrSF' and 'GrLivArea', 'BedroomAbvGrd' and 'TotRmsAbvGrd', 'OverallQual' and 'ExterQual', 'GarageArea' and 'GarageCars', 'GarageYrBuilt' and 'YearBuilt' and many more.

4. The most correlated features with the target label, 'SalePrice' were:

['OverallQual', 'GrLivArea', 'ExterQual', 'KitchenQual', 'BsmtQual', 'GarageCars', 'GarageArea', 'TotalBsmtSF', '1stFlrSF', 'FullBath', 'Alley', 'TotRmsAbvGrd', 'YearBuilt', 'YearRemodAdd']

Another correlation matrix was made to analyse how the top features are correlated with each other. This step was essential to understand which features provide the most information and which features are repetitive.

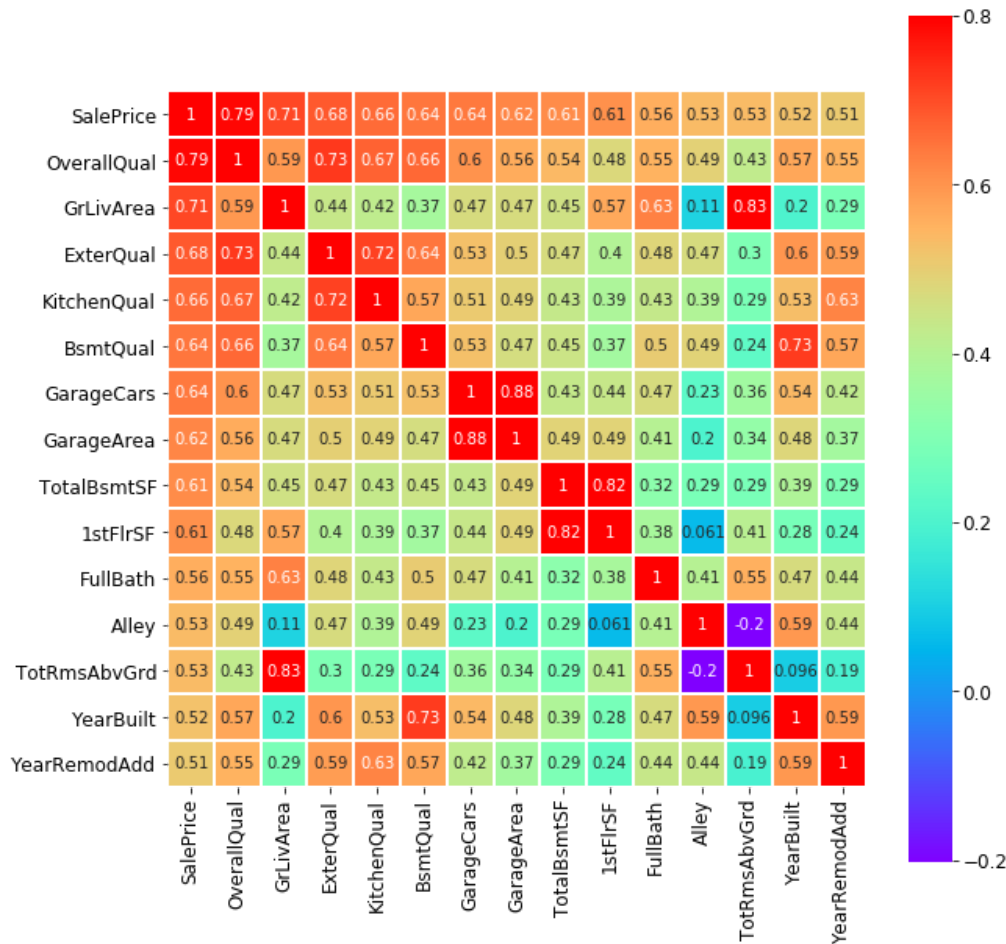


Fig 2.2.4. Top Features Correlation Matrix

Observations:

1. 'SalePrice' is the most correlated with 'OverallQual', 'ExterQual' and 'GrLivArea'.
2. 'OverallQual' is very strongly correlated with 'ExterQual'.
3. 'GrLivArea' is strongly correlated with 'TotRmsAbvGrd'.
4. 'ExterQual', 'OverallQual' and 'KitchenQual' are strongly correlated with each other.
5. 'BsmtQual' and 'YearBuilt' are strongly correlated with each other.
6. 'GarageCars' is strongly correlated with 'GarageArea', vice versa.
7. 'TotalBsmtSF' and '1stFlrSF' are strongly correlated with each other.

- 'FullBath', 'Alley' and 'YearRemodAdd' are not strongly correlated with any other top feature.

Therefore, the most informative features are:

```
["SalePrice", "OverallQual", "GrLivArea", "BsmtQual", "GarageCars", "TotalBsmtSF",  
"FullBath", "Alley", "YearRemodAdd"]
```

Scatter Matrix:

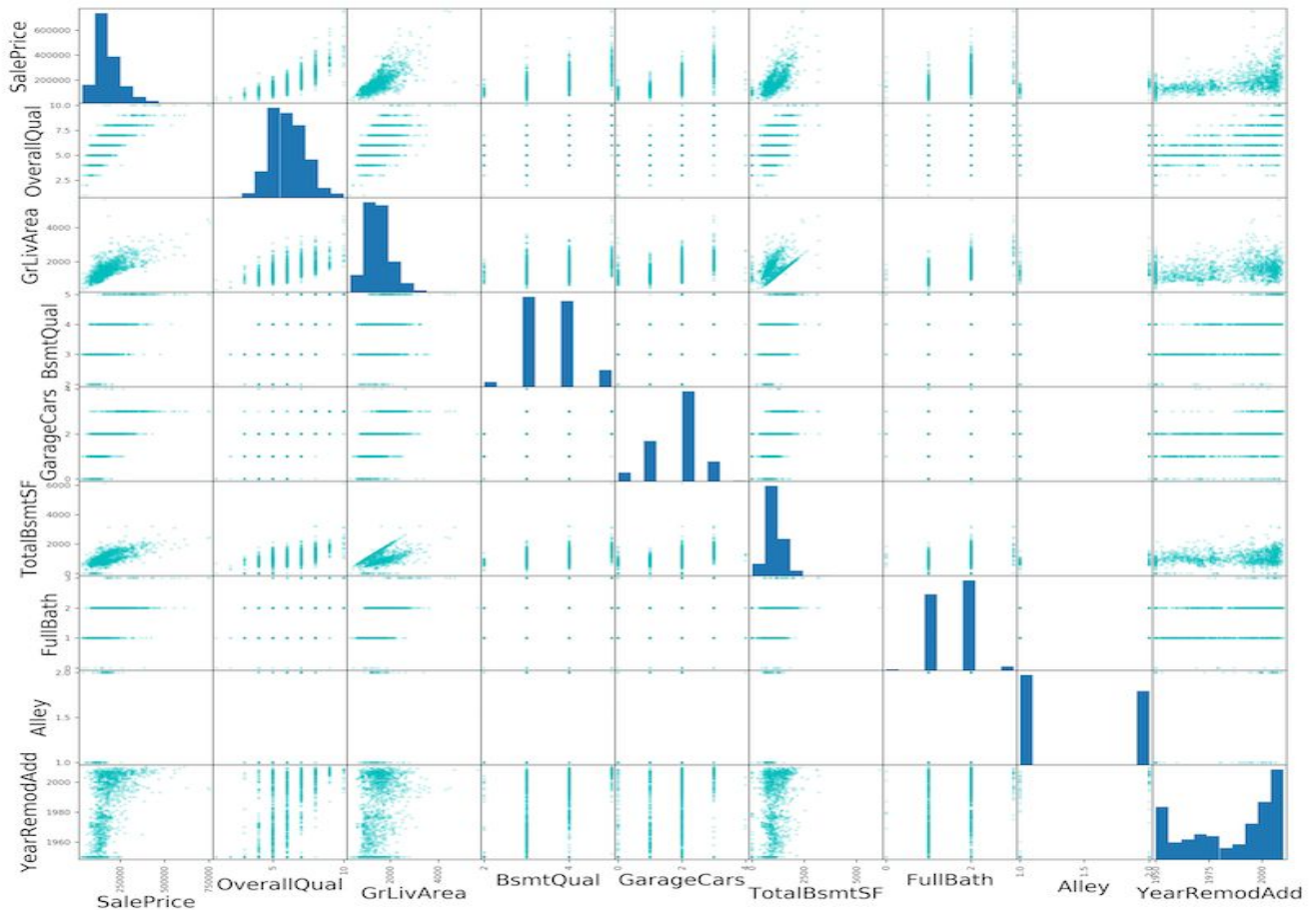


Fig 2.2.5. Scatter matrix of most informative features (sample)

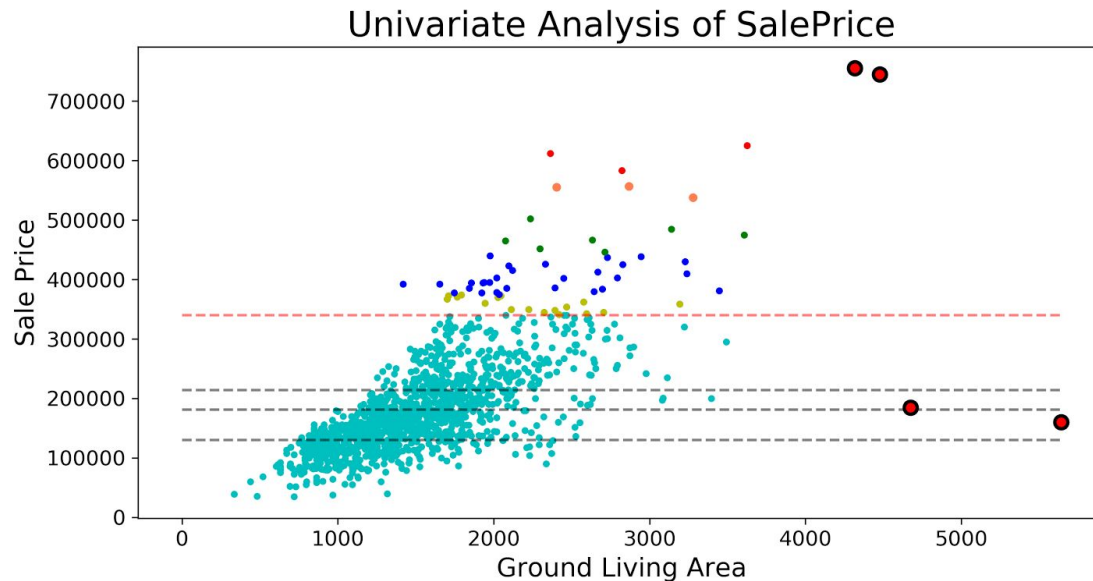
Observations:

- 'SalePrice' vs 'YearRemodAdd' shows a positive trend, but not exactly a linear positive trend. Prices seem to suddenly spike for houses that are built after 1990.
- As the 'GrLivArea' increases, the 'OverallQual' seems to increase, which indicates that the bigger the house, the better 'OverallQual' rating it has gotten. This is a bit weird. 'OverallQual' should not depend so much on how big the house is.
- There is some correlation between 'OverallQual' and 'YearRemodAdd' indicating the newer/newly remodeled the house, the better it's 'OverallQual'.
- In the 'GrLivArea' vs 'TotalBsmtSF', there is a line being formed showing that a lot of times 'GrLivArea' = 'TotalBsmtSF'. You can also see the houses where there is no

basement. Also, there are no houses which have 'GrLivArea' is smaller than 'TotalBsmtSF', which makes sense.

5. The 'GarageCars' is correlated with 'OverallQual' indicating the more the number of cars you can park in your house, the better its 'OverallQual'.
6. There is no strong evidence to support that the newer/newly remodeled the house ('YearRemodAdd'), the bigger it is ('GrLivArea'/'TotalBsmtSF'), or vice versa.
7. There is some correlation between the 'BsmtQual' and 'YearRemodAdd' indicating, the newer the house, the better quality basement it has.

Outlier Detection:



Observations:

- The 1399 points in cyan are not outliers.
- The black-dashed lines represent the Q1, Mean and Q3 of 'SalePrice', in order.
- The red-dashed line represents the upper_limit ($Q3 + 1.5 \times IQR$) for 'SalePrice'.
- The 17 points in yellow violate the upper_limit by less than 110%.
- The 29 points in blue violate the upper_limit by less than 130% but greater than 110%.
- The 7 points in green violate the upper_limit by less than 150% but greater than 130%.
- The 3 points in orange violate the upper_limit by less than 170% but greater than 150%.
- The 5 points in red violate the upper_limit greater than 170%.
- There are points which are highlighted with a black marker. These points are suggested to be removed from the training set [by the author](#). Two of these points are unusual sales (very large houses priced appropriately) and the other two are *partial sales*, which do not represent actual market values.

"I would recommend removing any houses with more than 4000 square feet from the data set (which eliminates these five unusual observations) before assigning it to students." - Dean DeCock

A **Bivariate Analysis** was also conducted using the set of features used for plotting the `scatter_matrix`. Two of the most interesting observations made were:

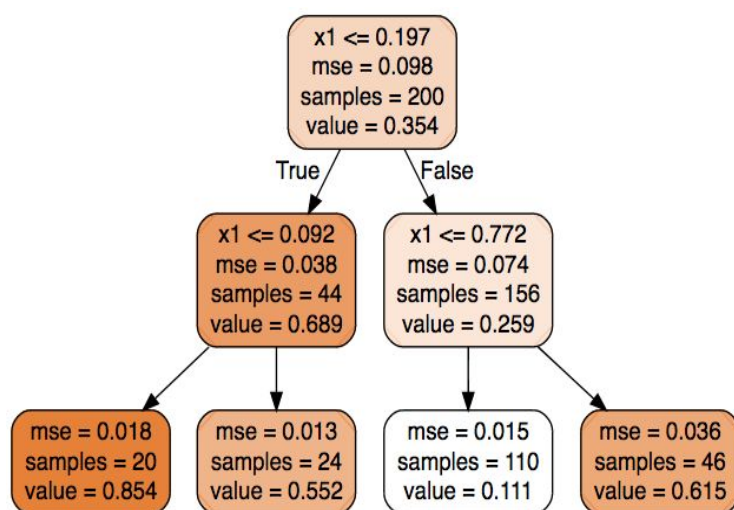
- A plot of '**TotalBsmtSF**' vs '**SalePrice**' graph clearly showed houses that do not have a basement. These houses were not priced above \$20,000. In addition, houses that have '**TotalBsmtSF**' more than 3000 square footage show that increasing the basement square footage beyond 2000 square feet does not contribute much to increase in the price.
- '**GarageCars**' vs '**SalePrice**' showed that as the number of cars you can park in your garage increases, the price of the house increases. The intriguing thing was that when the '**GarageCars**' = 4, the price actually falls when '**GarageCars**' = 3. This must have other factors affecting it like the total square footage, age, etc. of the house.

