Math 342W Final Project 2022

Peter Antonaros, Lamae Muharaj, Javendean Naipaul, Aracely Menjivar

May 26, 2022

1 Abstract

Predicting the sale prices of condos and co-ops can be a challenging task especially for a large city such as New York. Minute aspects of a neighborhood drastically alter the value of these homes and there is strong competition for estimating their costs. Companies such as Zillow have invested large sums of money into modeling this phenomena and more recently have begun to flip houses based on properties that their algorithms deem to be undervalued. In this paper, we outline the process of predicting home sale prices in Queens, New York as well as the final models themselves in comparison to the latest error metrics provided by Zillow.

2 Introduction

Prediction is a large business and with the surge of housing prices in the United States in the last few decades, many are interested in producing models to estimate the sale prices on homes. Since we do not have access to the same quantity or quality of data that a company such as Zillow has, we restrict ourselves to predicting sale prices of condos and co-ops within Queens, New York. Using a crowd sourced method of data collection through Amazon's Mechanical Turk, we are able to obtain over two thousand data points for approximately fifty zip codes in Queens. We began with an Ordinary Least Squares model to gauge how our features performed when predicting for sale price. Proceeding through a few different algorithms such as Lasso Regression and Regression Trees, we settled on a Bootstrap Aggregated Random Forest model for its ability to fit our complex hypothesis space variance minimization. Ultimately, our final model g predicts for $y = sale\_price$ with our observational unit being the property in question. In the following sections we outline our modeling process which includes how our data was obtained/manipulated, featurization/selection, and the model's respective metrics.

3 The Data

Our data first started as a link that directs the viewer to condos and co-ops within the borough of Queens posted to the OneKey MLS website in 2016 and 2017. This real

estate agency is one of the largest and most accurate multiple listing services in the state of New York. In order to obtain the information on the homes provided within the links, we commissioned 74 Amazon Mechanical Turk workers who were instructed to fill in our required columns to the best of their ability. Workers were paid for their work time in seconds, of which most averaged approximately 2.5 minutes and had high approval rates for their assignments completed.

Upon completion of our data collection we arrive at a data set consisting of $n = 2230$ samples from fifty five unique zip codes in Queens. With these figures, we can say that our coverage of the borough sits at approximately 80%, and represents the population of condos and co-ops sold in Queens relatively well. We refrain from stating that our data is a perfect sample of Queens since our coverage is not absolute and our samples are not in proportion with the number of condos and co-ops sold in each area of Queens during 2016-2017 period. Our constructed models must predict within the ranges of the present features and this is obvious when we entertain the idea of predicting for a condo or co-op in a zip code located in another state. The idea of interpolated prediction applies to our other features and must occur in order to achieve the error metrics we outline in this paper.

Ignoring data points related to MTurk in our set, we defined $p = 25$ features that we felt had influence over the sale price. These features serve as proxies to the causal drivers directly altering the sale price, and include number of bathrooms in the home and whether or not pets such as cats and dogs are allowed on the property. We would also like to note that we augmented our data with a census report from 2016 that provided us with the median income for nearly all zip codes in the United States. We will detail this further into the paper, but it is worth mentioning as it will increase our predictive accuracy.

Our response $y = saleprice$ is visualized in the following figure. One will notice a rightward skew with some of our y_i lying far along the tail. Although these may negatively impact some of our models, we retain them since they may capture valuable information about our phenomena through the use of other algorithms such as Random Forest.

3.1 Errors & Featurization

Prior to describing the featurization process we will include the following figure as it describes the raw columns imported from our CSV file. Notice that all columns from import are of type character and this was crucial to not change immediately.

Despite some of our columns being numeric measurements, they contained symbols such as \$ for dollar amounts along with commas to indicate places. Columns containing dollar amounts were pattern matched against known symbols and the symbols removed. Our regular expressions in conjunction with ignoring capitalization was rather valuable due to the presence of numerous misspellings and word abbreviation. An example of this that specifically focused on our worst offending feature *kitchen_type* is shown below. Clearly we see there are at least four unique possibilities when a value is present; *Eat In*, *Efficiency*, *Combo*, and *None*, yet **R** identifies 14 unique types of kitchen through no fault of its own. Once this process was completed for all columns we changed the types from strings to their proper data type.

Our first task was to take features that describe similar information and combine them into a single parent feature. An example of this occurs with features *common_charges*,

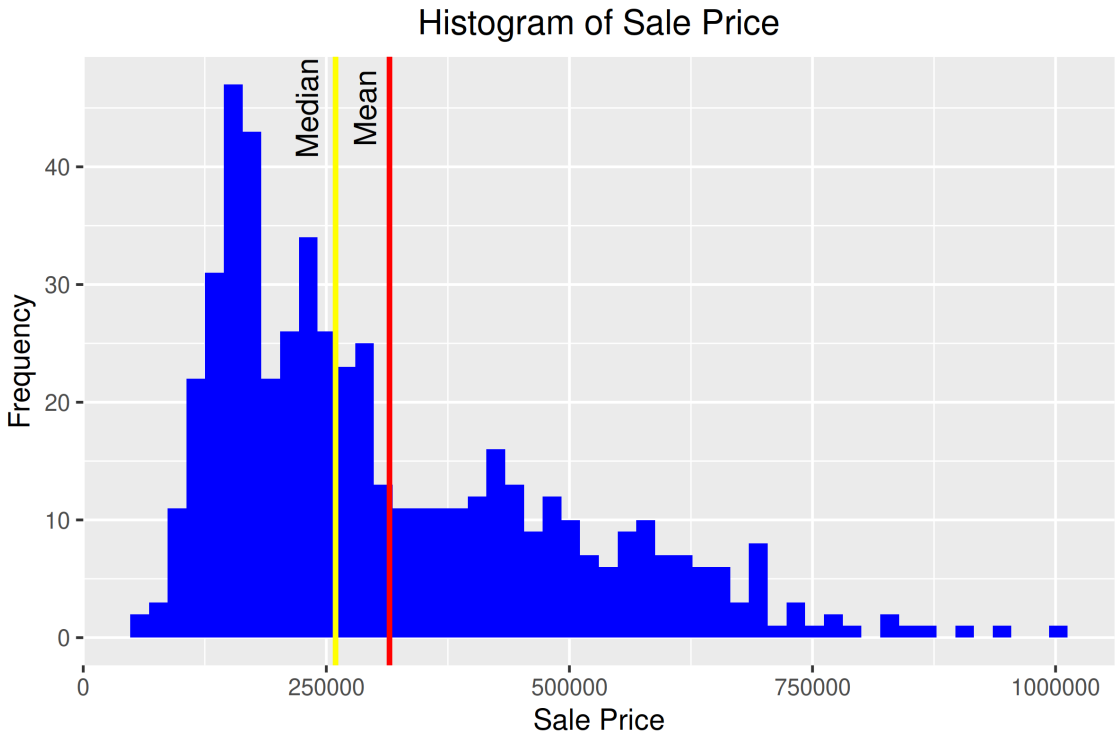

Figure 3.0.1: Visualizing the skew of sale price

maintenance_ cost, *parking_ charges*, and *total_ taxes*. We kept these as disjoint features for the sake of not losing granularity when collecting data, but now we can sum these values into a new feature which we named *total_ charges*. This action allowed us to capture this information as a single entity and simultaneously removed *NA* values since no row lacked all four child features. By similar process *num_ of_ bathrooms* was created as the summation of *num_ full_ bathrooms* and *num_ half_ bathrooms*. We did divide *num_ half_ bathrooms* by two so that our final feature would have a value such as 1.5, which represents 1 full and 1 half bathroom as you wuld see in a listing. Most of our other features remained as is, and no additional columns were created.

A last note we would like to make is our decision to treat *zip_ code* as numeric value rather than a categorical one. Treating zip codes as a numeric value allows for better generalizability through interpolation. An example from our data is the presence of the zip code 11385, but not 11386. Had we treated these as categorical, prediction in any Queens zip code outside of the ones present would lead to extrapolation. Since our data includes zip codes from 11004..11429, we could potentially predict on Queens zip codes not present in our data, but still within this range.

3.2 Missingness

Due to the nature of our data obtention it is almost certain that missing data will occur. Listings vary and despite having minimum informational standards, some of the features we sought to collect were either inapplicable for a particular home or simply not mentioned and

```
approx_year_built cats_allowed       common_charges     community_district_num  coop_condo
Min.   :1893      Length:2230        Length:2230        Min.   : 3.00          Length:2230
1st Qu.:1950      Class :character   Class :character   1st Qu.:25.00          Class :character
Median :1958      Mode  :character   Mode  :character   Median :26.00          Mode  :character
Mean   :1963                                            Mean   :26.33
3rd Qu.:1970                                            3rd Qu.:28.00
Max.   :2017                                            Max.   :32.00
NA's   :40                                              NA's   :19
date_of_sale       dining_room_type   dogs_allowed       fuel_type         full_address_or_zip_code
Length:2230        Length:2230        Length:2230        Length:2230        Length:2230
Class :character   Class :character   Class :character   Class :character   Class :character
Mode  :character   Mode  :character   Mode  :character   Mode  :character   Mode  :character



garage_exists      kitchen_type       maintenance_cost    num_bedrooms   num_floors_in_building
Length:2230        Length:2230        Length:2230        Min.   :0.000   Min.   : 1.000
Class :character   Class :character   Class :character   1st Qu.:1.000   1st Qu.: 3.000
Mode  :character   Mode  :character   Mode  :character   Median :2.000   Median : 6.000
                                                         Mean   :1.653   Mean   : 7.785
                                                         3rd Qu.:2.000   3rd Qu.: 7.000
                                                         Max.   :6.000   Max.   :34.000
                                                         NA's   :115     NA's   :650
num_full_bathrooms num_half_bathrooms num_total_rooms  parking_charges    pct_tax_deductibl
Min.   :1.000      Min.   :0.0000     Min.   : 0.000   Length:2230        Min.   :20.0
1st Qu.:1.000      1st Qu.:1.0000     1st Qu.: 3.000   Class :character   1st Qu.:40.0
Median :1.000      Median :1.0000     Median : 4.000   Mode  :character   Median :50.0
Mean   :1.231      Mean   :0.9535     Mean   : 4.139                      Mean   :45.4
3rd Qu.:1.000      3rd Qu.:1.0000     3rd Qu.: 5.000                      3rd Qu.:50.0
Max.   :3.000      Max.   :2.0000     Max.   :14.000                      Max.   :75.0
                   NA's   :2058       NA's   :2                           NA's   :1754
 sale_price          sq_footage     total_taxes          walk_score    listing_price_to_nearest_1000
Length:2230        Min.   : 100.0   Length:2230        Min.   : 7.00   Length:2230
Class :character   1st Qu.: 743.0   Class :character   1st Qu.:77.00   Class :character
Mode  :character   Median : 881.0   Mode  :character   Median :89.00   Mode  :character
                   Mean   : 955.4                      Mean   :83.92
                   3rd Qu.:1100.0                      3rd Qu.:95.00
                   Max.   :6215.0                      Max.   :99.00
                   NA's   :1210
```

Figure 3.1.2: Housing Data Summary (Initial Import)

```
 [1] "eat in"              "efficiency"          "Combo"                "combo"
 [5] "Eat In"              NA                    "Eat in"               "1955"
 [9] "eatin"               "efficiency kitchene" "efficiency kitchen"   "efficiemcy"
[13] "none"                "efficiency ktchen"
```

Figure 3.1.3: Kitchen Type Feature Unique Occurrences

therefore not recorded. These missing values increase error due to ignorance δ and without proper mitigation, can drastically limit predictive accuracy.

The naive method of contending with NA values is to use Listwise-Deletion. This process removes rows where any of the p columns are NA. Listwise Deletion has limited use cases, particularly when the total occurrences of NA values are insignificant compared to the n observations. Unfortunately in our data set, NA values occurred frequently and scattered throughout. A check of our summary implied that with the use of Listwise-Deletion, we would be left with a fragment of our original data set.

We decided on a two fold approach to solving this issue starting with the Miss Forest

algorithm to fill *NA* values. Formally, this algorithm is defined as a non-parametric approach to missing value imputation using Random Forests. Miss Forest begins by filling in *NA* values with either $\bar{x}_i$ or Mode$[x_i]$ dependent on whether the current column is numeric or categorical. Some implementations substitute the median of a column rather than the average yet they perform the same essential duty. The algorithm then iterates over each column treating the current as a response and using the others as features in a Random Forest model for prediction. The algorithm ends once a max iteration condition is satisfied, or when the difference in imputed values between iterations is less than a tolerance. We ran the algorithm with custom parameters which can be seen in our **R** notebook. The verbose log of our iterations are detailed in Figure 3.2.4 located below.

Our imputed data set now contained no *NA* values and the figure located below provides a summary of the iterations. Our next step was to create a missingness table which tracked *NA* values in each feature with binary indicators, 0's for non *NA* values and 1's for *NA* values. If we capture our response as a function of our original features $f(x_1, x_2, \ldots, x_p) + \delta_1$, then column binding our missingness table yields $f(x_1, x_2, \ldots, x_p, m_1, m_2, \ldots, m_p) + \delta_2$. The addition of $m_1, m_2, \ldots, m_p$ implies that $\delta_2 < \delta_1$, so we benefit from augmenting our data with this missingness table. These supplementary features may not be of any use after the feature selection process, but it is wise to consider them. They will either provide better predictive accuracy for the reasons above, or amplify noise in the data and will be filtered during selection.

In order for our results to be honest evaluations of our models, our train and holdout test splits were created such that imputed sale prices fell into the training split leaving real observed sale prices for our testing. Approximately 75% of our data was allocated to the training portion, with the remainder being reserved as the holdout. This decision will lead to error much higher than initially expected in an attempt to preserve out of sample honesty. We can also say that we expect our errors to be better representative of deploying these models in the real world because they serve as a pseudo upper bound. An additional benefit of choosing our splits in this way is it allows for interpretability of out of sample error. We will be able to state prediction metrics as they relate to observed sale prices, rather than prediction metrics on imputed sale prices.

3.3 Feature Selection

Recursive Feature Elimination, variable importance tests and correlation tests were used to select a subset of features from our data set. We decided on retaining our original data set and created a copy, in order to remove features which were deemed to be uninformative to our response. This allowed us to train models on both data sets in order to guage whether or not our feature selection was beneficial.

Briefly, Recursive Feature Elimination is a wrapper function for a machine learning algorithm (in our case Random Forests). It fits a model on the current subset of features, ranks them by importance and then continually fits a new model. Various stopping conditions exist, the most common being a target number of features to achieve whilst minimizing error. Since we ran Random Forests as the core model for this feature elimination wrapper, we used criterion specific to Random Forests in order to judge features against each other. This is a common approach and we will see later how a feature's level in a tree is a direct implication

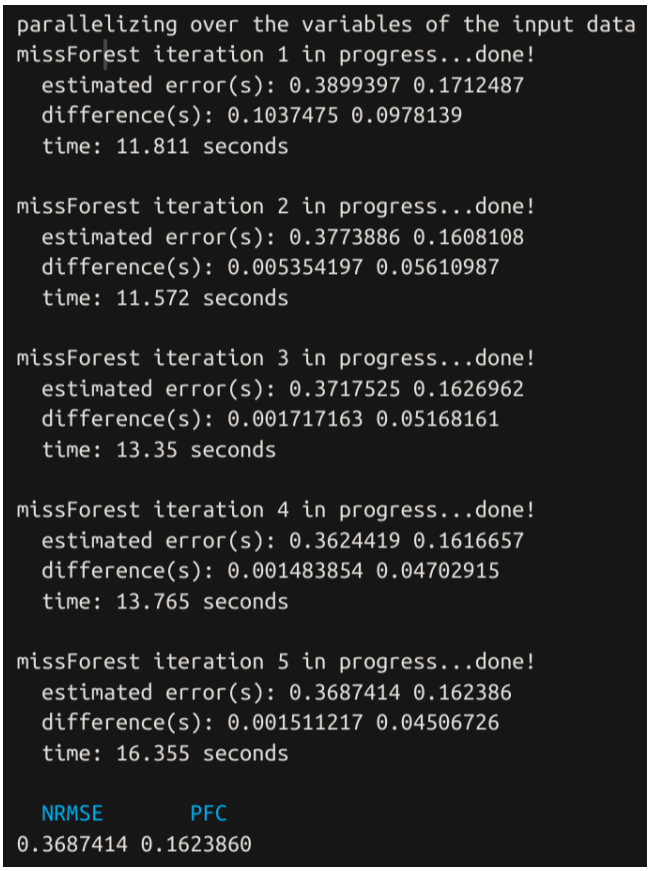

Figure 3.2.4: Miss Forest Algorithm with 500 Trees

of its importance to the response. Other methods of feature selection exist such as using Lasso Regression to bring a subset of feature coefficients to zero. We decided against this due to the instability of Lasso Regression and its seemingly arbitrary dropping of unimportant features that rank closely.