Algorithms and Techniques:

A *regression* algorithm helps us predict a target numeric value. A bunch of regressors is known as an *ensemble*. A *weak learner* is a regressor which does slightly better than random guessing. *Ensemble Learning* aggregates the predictions of a bunch of weak learners to arrive at a final answer. I have used many models in my implementation. The ones that have been used for *Stacking* are explained here.

DecisionTreeRegressor :

Decision Trees take the entire data as input. They first split the data into two subsets using a single feature k and a threshold t . This combination of k and t is such that it produces the purest subsets. The purest subsets are the ones that minimize the mean squared error. Once it splits the training set in two, it splits the subsets using the same method. Then, it splits the sub-subsets and so on, recursively. It stops splitting once it cannot find a split that will reduce the minimum impurity required or if it has reached its set maximum depth.



To make predictions for an instance, it traverses through the tree to find the node where there is a match.

Decision trees make very few assumptions about the training data. In other words, it is a *nonparametric* model. If we let the decision tree unconstrained, the tree structure adapts itself to the training data. This results in it memorizing or overfitting the training data. The nature of the *CART* algorithm makes Decision trees very sensitive to small or even no changes in the training data.

Fig 2.3.1. Sample Decision Tree Regressor

RandomForestRegressor:

In order to reduce the chances of overfitting, we can create multiple trees trained on different random sub-samples of the training set. This is what *Random Forests* do. Random Forests refer to an ensemble of Decision Trees.

How does a Random Forest Regression train and predict?

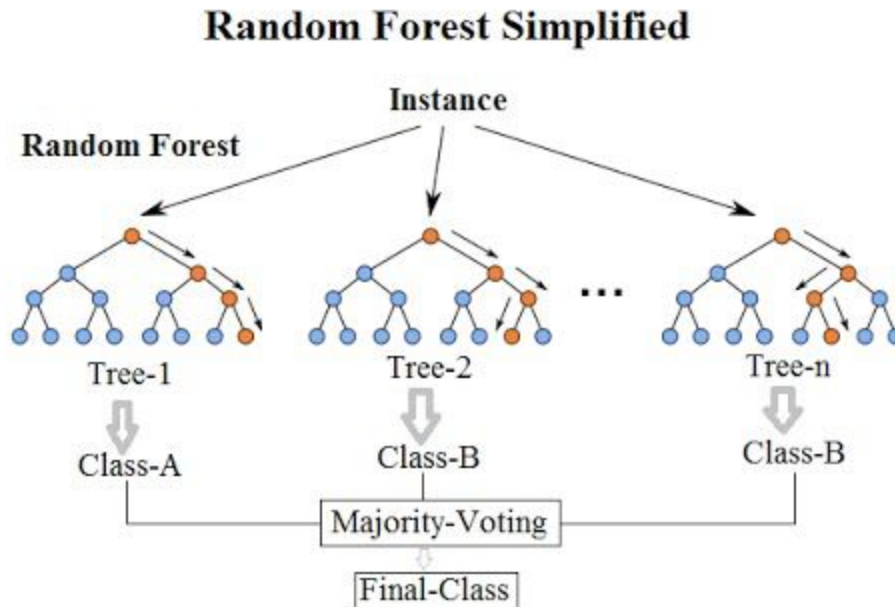


Fig 2.3.2. Sample Random Forest Regressor

1. Given the number of instances in the training set is M , RandomForestRegressor samples m instances at random *with replacement*.
2. Given the number of features is N , RandomForestRegressor randomly selects n features. These n features are only used to best split the nodes.
3. Using m training examples with n features, a decision tree is trained. The decision tree is grown depending on the `max_depth` parameter.
4. Step 1-3 are repeated for the set number of weak learners.

Predictions are made by averaging prediction over all the trees.

ExtraTreesRegressor:

This training algorithm refers to *Extremely Randomized Trees* ensemble. It works just like Random Forests with one exception. Random forests involved using a subset of the random features for each tree. *Extremely Randomized Trees* use random thresholds for each feature rather than searching for the best possible thresholds.

This trades in more bias for a lower variance. Random Forest or Decision Trees, in general, spend most of their time finding the best (feature, threshold) combination to best split each node. This makes Extra-Trees faster than regular Random Forests.

XGBRegressor:

Boosting refers to an ensemble learning technique wherein many weak learners are trained sequentially, each of which tries to correct its predecessor. Similarly, for each estimator, *Gradient Boosting* method tries to fit the new weak learner to the residual errors made by the previous weak learner.

A very simple example:

Let's say the target output, $y = 10$ and we have 3 weak learners.

If the output of the first weak learner is $\text{pred_y1} = 2$.

At this point, the output of the ensemble is $y_{\text{hat}} = \text{pred_y1} = 2$.

The second weak learner's target output will be, $y1_{\text{residual}} = y - \text{pred_y1} = 10 - 2 = 8$.

If the second weak learner's output is, $\text{pred_y2} = 6$, the output of the ensemble will become $y_{\text{hat}} = \text{pred_y1} + \text{pred_y2} = 2 + 6 = 8$.

The third weak learner's target output will be:

$$y2_{\text{residual}} = y1_{\text{residual}} - \text{pred_y2} = 8 - 6 = 2$$

If the output of the third weak learner is $\text{pred_y3} = 1$, the output of the ensemble will become $y_{\text{hat}} = \text{pred_y1} + \text{pred_y2} + \text{pred_y3} = 9$.

The algorithm will stop here as the set number of weak learners have been trained. 9 will be our final output, which is very close to the target output, 10.

This way, the ensemble's predictions gradually get better as more and more weak learners are added to it, fixing the residuals of the previous weak learner.

Traditionally, gradient-based implementations are slow because of the sequential nature in which each tree must be constructed and added to the model. XGBoost is an implementation of gradient boosted decision trees developed by Tianqi Chen and focusses on computational speed and model performance.

XGBoost has many advantages over Gradient Boosting:

- XGBoost helps in reducing overfitting as *regularization* is built in.
- XGBoost implements *parallel processing* by training each tree using as many cores as you want.
- XGBoost allows to run a *cross-validation* at each iteration of the boosting process. This makes it easy to get the exact *optimum number of boosting iterations* in a single run.
- Gradient Boosting is more of a *greedy algorithm* as it only stops splitting a node when it encounters a negative loss. XGBoost makes the splits up till the maximum depth specified and then starts *pruning* the tree backwards to remove splits beyond which there is no positive gain.

XGBoost has been very popular among machine learning competitions.

- "When in doubt, use xgboost". – Avito Winner's

Stacking:

Stacking is based on a very simple idea. Why do we need to aggregate the predictions of all predictors in an ensemble when we can train a model to perform the aggregation for us?

How does the training work?