4 Modeling

We would like to mention that of our two data sets, the first containing all features and the second containing the selected ones, we provide results for the latter. The only model we provide error metrics for the former data set is Lasso Regression because we were interested in how well Lasso could perform feature selection. The reasoning for this stems from the collinearities in our full feature data set which caused metrics to be untrustworthy. For those who wish to see results from training with all of our features for all models, the **R** notebook includes this. Described below are models that have been trained on the selected feature data set and errors provided were computed on both a test set derived from the training data and a final fully independent holdout test set.

4.1 Linear Regression (OLS) Modeling

Our Ordinary Least Squares Model with no interaction between features performed poorly which was expected. This model fails to capture many of the non linearities present and indicates that we need to employ an algorithm that makes use of a richer $\mathcal{H}$ space. Figure 4.1.5 shows our beta coefficients as well as the RMSE error on the test/holdout set as well as the R^2 on the holdout set. Despite a poor model, we can see that certain features such as *walk_score* and our augmented *median_income* play a crucial role in determining y. For every additional dollar that *median_income* increases by, *sale_price* rises by approximately four dollars. Ordinary Least Squares greatest benefit is interpretability of coefficients which grants us the ability to make better informed decisions later into the modeling process.

```
          (Intercept)              zip_code            walk_score       coop_condocondo          totalCharges
        -6.077211e+05          1.432852e+00          9.249337e+02          1.478112e+05          1.483792e-01
num_floors_in_building            sq_footage         totalBathrooms      approx_year_built         median_income
         5.825599e+03          1.497718e+02          5.958341e+04          2.540676e+02          4.117248e-01
community_district_num        date_of_sale02        date_of_sale03        date_of_sale04        date_of_sale05
         5.560222e+02         -1.153223e+04         -2.612196e+04         -5.208010e+04         -2.537119e+04
        date_of_sale06        date_of_sale07        date_of_sale08        date_of_sale09        date_of_sale10
        -4.007075e+04         -2.798675e+04          7.131984e+03         -2.483617e+04         -4.370906e+03
        date_of_sale11        date_of_sale12      kitchen_typeeatin kitchen_typeefficiency      kitchen_typenone
        -8.872095e+03         -5.665668e+03          1.726842e+03         -1.819912e+04          3.609610e+03
         num_bedrooms       num_total_rooms        sq_footage_miss
         2.911017e+04         -1.997282e+02         -1.014848e+04
[1] 51925.01
[1] 95920.74
[1] 0.7139837
```

Figure 4.1.5: Coefficients/RMSE(Test)/RMSE(Holdout)/R-Squared(Holdout)

4.2 Linear Regression (Lasso) Modeling

We will not focus heavily on this model since it was merely for experimentation purposes. Since we employed Recursive Feature Elimination, this model was used to judge the ability of Lasso to disregard features in comparison to RFE. We should also note the results in Figure 4.2.6 are cross validated. Although at first glance it appears that Lasso performs worse than OLS, these are a better representation of true performance regardless of the data sample. Cross validation was used solely for the purpose of a fair comparison to RFE.

```
[1] 56528.43
[1] 103330.2
[1] 0.6680901
```

Figure 4.2.6: RMSE(Test)/RMSE(Holdout)/R-Squared(Holdout)

4.3 Regression Tree Modeling

As stated in our Ordinary Least Squares section, we were in need of an algorithm that could utilize a richer hypothesis space. Regression Trees can capture this as well as interactions between features by testing multiple features at each split. In Figure 4.3.7 we can see the top levels of our tree constructed on the training data. We notice that *total_bathrooms*, *totalCharges* are the best initial choices. This is subject to change as a single Regression Tree is highly variable, in its feature selections and error performance.

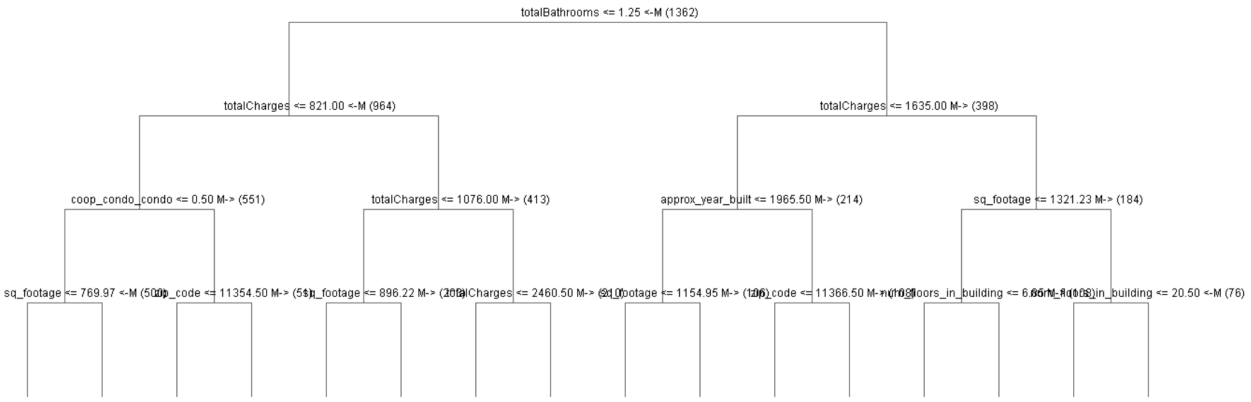

Figure 4.3.7: First few layers of our Regression Tree

Figure 4.3.8 indicates that our choice of Regression tree improved both RMSE and R^2 considerably. Although these results are variable, we would still expect Regression Trees to outperform Ordinary Least Squares. Our data is not linear and any linear model such as OLS and Lasso will not be able to capture this as well as Trees.

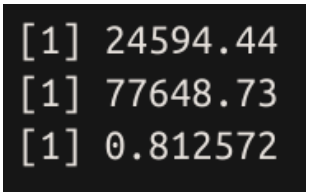

Figure 4.3.8: RMSE(Test)/RMSE(Holdout)/R-Squared(Holdout)

4.4 Random Forest Modeling

In order to solve the variability issue in Regression Trees, we use Random Forests. This algorithm employs an ensemble of trees built each built on a different subset of the total p features. By taking what is essentially an average over these trees, we reduce variance in our error metrics and produce a model that is consistent over different training splits. Although more computationally expensive, we can parallelize this process in our **R** function which reduces the training time considerably. Our decision was that we would finalize our model with the Random Forest algorithm, and in the next section we detail our results.

5 Performance Results for Random Forest Model

Figure 5.0.9 details the RMSE of our test and holdout set as well as the R^2 value for the holdout set. We see a distinct divide between the performance on the test and holdout set. This occurs because of our decision to train solely on the imputed $sale_price$ feature and reserve the true observed ones in our holdout set. The test set is a subset of our training set and thus we expect our model to perform well on this since all sale prices derive from our Miss Forest Imputation. The true measure of model performance would be more along the lines of our holdout metrics since we test against observed values. In our **R** notebook, we have also bootstrapped this model in order to reduce variance further. We exclude the results since this process only has a slight effect on our holdout set.

```
[1] 15325.94
[1] 73719.6
[1] 0.8310603
```

Figure 5.0.9: RMSE(Test)/RMSE(Holdout)/R-Squared(Holdout)

Lastly, we present our 'mtry' tuned tree results as well as g_{final} which has been trained on all data and whos out of bag metrics are provided.

```
Random Forest

1362 samples
  15 predictor

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 2 times)
Summary of sample sizes: 1089, 1090, 1091, 1089, 1089, 1091, ...
Resampling results across tuning parameters:

  mtry  RMSE      Rsquared   MAE
   1    37239.67  0.9657834  29555.70
   2    21543.57  0.9821357  15031.71
   3    19203.36  0.9846001  12838.51
   4    18566.30  0.9851069  12258.60
   5    18512.15  0.9848871  12161.34
   6    18387.76  0.9849128  12076.28
   7    18352.06  0.9848289  12017.47
   8    18560.96  0.9843686  12145.96
   9    18660.99  0.9840982  12200.19
  10    18890.44  0.9836488  12322.11
  11    19115.77  0.9831188  12431.45
  12    19402.62  0.9825566  12673.40
  13    19856.16  0.9816389  12950.45
  14    20182.60  0.9809521  13181.64
  15    20640.80  0.9800170  13484.44
  16    20672.23  0.9799412  13501.93

RMSE was used to select the optimal model using the smallest value.
The final value used for the model was mtry = 7.
[1] 70650.47
[1] 0.8448343
```

Figure 5.0.10: RMSE & R-Squared(Test per mtry)/RMSE(Holdout)/R-Squared(Holdout)

```
Random Forest

2230 samples
  15 predictor

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 2 times)
Summary of sample sizes: 1783, 1783, 1783, 1785, 1786, 1783, ...
Resampling results across tuning parameters:

  mtry  RMSE      Rsquared   MAE
   1    48494.28  0.9309002  32830.18
   2    35679.91  0.9494745  18290.94
   3    33992.52  0.9526466  16608.79
   4    33288.71  0.9541931  16155.45
   5    33104.24  0.9545238  16098.49
   6    33080.69  0.9544316  16191.54
   7    33005.10  0.9545790  16257.75
   8    33085.67  0.9542809  16430.65
   9    33075.51  0.9542466  16562.81
  10    33314.83  0.9535740  16797.35
  11    33384.33  0.9533226  16920.37
  12    33563.70  0.9527764  17133.47
  13    33752.38  0.9521827  17432.14
  14    34164.72  0.9509238  17788.51
  15    34583.33  0.9496928  18187.82
  16    34496.38  0.9499648  18131.94

RMSE was used to select the optimal model using the smallest value.
The final value used for the model was mtry = 7.
```

Figure 5.0.11: Out of bag metrics

6 Discussion

The state of the original data set provided us with many challenges to overcome and issues to contend with. Our largest source of error stems from the fact that our models were trained on purely imputed sale prices. Ideally, if this data had been filled, our models certainly would have performed better. Of the three main types of Machine Learning error, we have strong reason to believe that error due to ignorance is the culprit of high out of sample error. Error due to Misspecification and Estimation are relatively low with the use of Bagged Random Forests, but certainly can be improved as well. We would like to note that a key take-away from this modeling process is which features are relevant when considering housing sale prices. Using the results from our feature selection and tree models, we might begin to think of other higher quality features. Our $median_income$ feature augmented from the census proved to be valuable and so we might consider other factors such as crime rates, poverty rates, distance from public transit and so on. Although these are not direct causal drivers of sale price, they stand in for attributes which people consider when purchasing a home. We can say with confidence that our models will not outperform those created by companies such as Zillow, but they do grant us an inside view of their possible feature sets and algorithms. Despite being a mediocre model, we believe implementing some of the changes discussed here would yield a usable model for amateur prediction.

Acknowledgements

1) Minecraft Forum: Assisted with Java garbage collection options to prevent overflow of memory pools.

2) Stack Overflow: Syntax errors resolved through posts.

3) RStudio Community: RStudio related errors resolved through posts.