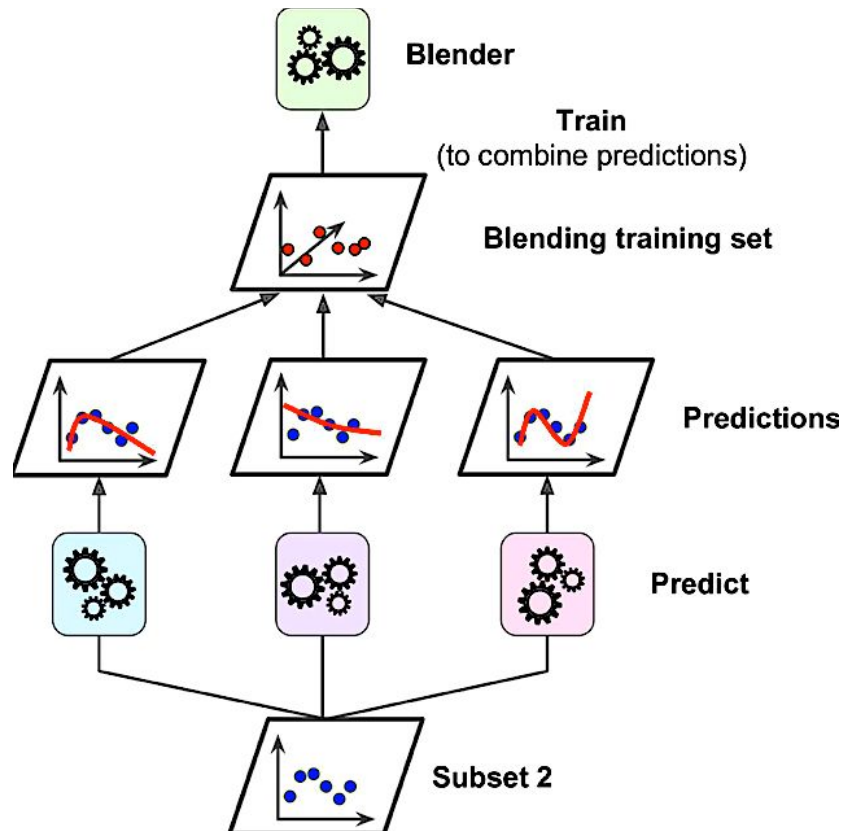


Fig 2.3.3. Training the Blender using Holdout Sets

1. Split the total training set into n_folds .
2. Let's say we have n_m base models. For each base-model, conduct n_fold iterations. In each iteration, train a clone of the base-model with $n - 1$ folds and predict on the remaining fold.
3. This gives you clean predictions for each training example, by each base-model. "Clean", meaning the predictors (clone-base-models) never saw these training examples during training.
4. Every time, you predict on a certain fold (certain indices), using a clone-base-model, record them. These are known as out-of-fold predictions.
5. These out-of fold predictions will include n_m predictions for each training example, where n_m is the number of base-models used.
6. We can then create a new training set using these predicted values as input features. This will make our training set n_m dimensional.
7. Finally, a *blender* or *meta-learner* is trained on this new training set. It will learn to predict the original target values, given the out-of-fold predictions.

How does it predict?

1. For each base model, we average the predictions made by all clones of the base model on the test data.
2. This gives us a set of n_m predictions for each test example, where n_m is the number of base-models used, similar to the out-of-fold predictions.
3. Finally, the *blender* or *meta-learner* predicts using the average of predictions of all base-models.












Benchmark Model:

Dean De Cock splits this project into 3 parts for his students. The first model requires students to submit a simplistic model (Model1) and should have a minimum R2 Score of 0.73. He further provides a benchmark model to have an R2 score of 0.80 as 80% of the variation in the residential sale prices can be explained by simply using the total square footage ('TotalBsmSF' + 'GrLivArea') and the 'Neighbourhood' feature.

“To give readers a benchmark, I found about 80% of the variation in residential sales price can be explained by simply taking into consideration the neighborhood and total square footage (TOTAL BSMT SF + GR LIV AREA) of the dwelling.” - Dean DeCock

[Kaggle](#) provides a benchmark model too at 0.408090, which is evaluated as the Root-Mean-Squared-Error between the logarithm of the predicted value and the logarithm of the observed sale price.

I will use the benchmark model provided by Kaggle.

Overview	Data	Kernels	Discussion	Leaderboard	Rules	Team	My Submissions	Submit Predictions		
2226	▼ 260	rohitbharti						0.38326	6	1mo
2227	▼ 260	ManuelProy						0.38814	2	16d
2228	▼ 260	TZech						0.39174	2	17d
2229	new	algar11						0.39323	2	1h
2230	▼ 259	Kartikay Gupta						0.40010	1	1mo
2231	▼ 259	Yongli Peng						0.40023	1	1mo
2232	▼ 259	AngelMo						0.40525	4	2mo
2233	▼ 259	chanming						0.40861	5	9d
📍		Sample Submission Benchma...						0.40890		
2234	▼ 259	hriday						0.40890	1	2mo
2235	▼ 259	Ethan 3						0.40890	2	2mo

III. Methodology

Data Preprocessing:

- Download the data.
- Remove "Id" columns as it's not required for prediction.
- Combine train and test set as they will require same transformations.
- Convert 'MSSubClass' and 'MoSold' from numerical to categorical columns.
- Make a list of continuous, discrete, ordinal and nominal variable names, manually using [Dean DeCock's](#) explanation of the feature space.
- Create list of numerical columns by combining continuous, discrete and ordinal variables.
- Make a list of categorical (nominal) columns (those columns in the data which are not numerical).
- Convert ordinal features to integer values
 - Features relating to condition and quality of an element were transformed together.
 - Features relating to basement finish were transformed together.
 - Other ordinal features included: ['Alley', 'BsmtExposure', 'CentralAir', 'Fence', 'Functional', 'LandSlope', 'LotShape', 'OverallQual', 'PavedDrive', 'Street', 'Utilities'] were also converted.
- Missing Data:

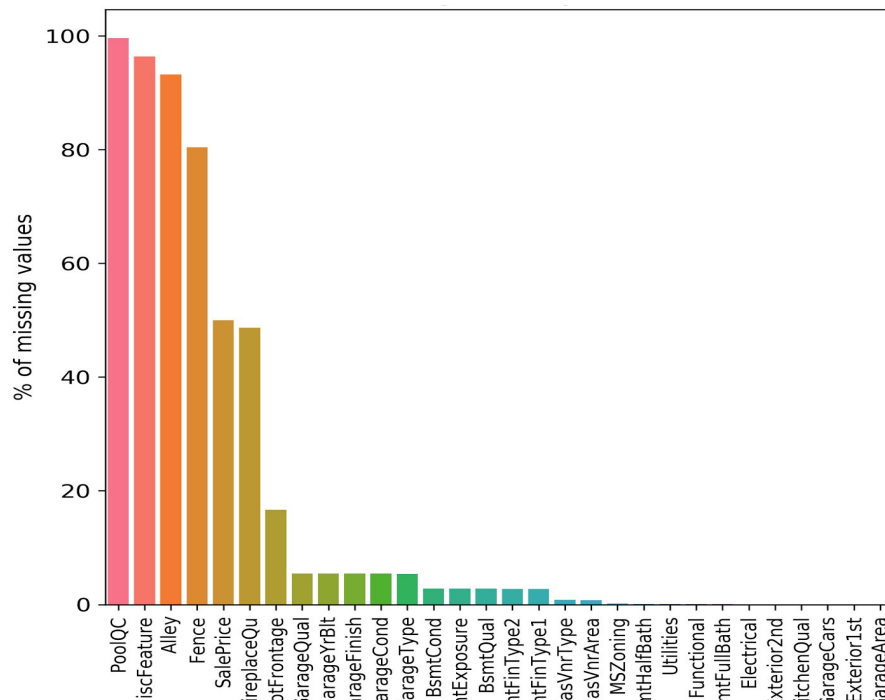


Fig 3.1.1. Percentage of Missing Data, by Feature

- Missing Numerical Data:
 - ['Alley', 'BsmtCond', 'BsmtExposure', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtFinType1', 'BsmtFinType2', 'BsmtFullBath', 'BsmtHalfBath', 'BsmtQual', 'BsmtUnfSF', 'Fence', 'FireplaceQu', 'Functional', 'GarageArea', 'GarageCars', 'GarageCond', 'GarageQual', 'GarageYrBlt', 'KitchenQual', 'LotFrontage', 'MasVnrArea', 'PoolQC', 'TotalBsmtSF', 'Utilities']
 - Except 'Functional', 'KitchenQual', 'LotFrontage' and 'Utilities', all other missing numerical values imputed with a 0 as they indicate the absence of an element in the house.
 - 'Functional' and 'KitchenQual' were imputed with their most frequent value.
 - 'LotFrontage' was imputed by the median of 'LotFrontage' of the neighborhood of the house.
 - All, except one value in 'Utilities' = 4. 'Utilities' was dropped as it did not provide any information gain. In other words, it does not help us differentiate between different houses.
- Missing Categorical Data:
 - ['Electrical', 'Exterior1st', 'Exterior2nd', 'GarageFinish', 'GarageType', 'MSZoning', 'MasVnrType', 'MiscFeature',]
 - Except 'GarageFinish', 'GarageType' and 'MiscFeature', all categorical features filled in by their most occurring value.
 - 'GarageFinish', 'GarageType' and 'MiscFeature' were filled in by "None".
- Outlier Removal:
 - Univariate Analysis and Bivariate Analysis conducted.
 - Only the instances with 'GrLivArea' >= 4000 were removed, as recommended by the author of the data.
- Feature Engineering: Features were created in two ways:
 - Simplification of existing features, especially the ordinal features.

"As a group, the 23 ordinal variables present special difficulties. Almost all of these variables are quality related, with the expectation that higher categories should yield a coefficient at or above the previous category. In some of my initial modeling, I found that the estimated coefficients for a number of these categories did not follow this rule, likely due to interrelations with other variables within the model. While not incorrect, this situation leads to confusing interpretations (lower quality is better?) for the students. I found that some of these anomalies could be remedied by collapsing some of the larger five and ten point quality scales into fewer categories." - Dean DeCock

The following features are simplified by collapsing their larger scales into three categories, each:

- ['SimpleBsmtQual', 'SimpleExterQual', 'SimpleFireplaceQu', 'SimpleKitchenQual', 'SimpleBsmtScore', 'SimpleOverallQual', 'SimpleGarageScore', 'SimpleFireplaceScore', 'SimplePoolScore', 'SimpleBsmtCond', 'SimplePoolQC', 'SimpleHeatingQC', 'SimpleFunctional', 'SimpleGarageCond', 'SimpleGarageQual', 'SimpleKitchenScore', 'SimpleOverallCond', 'SimpleExterCond', 'SimpleBsmtFinType1', 'SimpleBsmtFinType2']
- Combination of existing features:
 - These are made by combining the condition and quality attributes of one element of the house: ['ExterState', 'BsmtState', 'OverallState', 'GarageState'].
 - All 'State' features are combined to form 'Agg_State'.
 - These are made by combining the area/number of a certain element with their quality: ["KitchenScore", "FireplaceScore", "GarageScore", "PoolScore", "BsmtScore"]
 - Similarly, area/number of a certain elements are also combined with their simple quality features.
 - All 'Score' features are combined to form 'Agg_Score'.
 - All 'Bath' space variables are combined to create 'TotalBath' representing the total bath square footage in the house.
 - Square footage of basement, ground, 1st and 2nd floor are combined to form 'TotalSF'.
 - All 'Porch' space-variables are combined to form 'TotalPorchSF'.
 - Another feature representing the bedrooms/room of a house was created.
- All engineered features which had an absolute correlation coefficient of above 0.5 are retained. The rest, dropped.
- New features include:
 - ['TotalSF', 'Agg_Score', 'BsmtScore', 'SimpleBsmtScore', 'Agg_State', 'SimpleOverallQual', 'GarageScore', 'SimpleGarageScore', 'TotalBath', 'SimpleExterQual', 'ExterState', 'BsmtState', 'SimpleKitchenQual', 'OverallState', 'SimpleFireplaceQu', 'SimpleBsmtQual', 'FireplaceScore', 'SimpleFireplaceScore']
- At this point, there are a total of 96 features.
- Analysing Skewness: 53 of the 72 numerical features have an absolute skewness coefficient of more than 0.5 and are log transformed using `numpy.log1p()` function.
- Lastly, dummy variables are created for 23 categorical columns having a total of 193 categories. This gave a total of 265 features [72 (numerical features) + 193 (one-hot encoded features)].
- Finally, the prepared dataset is split back into train and test sets. The train set has 1456 instances (4 outliers removed) and the test set has 1459 instances.

Implementation:

1. **scorer**: A custom scorer was made to measure the root mean squared log error (RMSLE).
2. **scores()**: A `scores()` function is created to get a quick summary of any estimator's performance. It takes in 3 arguments i.e. (estimator, training set, targets) and returns a series with :
 - 3 fold cross validation RMSLE scores
 - Mean of cross-validation RMSLE score
 - Standard Deviation of cross-validation RMSLE scores
 - Training Time
3. The **dirty models** (set at default parameters) created were:
 - DecisionTreeRegressor
 - AdaBoostRegressor
 - RandomForestRegressor
 - SVR (Support Vector Machine Regressor)
 - ExtraTreesRegressor
 - GradientBoostingRegressor
 - Ridge (with RobustScaler()) to prevent problems from outliers.
 - Lasso (with RobustScaler()) to prevent problems from outliers.
 - XGBRegressor
4. **Comparing Dirty Models**: Using a `for` loop, for each model, 3-fold cross validation is conducted using the `scores()` function. The results of the `scores()` function are appended to a `results` DataFrame, presented once all the dirty models are trained.

Here is a diagram summarising the comparison between the dirty models in terms of training time and mean cross-validation score.

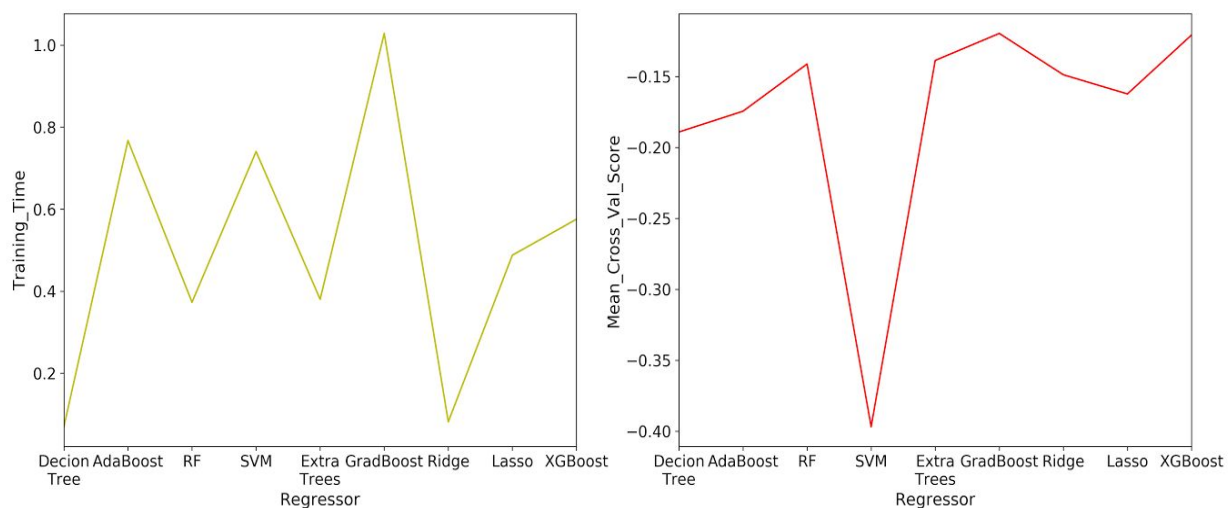


Fig 3.2.1. All estimators compared

Observations:

- The top 4 performing models are RandomForestRegressor, ExtraTreesRegressor, GradientBoostingRegressor, and XGBRegressor.
- GradientBoostingRegressor takes the most training time, followed by AdaBoostRegressor and then SVR.
- SVR is one of the most time consuming and the worst performing model.
- The 3 models, RandomForestRegressor, ExtraTreesRegressor and XGBRegressor provide a good balance of performance and training time.

5. Feature Selection:

Before fine-tuning the top 3 models, a round of feature selection is done. This helps in getting rid of unnecessary features which provide no information. Moreover, fine-tuning involves training your model multiple times. Reducing the feature space will reduce the model training time.

Process:

- Obtain feature importances via RandomForestRegressor's `.feature_importances_` method and arrange the feature names in descending order of importance.
- Calculate number of top n features required to satisfy each importance level in `[0.50, 0.60, 0.70, 0.80, 0.90, 0.93, 0.96, 0.99, 0.99999]`.
- 2 lists are made `train_time_list` and `rmsle_list`.
- For each importance level:
 - The training set is sliced using top n features representing the importance level.
 - Training time and Cross Validation Score is found out using the `scores()` function trained on the sliced training set. These are then appended to their specific lists.
- A line plot is created, plotting `train_time(Increasing Feature Importance)` vs `mean_cross_val_score` generated for each level of importance.

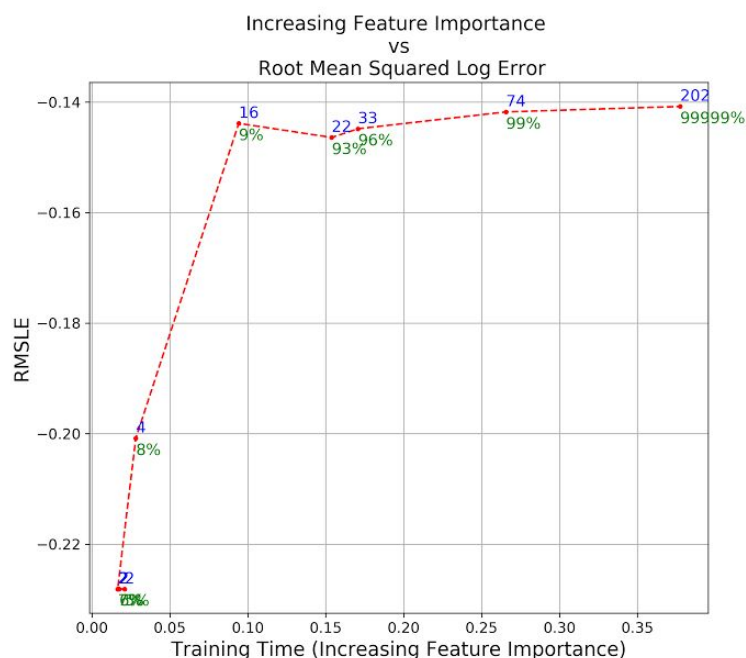


Fig 3.2.2. Increasing performance trained on top n features.

Note: There is a small error in the above picture. 80% , 90% and 99.999% have been incorrectly written as 8%, 9% and 99999%, respectively.

Observations:

1. The top 16 features represent 90% importance level.
2. The top 22 features represent 93% importance level and the top 33 features represent 96% importance level.
3. We can see the top 74 features accommodate approximately 99% of the feature importances. Our original feature space is 265 dimensional. That is more than 3 times 74!
4. Further, we see that the performance plateaus after using the top 74 features. Using the top 202 features gives an extremely small improvement while taking 0.10 seconds more to train.

We can use these top 74 features to train our 3 models:

- RandomForestRegressor,
- ExtraTreesRegressor and
- XGBRegressor

We can also compare each model's performance when trained on the reduced dataset with its' performance when trained on the full dataset. This will tell us if it's viable to use the top 74 features for the top 3 models.

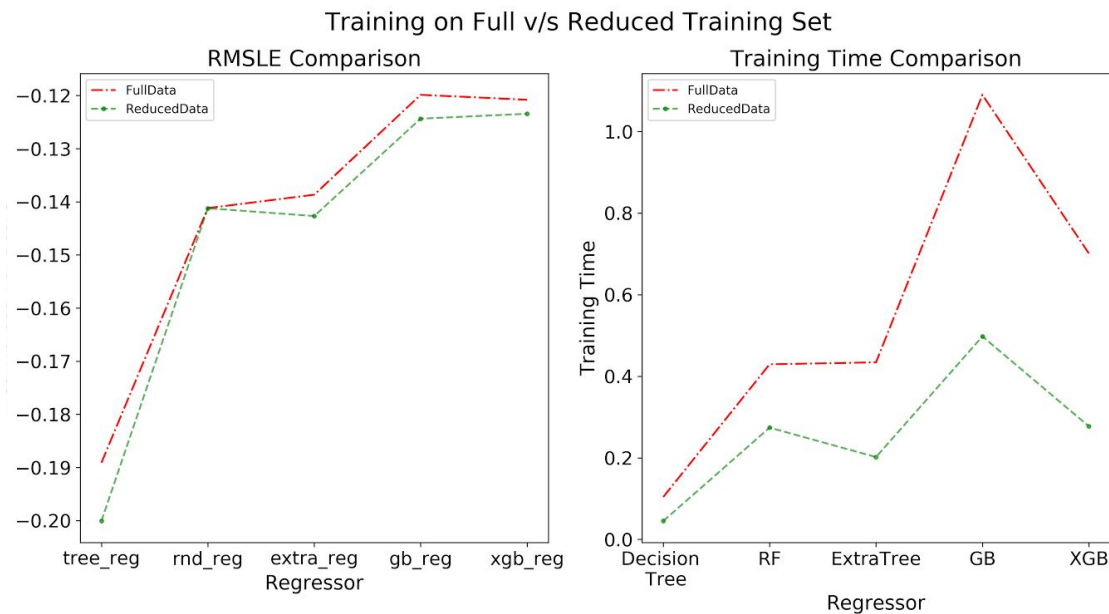


Fig 3.2.3. Comparing performance when trained on full v/s reduced dataset

Observations:

- i. There is a small drop in performance for each of the models except for RandomForestRegressor, which makes sense as the top 74 features are derived from this algorithm.
- ii. The training time for each algorithm is cut by 50% or more(aprx.).

- iii. The `DecisionTreeRegressor` shows the most drop in performance, followed by `ExtraTreeRegressor` and `GradientBoostingRegressor`.
- iv. Again, the top 3 models remain the same, providing a good balance of performance and training time.