Code Appendix

````
Packages/Libraries & Setup
```{r,set.seed(342)}
#Set cache for seed
knitr::opts_chunk$set(cache = T)
#Memory allocation for Java ~10gb and Garbage Collection
options(java.parameters = c("-XX:+UseConcMarkSweepGC", "-Xmx10000m"))
#Packages to load
pacman::p_load(
  ggplot2,
  tidyverse,
  data.table,
  R.utils,
  magrittr,
  dplyr,
  testthat,
  YARF,
  lubridate,
  missForest,
  parallel,
  doParallel,
  caret,
  glmnet
)
#Set CPU cores for YARF
num_of_cores = 8
set_YARF_num_cores(num_of_cores)
#Initialize rJava
library(rJava)
gc()
.jinit()
```


The Data
```{r}
#Set our file path & read in file
housingDataFilePath =
↪  "/home/peterjr/RepoCollections/MATH_342W_FinalProject/Datasets/housing_data_2016_201
#Keep a unaltered "True" copy
housingDataTrue = data.table(fread(housingDataFilePath))
housingData = housingDataTrue
housingData
#Relevant columns begin at the column labeled (URL)
````
````

```

Initial Data Preparation I (Dropping Irrelevant Columns & Storing Possible
↪  Ones for Later Use)
```{r}
#Dropping Mturk columns that are not relevant to our housing model
housingData[,c(1:27):=NULL]
#Save the urls in case they are needed
housingURLS = housingData[,.(URL)]
#Dropping URL from the data table
housingData[,URL:=NULL]
#Dropping other useless url column from data table (ALL NA's)
housingData[,url:=NULL]
#Dropping model_type because similar information is contained in other
↪  columns
housingData[,model_type:=NULL]
housingData
```


Initial Data Preparation II (Writing some notes about Columns)
```{r}
#Getting the column names to write some notes about each column
names(housingData)
#Getting some general information about the table
summary(housingData)
```

Data Cleaning I (Symbol Removal & Establishing Column Types)
```{r}
#First lets deal with the String columns that have $ symbols and convert to
↪  integer
#Extract dollar sign columns as subset to operate on
dollarSymbolSubset =
↪  housingData[,.(common_charges,maintenance_cost,parking_charges,sale_price,total_taxe
#Remove dollar signs based on pattern matching
dollarSymbolSubset[] =
↪  lapply(dollarSymbolSubset,gsub,pattern="$",fixed=TRUE,replacement="")
#Also Remove any commas that may appear for large values
dollarSymbolSubset[] =
↪  lapply(dollarSymbolSubset,gsub,pattern=",",fixed=TRUE,replacement="")
#Replace the columns in housing Data with the new dollarSymbolSubset
housingData[,c("common_charges","maintenance_cost","parking_charges","sale_price","total
```
```

```
↪ dollarSymbolSubset[,c("common_charges","maintenance_cost","parking_charges","sale_pr
#Now we need to convert these columns in housing data to integer type
housingData[,c("common_charges","maintenance_cost","parking_charges","sale_price","total
↪ =
↪ lapply(housingData[,c("common_charges","maintenance_cost","parking_charges","sale_pr
↪ as.numeric)
#########################################################################
#Second lets deal with changing cats_allowed and dogs_allowed to factors
housingData[,sum(is.na(cats_allowed))] # No NA values for cats_allowed
housingData[,sum(is.na(dogs_allowed))] # No NA values for dogs_allowed
#Changing to factors for cats and dogs allowed
unique(housingData[,cats_allowed]) # 3 "unique" values
#Lets deal with the y instead of a yes
housingData$cats_allowed[grepl("y", housingData$cats_allowed)] = "yes"
length(unique(housingData[,cats_allowed])) # 2 unique values
#Lets do the same for dogs
unique(housingData[,dogs_allowed]) # 3 "unique" values"
housingData$dogs_allowed[grepl("yes89", housingData$dogs_allowed)] = "yes"
length(unique(housingData[,cats_allowed])) # 2 unique values
#Factor them
housingData[,c("cats_allowed","dogs_allowed")] =
↪ lapply(housingData[,c("cats_allowed","dogs_allowed")], as.factor)
levels(housingData$cats_allowed) #Check levels
levels(housingData$dogs_allowed) #Check levels
#########################################################################
#Third lets deal with other String columns to be factored (track NA's for
↪ later)
housingData[,sum(is.na(coop_condo))] # No NA values for coop_condo
length(unique(housingData[,coop_condo])) # 2 unique values
#Factor it
housingData[,coop_condo := factor(coop_condo)]
levels(housingData$coop_condo)
housingData[,sum(is.na(dining_room_type))] # 448 NA values for
↪ dining_room_type
length(unique(housingData[,dining_room_type])) # 6 unique values including
↪ NA
length(which(housingData$dining_room_type == "none")) #none occurs 2 times
length(which(housingData$dining_room_type == "dining area")) #dining area
↪ occurs 2 times
#Lets deal with the issue of "dining area" as the room type and consider it
↪ as type other
housingData$dining_room_type[grepl("dining area",
↪ housingData$dining_room_type)] = "other"
```

```
housingData$dining_room_type[grepl("none", housingData$dining_room_type)] =
↪  "other"
length(unique(housingData[,dining_room_type])) # 4 unique values including
↪  NA
housingData[,dining_room_type := factor(dining_room_type)]
levels(housingData$dining_room_type)
housingData[,sum(is.na(fuel_type))] # 112 NA values for dining_room_type
length(unique(housingData[,fuel_type])) # 7 "unique" values including NA
#Lets deal with the capitalization issues for fuel_typenone
housingData[,fuel_type := tolower(fuel_type)]
housingData$fuel_type[grepl("none", housingData$fuel_type)] = "other"
length(unique(housingData[,fuel_type])) # r unique values including NA
housingData[,fuel_type := factor(fuel_type)]
levels(housingData$fuel_type)
housingData[,sum(is.na(kitchen_type))]# 16 NA values for dining_room_type
length(unique(housingData[,kitchen_type])) # 14 "unique" values including NA
#Lets deal with the upper case lower case kitchen type differences
housingData[,kitchen_type:=tolower(kitchen_type)] # Lowercase everything to
↪  pattern match
length(unique(housingData[,kitchen_type])) # 11 "unique" values including NA
#Lets now deal with spaces creating more unique values
housingData[,kitchen_type := lapply(kitchen_type,gsub,pattern="eat
↪  in",fixed=TRUE,replacement="eatin")]
length(unique(housingData[,kitchen_type])) # 10 "unique" values including NA
#Lets lets deal with the misspellings of efficiency kitchen
housingData$kitchen_type[grepl("effic", housingData$kitchen_type)] =
↪  "efficiency"
length(unique(housingData[,kitchen_type])) # 6 unique values including NA
#Finally lets deal with 1955 and replace that with NA -> I am assuming here
↪  1955 is wrong and not a type of kitchen
housingData[, kitchen_type := sapply(kitchen_type, function(x) replace(x,
↪  which(x=="1955"), NA))]
length(unique(housingData[,kitchen_type])) # t unique values including NA
↪  (no 1955 -> NA)
housingData[,kitchen_type := factor(kitchen_type)]
levels(housingData$kitchen_type)
#############################################################################
#Fourth lets deal with the Garage column (track NA's for later)
housingData[,sum(is.na(garage_exists))] # 1826 NA values for garage exists
length(unique(housingData[,garage_exists])) # 7 "unique" values
#Lets deal with the capitalization and misspelling of yes
housingData[,garage_exists := tolower(garage_exists)]
housingData$garage_exists[grepl("y", housingData$garage_exists)] = "yes"
length(unique(housingData[,garage_exists])) # 5 unique values including NA
#Lets treat underground and ug as yes
```

```
housingData$garage_exists[grepl("u", housingData$garage_exists)] = "yes"
length(unique(housingData[,garage_exists])) # 3 unique values including NA
#Lets treat 1 as a yes
housingData$garage_exists[grepl("1", housingData$garage_exists)] = "yes"
length(unique(housingData[,garage_exists])) # 2 unique values including NA
#Fill NA's in garage with No's -> Use 1s in missingness to indicate this
 ↪  later om.
housingData[, c("garage_exists")][is.na(housingData[, c("garage_exists")])]
 ↪  = "no"
housingData[,c("garage_exists")] = lapply(housingData[,c("garage_exists")],
 ↪  as.factor)
#setattr(housingData$garage_exists,"levels",c("no","yes"))
#housingData[,garage_exists := factor(garage_exists)]
levels(housingData$garage_exists)
################################################################################
#Fifth lets take the date column treat is a an unordered factor
#In order to limit the total number of levels in Date, lets just grabs the
 ↪  months
#We sacrifice some granularity, but hopefully this generalize better
housingData$date_of_sale = format(as.Date(housingData$date_of_sale,
 ↪  format="%m/%d/%Y"),"%m")
housingData[,date_of_sale:= factor(date_of_sale,ordered=FALSE)]
length(unique(housingData[,date_of_sale])) #13 including NA which is what we
 ↪  want
#Lets take a look at our data set now
ncol(housingData)
summary(housingData)
```

Data Manipulation I (Creating New Features)

```{r}
#First lets just add up all the charges into a single column
#Assign new column totalCharges to be the row sum of the chargeCols ignoring
 ↪  NA's
housingData[, totalCharges := rowSums(.SD,na.rm=TRUE), .SDcols =
 ↪  c("common_charges","maintenance_cost","parking_charges","total_taxes")][]
housingData[,sum(is.na(totalCharges))] # No NA's here which is good since
################################################################################
#Second lets extract the zip codes and assign them to their own column
#Lets use a regular expression to extract the zip code out of this field
housingData[,zip_code :=
 ↪  substr(str_extract(full_address_or_zip_code,"[0-9]{5}"),1,5)]
housingData[,zip_code := as.numeric(zip_code)]
```

```
#We can now drop the full_address column since we wont need that
housingData[,full_address_or_zip_code := NULL]
#############################################################################
#Third lets add up full and half bathrooms
#Lets divide the half bathroom columns by 2 so that when we add them it is
 ↪  more granular
housingData[,num_half_bathrooms:=num_half_bathrooms/2]
#Assign a new column to represent the total number of bathrooms
housingData[,totalBathrooms :=rowSums(.SD,na.rm=TRUE), .SDcols =
 ↪  c("num_full_bathrooms","num_half_bathrooms")][]
#############################################################################
#Fourth lets bring in some extra data that shows median income by zipcode
queensIncomeDataFilePath =
 ↪  "/home/peterjr/RepoCollections/MATH_342W_FinalProject/Datasets/income_queens_2016.cs
queensIncomeData = data.table(read.csv(queensIncomeDataFilePath))
#Grab columns we want and remove the first row description of columns
queensIncomeData = queensIncomeData[-1,.(GEO_ID,S1901_C01_012E)]
#Change Data Type
queensIncomeData[,zip_code := as.numeric(GEO_ID)]
#Rename median income column
setnames(queensIncomeData, "S1901_C01_012E", "median_income")
queensIncomeData[,median_income := as.numeric(median_income)]
#Drop the geo_id column
queensIncomeData[,GEO_ID := NULL]
#############################################################################
#Fifth lets join this to our housing data on the zipcode
#We are doing a left join because I want everything in housing preserved ->
 ↪  median income can be imputed
housingData = left_join(housingData,queensIncomeData,by.x = "zip_code",by.y
 ↪  = "zip_code")
housingData[,sum(is.na(median_income))] # 64 NA values, not bad since most
 ↪  are getting filled, should be easy to impute
```