Refinement:

For fine-tuning the model, I used scikit-learn's `GridSearchCV` class.

The performance on the optimised models was either not different or worse than the default models.

Here is a graph summarising the model performance comparison:

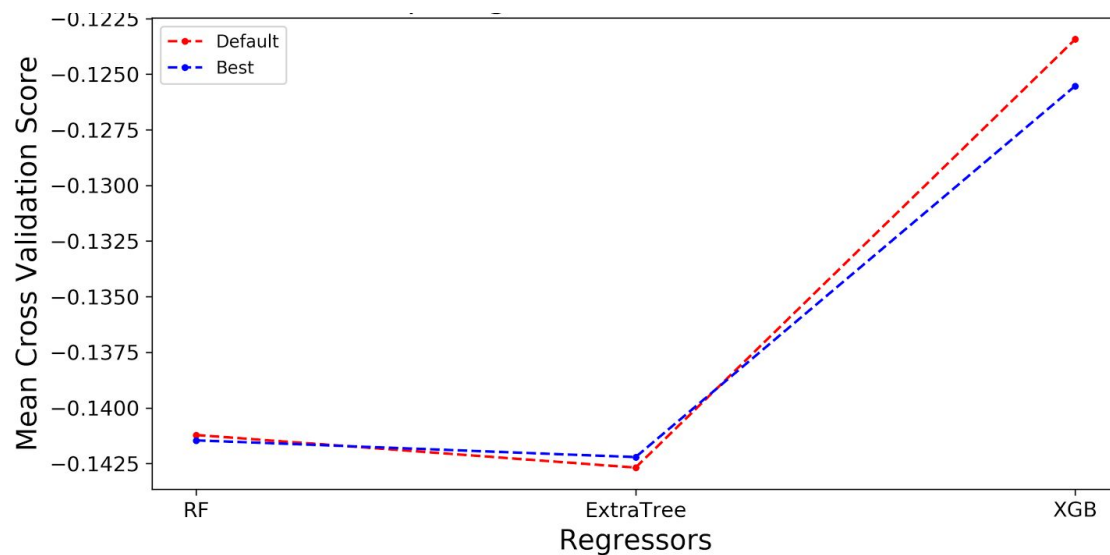


Fig 3.3.1: Comparing default v/s optimised model performance

Observations:

1. The default model of `RandomForestRegressor` performs slightly better than the optimised model.
2. The optimised model of `ExtraTreeRegressor` performs slightly better than the default model.
3. `XGBRegressor` takes a much bigger dip in performance, when optimised as compared to the other two models.
4. As there is not much difference between the mean cross validation scores of the optimised models vs the default models, I have used the optimised models for *stacking*.

Stacking Models:

I would like to thank [Serigne](#) for providing the code for stacking process. Brilliant! There are two types of stacking that have been performed. For each, a class has been built. Each class has these three base classes:

- **BaseEstimator**: Base class for all estimators in Scikit-Learn. Not adding `*args` or `**kwargs` in the `_init_` method gives access to two methods:
 - `get_params()` and
 - `set_params()`, useful for hyperparameter tuning.
- **RegressorMixin**: Mixin class for all regression estimators in Scikit-Learn. Provides a `score()` function which returns the coefficient of determination or R2 score.
- **TransformerMixin**: Mixin class for all transformers in Scikit-Learn. Provides a `fit_transform` function.

Simple Stacking Approach: This approach just involves predicting using a clone from each base model. And then averaging the predictions from each to arrive at a final answer. The class `AveragingModels` takes in a set of base-models.

- Its `fit()` function:
 - creates a list of clones of the base-models. Then, it fits the clones to training data.
- The `predict()` function:
 - Creates a stack of columns, shaped $m \times n_m$, where m is the number of predictions, and n_m is the number of clone-base models being used.
 - In other words, we get n_m predictions for each instance.
 - Returns the average of predictions made by each clone-base models, for each instance.

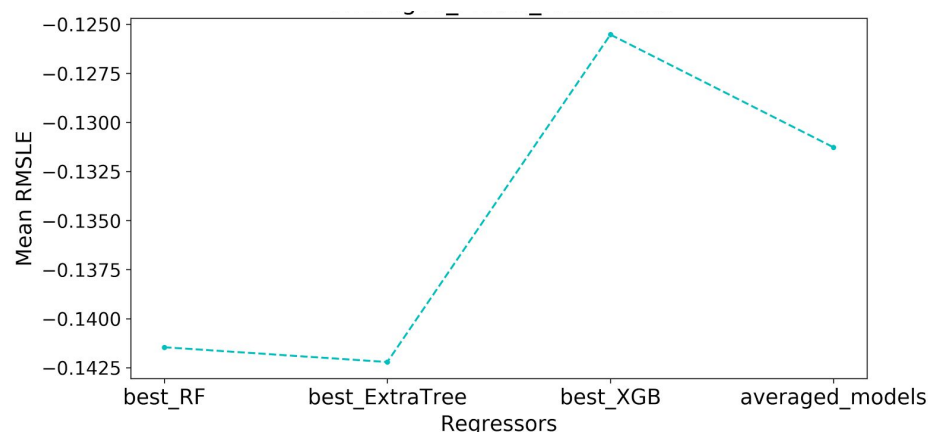


Fig 3.2.4. Comparing base model performance with averaged model performance

The average model does not perform as well as the `XGBRegressor`. We can look into more advanced Stacking techniques.

Stacking : Training a Meta-Learner: This is the approach explained in [Algorithms and Techniques](#) section.

I have used RandomForestRegressor, ExtraTreesRegressor and XGBRegressor as my base model. The meta_model here is Lasso, short for [Lasso Regression](#).

- The class StackedAveragedModels takes in 3 arguments.
 - A list of base-models to be used.
 - A meta_model or blender.
 - n-folds, the number of folds you want to divide the training set into.
- The fit() method:
 - Creates a list, self.base_models_ with n_m empty lists where n_m is the number of base-models being used.
 - Creates a clone of the meta learner.
 - Creates a K-fold object to split the training instance into n_folds.
 - Creates an empty array of out_of_fold_predictions. It has rows equal to the number of training examples. It has columns equal to the number of base-models being used (n_m).
 - For each base-model:
 - For each train-holdout split given:
 - Make a clone of the model, (called instance).
 - Append it to the its own (base-model) list in the self.base_models_ list.
 - Fit the instance to the current training examples, and then predict on the holdout examples.
 - Depending on which base-model you use, insert the predictions made on the holdout examples to the out_of_fold_predictions's respective (column, indices). This will give you clean predictions for each example, by each base-model.
 - Train the meta_model using the out_of_fold_predictions as input and the target labels as outputs.
- The predict() method:
 - Take the cloned-instances of one base-model.
 - Make predictions on the entire test set using all cloned-instances, one by one. Now, you will have n_cm predictions for each example in the test set, where n_cm is the number of cloned-instances per base-model.
 - Take the average over predictions made by cloned-instances, for each example in the test set. This will account for predictions made by one base-model.
 - Perform the above steps for each base model, returning the average prediction made by cloned-instance for each. This will give us n_m predictions for each example in the test set, where n_m is the number of base-models used. These are also known as meta_features.
 - Return a prediction made by the metamodel,, using the meta_features as input.