Initial Data Exploration I (Basic Visualization & Stats)

```{r}
#############################################################################
#First lets take a look at sale_price. It is important we understand this
 ↪  since it is our response
sale_density = ggplot(housingData)+
  geom_density(aes(x=sale_price)) # From here we can see a concentration
 ↪  around ~ 225k
sale_density
```

```
##############################################################################
#Second lets take a look at some basic statistics about sale_price
sd(housingData$sale_price,na.rm = TRUE)
median(housingData$sale_price,na.rm = TRUE)
mean(housingData$sale_price,na.rm = TRUE) # Mean higher than median makes
 ↪  sense with tail in graph above
min(housingData$sale_price,na.rm = TRUE)
max(housingData$sale_price,na.rm = TRUE)
##############################################################################
#Third lets look at some of the columns against sale_price
#I am looking at columns that I need will have the biggest influence on
 ↪  sale_price
bedrooms_sale = ggplot(housingData)+
  geom_point(aes(x=num_bedrooms, y=sale_price))# Looking at num_bedrooms VS
 ↪  sale_price
cats_sale = ggplot(housingData)+
  geom_point(aes(x=cats_allowed, y=sale_price)) # Looking at cats_allowed VS
 ↪  sale_price
dogs_sale = ggplot(housingData)+
  geom_point(aes(x=dogs_allowed, y=sale_price)) # Looking at dogs_allowed VS
 ↪  sale_price
#This is a feature we created from num_full_bathrooms +
 ↪  (num_half_bathrooms)/2
bathroooms_sale = ggplot(housingData)+
  geom_point(aes(x=totalBathrooms, y=sale_price)) # Looking at
 ↪  totalBathrooms VS sale_price
#This is a feature we created by adding up all of the chargest columns
charges_sale = ggplot(housingData)+
  geom_point(aes(x=totalCharges, y=sale_price)) # Looking at totalCharges VS
 ↪  sale_price
walk_sale = ggplot(housingData)+
  geom_point(aes(x=walk_score, y=sale_price)) # Looking at walk_score VS
 ↪  sale_price
cats_sale
dogs_sale
bedrooms_sale
bathroooms_sale
charges_sale
walk_sale
```


Initial Data Exploration II (Better visualizations)
```{r}
```

```
ggplot(data=subset(housingData, !is.na(sale_price))) +
  aes(x = sale_price) +
  geom_histogram(bins = 50L, fill = "blue")+
  geom_vline(data = subset(housingData, !is.na(sale_price)), aes(xintercept
↪  = mean(sale_price)), color = "red",size=1)+
  annotate("text", x=290000, y=45, label=paste("Mean"),size=4.1,angle=90)+
  geom_vline(data = subset(housingData, !is.na(sale_price)), aes(xintercept
↪  = median(sale_price)), color = "yellow",size=1)+
  annotate("text", x=230000, y=45, label=paste("Median"),size=4.1,angle=90)+
  labs(x = "Sale Price", y = "Frequency")+
  ggtitle("Histogram of Sale Price")+
  theme(plot.title = element_text(hjust = 0.5))

#Uncomment the following line if we want to save this picture to our
↪  notebook directory
#gsave("SalePriceHist.png",width=6, height=4,dpi=400)
```

Establishing a Missingness Table

```{r}
#############################################################################
#First lets grab the columns that are of interest to us
housingData =
↪  housingData[,.(approx_year_built,cats_allowed,community_district_num,coop_condo,date

↪  dogs_allowed,fuel_type,garage_exists,kitchen_type,num_bedrooms,num_floors_in_buildin

↪  sale_price,sq_footage,walk_score,totalCharges,zip_code,median_income)]
#############################################################################
#Second lets build up our missing table 0/1 where 1 indicates a NA value in
↪  the housingData
#Create a missing data table and fill with zeros
colNames = names(housingData)
missRows = nrow(housingData)
missCols = ncol(housingData)
missingData = setNames(data.table(matrix(0,nrow = missRows, ncol =
↪  missCols)), colNames)
setnames(missingData,1:ncol(missingData),
↪  paste0(names(missingData)[1:ncol(missingData)], '_miss'))
#Data Set with 1s indicating missing in housingData
missingData[is.na(housingData)] = 1
#Due to the nature of the construction of the missing table, all columns in
↪  housingData have a corresponding *_miss column
```

```
#This may not be entirely accurate, since some of our columns in housingData
 ↪  have no NA's thus the *_miss column will be all 0's
#Remove missing columns where the sum is 0. Implies housingData did not have
 ↪  any NAs.
checkZero= function(x){
    if(sum(x)==0){
      TRUE
    }
}
length(missingData[,sapply(missingData,  checkZero)]) # 7 columns with no
 ↪  missingness, we will drop these, since no imputation will occur here
missingData = missingData[, colSums(missingData != 0) > 0, with = FALSE]
#Lets also drop missingness for sale_price. This will be made clear later,
 ↪  but since we plan on training on all of the imputed sale prices
#our missing will be all 1's aka a zero variance feature.
missingData = missingData[,!c("sale_price_miss")]
#Lets mark the indices where sale price is missing for reasons that will be
 ↪  made clear later
salePriceMissingIndices = which(is.na(housingData$sale_price))
salePriceFilledIndices = which(!is.na(housingData$sale_price))
```

Imputation Using The MissForest Algorithm

```{r}
#############################################################################
#Lets impute our data set including sale price
imputeSet = housingData
#Setting up parallelization cluster
cluster = makePSOCKcluster(num_of_cores)
registerDoParallel(cluster)
#Initialize the missForest algorithm with 100 trees and parallelization
Ximp = missForest(imputeSet,verbose = TRUE, maxiter = 5, ntree = 100,
 ↪  parallelize = "variables")
#Stop the cluster
stopCluster(cluster)
registerDoSEQ()
#Get our final imputed Dataset and bind it to the missiningness table
finalHousingData = cbind(Ximp$ximp,missingData)
finalHousingData
Ximp$OOBerror
```

Establishing Holdout Set I

```{r}
#Prior to any feature selection/modeling we want to establish a hold out set
 ↪  from our finalHousing Data
#We do this so that we can truly consider our hold out test set to be
 ↪  independent from any of the processes we do below
holdout_K=5
holdout_prop = 1 / holdout_K
#Where sale price was NA prior to imputing  ~ 75% of ALL data
salePriceNA_Data = finalHousingData[salePriceMissingIndices,]
#This is crucial to note since our errors will be more honest albeit larger.
#If we test on imputed data we are essentially computing prediction error on
 ↪  a prediction rather than real data
#Most likely this will result in worse error, but it will generalize better
 ↪  in the real world.
#Where sale price was not NA ~ 25% of ALL data
salePriceFilled_Data = finalHousingData[salePriceFilledIndices,]
#Training & Testing data (All Features)
finalHousingData_Train = salePriceNA_Data
finalHousingData_Test = salePriceFilled_Data
X_all_holdout = finalHousingData_Test[,!c("sale_price")]
y_all_holdout = finalHousingData_Test$sale_price
finalHousingData_Train
finalHousingData_Test
```

Feature Importance

```{r}
#Setting up parallelization cluster
cluster = makePSOCKcluster(num_of_cores)
registerDoParallel(cluster)
################################################################
#Evaluating Feature Importance
# 5 fold cross validation repeated 2 times
control_selection =  rfeControl(functions=rfFuncs, method="repeatedcv",
 ↪  number=5,repeats=2)
#We want to train it on the entire data just so we can see what subset of
 ↪  features are the best (excluding sale price since this is our response)
trained_selection =
 ↪  rfe(data.matrix(finalHousingData_Train[,!c("sale_price")]),data.matrix(finalHousingD
#Stop the cluster
stopCluster(cluster)
```

```
registerDoSEQ()
#Uncomment the following line to see a printout of the trained selection
#print(trained_selection)
predictors(trained_selection)
#Plot our RMSE by the number of variables
ggplot(data = trained_selection)+theme_bw()
feat_Importance = data.frame(feature = row.names(varImp(trained_selection)),
↪  importance = varImp(trained_selection)[,1])
ggplot(data = feat_Importance,
↪  aes(x=reorder(feature,-importance),y=importance ,fill = feature))+
  geom_bar(stat="identity")+
  labs(x = "Features", y = "Variable Importance")
```