Let's see how we did:

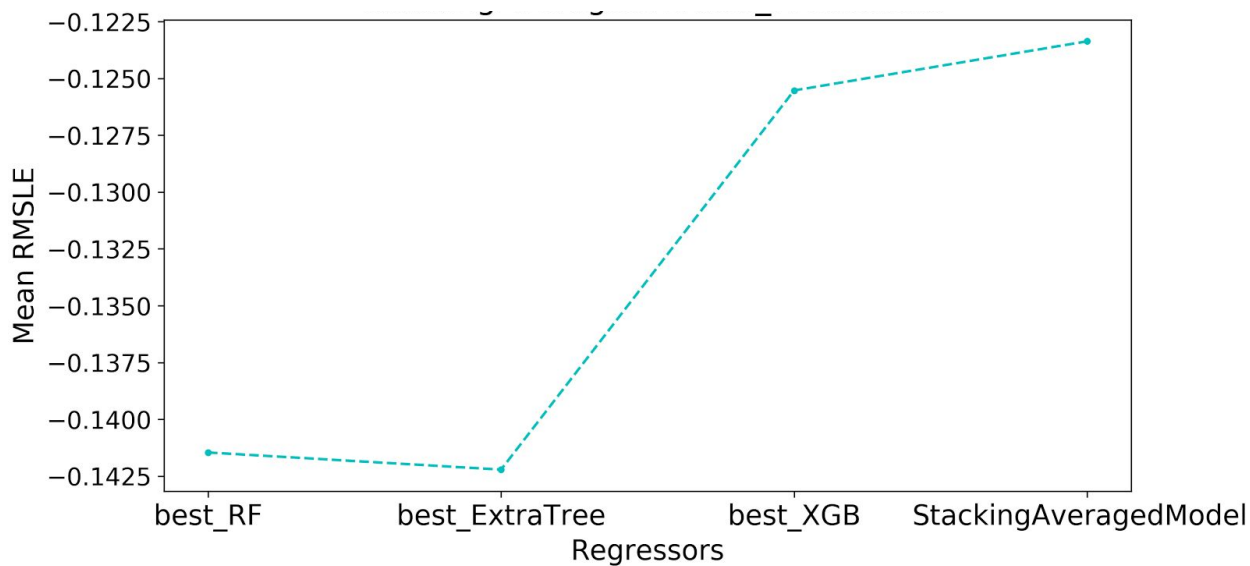


Fig 3.2.5 Comparing base model performance with Stacked averaged model performance

The Stacked Average Model outperforms every base algorithm on the local mean cross validation score! Great! We will now use this model to make prediction on the test dataset!

IV. Results

Model Evaluation and Validation

I first reduced the test set and then predicted using the Stacked Average Model. This submission lands me in the top 45 percentile of the Kaggle Leaderboard.

Overview	Data	Kernels	Discussion	Leaderboard	Rules	Team	My Submissions	Submit Predictions
1062	▼ 115	Thanawin						0.13086 27 17d
1063	new	beaybenja						0.13086 1 3d
1064	▼ 116	Aditya Bahl						0.13089 6 21d
1065	▼ 115	NguyenNgoc						0.13102 10 2mo
1066	▼ 115	SteveZhang						0.13102 10 2mo
1067	▼ 115	Doha						0.13105 15 19d
1068	new	Rishabh Chopra						0.13106 2 now
Your Best Entry ↑								

Fig 4.1.1. Kaggle Leaderboard at the time of submission

The performance on the test set and the mean cross validation score differed by 0.007693. This goes to show that the cross validation score was a good estimate of the real performance.

Justification

The ensembling methods have greatly improved the performance of the model.

The final model get's an RMSLE of 0.13106, which is more than 3 times better as compared to the benchmark model set by Kaggle.

In terms of R2 score, the model's prediction on the training set is able to explain 98.15% of the variability in house prices. This is again, much better than the benchmark R2 score of 0.80.

V. Conclusion

Free-Form Visualization

As I have used many visualizations until now. One would have understood I love making them. My final visualization for this project plots the predicted values made on the training set against the real values. Observe how it's making a straight line indicating that most of the variation in the target labels are caught by the model.

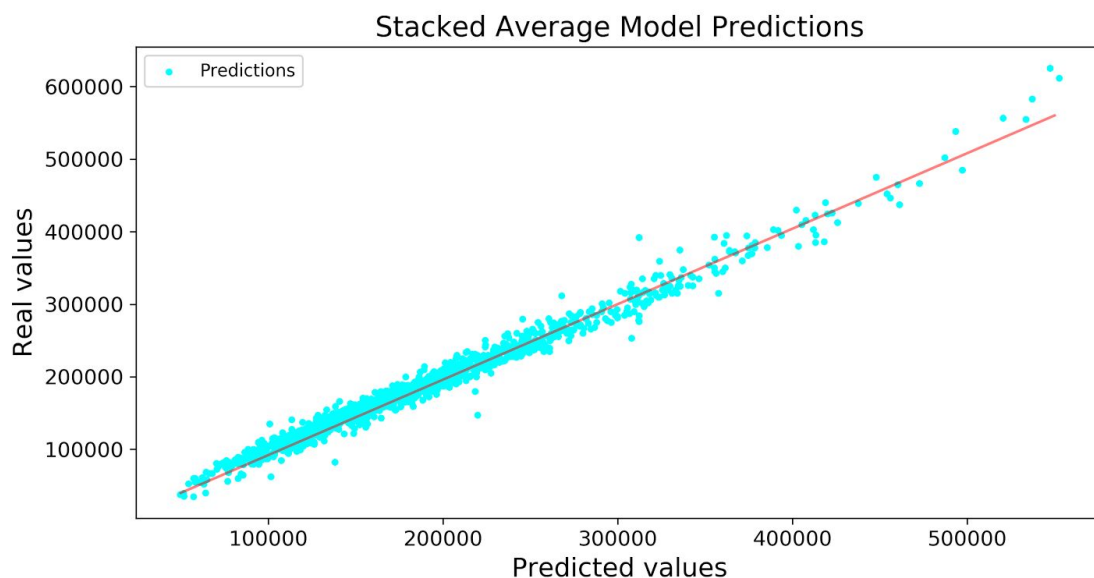


Fig 5.1.1. Training Set: Predicted values v/s Real values

Reflection

I started this project by reading the documentation of the data provided by Dean De Cock. Then, I visualised various aspects of the data. After gaining insights, I understood that it would be best to split the features into categories. This project required many preprocessing steps which had to be done in a proper sequence. I further learned about feature engineering, stacking and feature selection from books, kernels, and blogs. Implementation involved training many dirty models, conducting feature selection and choosing the best three models which perform best, given the reduced data. Then, I implemented stacking using the top 3 base models to get my final submission.

Improvement

This project has taught me a lot and will continue to do so. Here are some of the work I will pursue after MLND:

- Feature Engineering: Especially using the “Neighborhood” variable.
- Outlier detection: Seeing how deleting more than just 4 instances from the training set affect the model’s performance.
- Fine Tuning: How to better fine tune the base models in order to get better results from stacking.
- Trying our other base model combinations.
- Learn about better feature selection techniques.

References

1. [Ames, Iowa: Alternative to the Boston Housing Data as an End of Semester Regression Project](#)
2. [Data Documentation](#)
3. [Kaggle: House Prices: Advanced Regression Techniques](#)
4. [Complete Tutorial on Tree Based Modeling](#)
5. [Introduction to XGBoost](#)
6. [Hands-on Machine Learning with Scikit-Learn and TensorFlow](#)