Contending With Collinear Features

```{r}
#Lets build a table consisting of only numeric values from finalHousingData
numericOnlyData2 = finalHousingData_Train[ , .SD, .SDcols = is.numeric]
ncol(numericOnlyData2) # total numeric columns
#We expect at most p perfect collinearities in our pxp correlation matrix
↪  when i==j
#Greater than p columns indicates that there is perfect collinearity when
↪  i!=j
correlationMatrix2 = as.matrix(cor(numericOnlyData2))
length(which(correlationMatrix2==1))
```

Feature Selection (Using Results From Feature Importance & Collinearity
↪ Exploration)

```{r}
#Here we implement feature selection based on the results provided from RFE
↪  and our test of perfect linearity between features
#This was done in an effort to reduce the noise produced by irrelevant
↪  features in the  hopes of reducing model prediction error
#Let's get a list of our features ranked by importance from the previous
↪  cell
varImp(trained_selection)
#Thinking about this logically, it would be wise to retain sale_price_miss
↪  for the following reasons.
#For starters, sale_price was imputed and so it would be wise to retain a
↪  marker indicating this
```

```
#The sale price missing leads to there being no date of sale. No date of
↪  sale can just mean that is was never marked but a sale may have still
↪  occured
#This is a judgement call here and we choose to retain sale_price_miss
#Get the subset of features from the trained selection
subsetF = c(predictors(trained_selection))
#Create a secondary finalHousingData table with only our selected features &
↪  response
finalHousingDataImpFeat_Train = finalHousingData_Train[,..subsetF]
#Add back our sale price and sale price missingsince subsetF did not include
↪  sale_price as it was excluded from feature importance due to it being
↪  our response
finalHousingDataImpFeat_Train[,sale_price :=
↪  finalHousingData_Train[,c("sale_price")]]
```

Establishing Holdout Set II

```{r}
#Since we are creating a secondary data set with only our selected features
↪  we want to use the same holdout set created above without unselected
↪  features
#We do this so that we can truly consider our hold out test set on the sub
↪  features to be independent from any of the processes we do below
finalHousingDataImpFeat_Test = finalHousingData_Test[,..subsetF]
#Add back our sale price since subsetF did not include sale_price as it was
↪  excluded from feature importance due to it being our response
finalHousingDataImpFeat_Test[,sale_price :=
↪  finalHousingData_Test[,c("sale_price")]] #This is our holdout set here
X_imp_holdout = finalHousingDataImpFeat_Test[,!c("sale_price")]
y_imp_holdout = finalHousingDataImpFeat_Test$sale_price
finalHousingDataImpFeat_Train
finalHousingDataImpFeat_Test
```

Quick Check on our Full Feature Set and Important Feature Set

```{r}
#Ensure the rows in both are the same...columns will obviously be different
↪  since *ImpFeat* contains less features
setequal(dim(finalHousingData_Train)[1],
↪  dim(finalHousingDataImpFeat_Train)[1])
```

```
setequal(dim(finalHousingData_Test)[1],
↪  dim(finalHousingDataImpFeat_Test)[1])
```

Splitting Sets Into Train & Test

```{r}
#Let's leave ~20% of our total data for testing
K=5
prop = 1 /K
#All Feature Set
#Training & Testing data (All Features)
trainIndices_all = sample(1 : nrow(finalHousingData_Train), round((1 - prop)
↪  * nrow(finalHousingData_Train)))
testIndices_all =  setdiff(1 : nrow(finalHousingData_Train),
↪  trainIndices_all)
finalHousingData_subTrain = finalHousingData_Train[trainIndices_all,]
finalHousingData_subTest = finalHousingData_Train[testIndices_all,]
X_train_all= finalHousingData_subTrain[,!c("sale_price")]
y_train_all = finalHousingData_subTrain$sale_price
X_test_all = finalHousingData_subTest[,!c("sale_price")]
y_test_all = finalHousingData_subTest$sale_price
#########################################################################
#Important Feature Set
#Training & Testing data (Important Features)
trainIndices_imp = sample(1 : nrow(finalHousingDataImpFeat_Train), round((1
↪  - prop) * nrow(finalHousingDataImpFeat_Train)))
testIndices_imp = setdiff(1 : nrow(finalHousingDataImpFeat_Train),
↪  trainIndices_imp)
finalHousingDataImpFeat_subTrain =
↪  finalHousingDataImpFeat_Train[trainIndices_imp,]
finalHousingDataImpFeat_subTest =
↪  finalHousingDataImpFeat_Train[testIndices_imp]
X_train_imp= finalHousingDataImpFeat_subTrain[,!c("sale_price")]
y_train_imp = finalHousingDataImpFeat_subTrain$sale_price
X_test_imp = finalHousingDataImpFeat_subTest[,!c("sale_price")]
y_test_imp = finalHousingDataImpFeat_subTest$sale_price
```

Quick Check For Above Cell

```{r}


```

```
setequal((dim(finalHousingData_subTrain)[1]+dim(finalHousingData_subTest)[1]),
↪  dim(finalHousingData_Train)[1])
setequal((dim(finalHousingDataImpFeat_subTrain)[1]+dim(finalHousingDataImpFeat_subTest)[
↪  dim(finalHousingDataImpFeat_Train)[1])
```

Linear Regression Model (Full Data Set)

```{r}
#Lets run a traditional OLS with all of our features
lin_mod_all = lm(y_train_all~.,X_train_all,x = TRUE, y = TRUE)
#Test set performance
yHats_OLS_test_all = predict(lin_mod_all,X_test_all)
oosRMSE_OLS_test_all =
↪  sqrt(sum((y_test_all-yHats_OLS_test_all)^2)/length(y_test_all))
#Hold out set performance
yHats_OLS_holdout_all = predict(lin_mod_all,X_all_holdout)
oosRMSE_OLS_holdout_all =
↪  sqrt(sum((y_all_holdout-yHats_OLS_holdout_all)^2)/length(y_all_holdout))
oosRMSE_OLS_test_all
oosRMSE_OLS_holdout_all
#Notice we are being warned about a rank deficiency in our full feature data
↪  set. This is expected since the features are too closely correlated
#We should not trust the first value because of this
```

Linear Regression Model (Sub Data Set)

```{r}
#Lets run a traditional OLS with all of our features
lin_mod_imp = lm(y_train_imp~.,X_train_imp,x = TRUE, y = TRUE)
#Test set performance
yHats_OLS_test_imp = predict(lin_mod_imp,X_test_imp)
oosRMSE_OLS_test_imp =
↪  sqrt(sum((y_test_imp-yHats_OLS_test_imp)^2)/length(y_test_imp))
#Hold out set performance
yHats_OLS_holdout_imp = predict(lin_mod_imp,X_imp_holdout)
oosRMSE_OLS_holdout_imp =
↪  sqrt(sum((y_imp_holdout-yHats_OLS_holdout_imp)^2)/length(y_imp_holdout))
SSR_olsImp_Holdout = sum((y_imp_holdout - yHats_OLS_holdout_imp) ^ 2)  ##
↪  residual sum of squares
SST_olsImp_Holdout = sum((y_imp_holdout - mean(y_imp_holdout)) ^ 2)  ##
↪  total sum of squares
Rsq_olsImp_Holdout = 1 - SSR_olsImp_Holdout/SST_olsImp_Holdout
```

```
lin_mod_imp$coefficients
oosRMSE_OLS_test_imp
oosRMSE_OLS_holdout_imp
Rsq_olsImp_Holdout
```

Cross Validated Linear Model (Full & Sub Data Set)

```{r}
train_cv = trainControl(method = "cv", number = K)
#Create a model that is cross validated on the training portion of our all
 ↪  feature data
ols_all_cv = train(sale_price~.,
 ↪  data=data.matrix(finalHousingData_subTrain),method="lm", trControl =
 ↪  train_cv)
#Create a model that is cross validated on the training portion of our
 ↪  important feature data
ols_imp_cv = train(sale_price~.,
 ↪  data=data.matrix(finalHousingDataImpFeat_subTrain),method="lm",
 ↪  trControl = train_cv)
#Predict for both models
yHats_OLS_all_cvTest = predict(ols_all_cv,data.matrix(X_test_all))
yHats_OLS_imp_cvTest = predict(ols_imp_cv,data.matrix(X_test_imp))
#Test set performance
oosRMSE_OLS_all_cvTest =
 ↪  sqrt(sum((y_test_all-yHats_OLS_all_cvTest)^2)/length(y_test_all)) #Here
 ↪  there is no difference between y_test and y_test_sub
oosRMSE_OLS_imp_cvTest =
 ↪  sqrt(sum((y_test_imp-yHats_OLS_imp_cvTest)^2)/length(y_test_imp)) #It is
 ↪  done merely for consistency in var names
#Predict for both models
yHats_OLS_all_cvHoldout = predict(ols_all_cv,data.matrix(X_all_holdout))
yHats_OLS_imp_cvHoldout = predict(ols_imp_cv,data.matrix(X_imp_holdout))
#Hold out set performance
oosRMSE_OLS_all_cvHoldout =
 ↪  sqrt(sum((y_all_holdout-yHats_OLS_all_cvHoldout)^2)/length(y_all_holdout))
oosRMSE_OLS_imp_cvHoldout =
 ↪  sqrt(sum((y_imp_holdout-yHats_OLS_imp_cvHoldout)^2)/length(y_imp_holdout))
SSR_olsImp_cvHoldout = sum((y_imp_holdout - yHats_OLS_imp_cvHoldout) ^ 2)
 ↪  ## residual sum of squares
SST_olsImp_cvHoldout = sum((y_imp_holdout - mean(y_imp_holdout)) ^ 2)  ##
 ↪  total sum of squares
Rsq_olsImp_cvHoldout = 1 - SSR_olsImp_cvHoldout/SST_olsImp_cvHoldout
oosRMSE_OLS_all_cvTest
```

```
oosRMSE_OLS_all_cvHoldout
oosRMSE_OLS_imp_cvTest
oosRMSE_OLS_imp_cvHoldout
Rsq_olsImp_cvHoldout
#Notice we are being warned about a rank deficiency in our full feature data
 ↪  set. This is expected since the features are too closely correlated
#We should not trust the first two values because of this
```

Linear Regression Model Cross Validated Lasso (Full Dataset)

```{r}
#This is mainly for fun to see how a cross validated Lasso Regression Model
 ↪  can tame the rank deficiency in our full feature data set
lin_mod_lasso =
 ↪  cv.glmnet(data.matrix(X_train_all),y_train_all,nfolds=K,alpha = 1)
opt_Lambda = lin_mod_lasso$lambda.min
#Test Performance
yHats_LassoTest = predict(lin_mod_lasso, data.matrix(X_test_all),s =
 ↪  opt_Lambda)
oosRMSE_Lasso_Test =
 ↪  sqrt(sum((y_test_all-yHats_LassoTest)^2)/length(y_test_all))
#Holdout Set Performance
yHats_LassoHoldout = predict(lin_mod_lasso, data.matrix(X_all_holdout),s =
 ↪  opt_Lambda)
oosRMSE_Lasso_Holdout =
 ↪  sqrt(sum((y_all_holdout-yHats_LassoHoldout)^2)/length(y_all_holdout))
SSR_lasso_cvHoldout = sum((y_imp_holdout - yHats_LassoHoldout) ^ 2)  ##
 ↪  residual sum of squares
SST_lasso_cvHoldout = sum((y_imp_holdout - mean(y_imp_holdout)) ^ 2)  ##
 ↪  total sum of squares
Rsq_lasso_cvHoldout = 1 - SSR_lasso_cvHoldout/SST_lasso_cvHoldout
oosRMSE_Lasso_Test
oosRMSE_Lasso_Holdout
Rsq_lasso_cvHoldout
#At this point we will stop using the full feature data and stick with our
 ↪  important feature data set
```

Regression Tree Model (Important Feature Data Set)

```{r}
#Lets fit a regression tree to our important feature set
```

```
regTree_mod = YARFCART(X_train_imp, y_train_imp, calculate_oob_error =
↪  FALSE)
#Test performance
yHats_RegTree_Test = predict(regTree_mod,X_test_imp)
oosRMSE_RegTree_Test =
↪  sqrt(sum((y_test_imp-yHats_RegTree_Test)^2)/length(y_test_imp))
#Holdout Set Performance
yHats_RegTree_Holdout = predict(regTree_mod,X_imp_holdout)
oosRMSE_RegTree_Holdout =
↪  sqrt(sum((y_imp_holdout-yHats_RegTree_Holdout)^2)/length(y_imp_holdout))
SSR_regTree_Holdout = sum((y_imp_holdout - yHats_RegTree_Holdout) ^ 2)  ##
↪  residual sum of squares
SST_regTree_Holdout = sum((y_imp_holdout - mean(y_imp_holdout)) ^ 2)  ##
↪  total sum of squares
Rsq_regTree_Holdout = 1 - SSR_regTree_Holdout/SST_regTree_Holdout
#Uncomment the following line to save an illustration of the tree
#illustrate_trees(regTree_mod, max_depth=5, open_file=TRUE)
oosRMSE_RegTree_Test
oosRMSE_RegTree_Holdout
Rsq_regTree_Holdout
```

Random Forest Model (Important Feature Data Set)

```{r}
#Lets fit a random Forest to our important feature set
rf_mod = YARF(X_train_imp, y_train_imp, calculate_oob_error = FALSE)
#Test performance
yHats_rf_Test = predict(rf_mod,X_test_imp)
oosRMSE_rf_Test = sqrt(sum((y_test_imp-yHats_rf_Test)^2)/length(y_test_imp))
#Holdout Set Performance
yHats_rf_Holdout = predict(rf_mod,X_imp_holdout)
oosRMSE_rf_Holdout =
↪  sqrt(sum((y_imp_holdout-yHats_rf_Holdout)^2)/length(y_imp_holdout))
SSR_rf_Holdout = sum((y_imp_holdout - yHats_rf_Holdout) ^ 2)  ## residual
↪  sum of squares
SST_rf_Holdout = sum((y_imp_holdout - mean(y_imp_holdout)) ^ 2)  ## total
↪  sum of squares
Rsq_rf_Holdout = 1 - SSR_rf_Holdout/SST_rf_Holdout
oosRMSE_rf_Test
oosRMSE_rf_Holdout
Rsq_rf_Holdout
```

Bagged Random Forest Model (Important Feature Data Set)

```{r}
#Lets fit a bagged random forest to our important feature set
rfBag_mod = YARFBAG(X_train_imp, y_train_imp, calculate_oob_error = TRUE)
#Out of Bag Performance
oosRMSE_brf_Bag = rfBag_mod$rmse_oob
#Holdout Set Performance
yHats_brf_Holdout = predict(rfBag_mod,X_imp_holdout)
oosRMSE_brf_Holdout =
 ↪  sqrt(sum((y_imp_holdout-yHats_brf_Holdout)^2)/length(y_imp_holdout))
SSR_rfBag_Holdout = sum((y_imp_holdout - yHats_brf_Holdout) ^ 2)  ##
 ↪  residual sum of squares
SST_rfBag_Holdout = sum((y_imp_holdout - mean(y_imp_holdout)) ^ 2)  ## total
 ↪  sum of squares
Rsq_rfBag_Holdout = 1 - SSR_rfBag_Holdout/SST_rfBag_Holdout
oosRMSE_brf_Bag
oosRMSE_brf_Holdout
Rsq_rfBag_Holdout
```

Bagged Random Forest Model Optimization (Hyper-Parameter Tuning)

```{r}
#Hyper-Parameter Tuning
#Setting up parallelization cluster
cluster = makePSOCKcluster(num_of_cores)
registerDoParallel(cluster)
control_rf = trainControl(method='repeatedcv', number=K, repeats=2,search =
 ↪  'random')
mtry = ncol(finalHousingDataImpFeat_subTrain) # Columns in our important
 ↪  feature set
nTree = 500
tunegrid = expand.grid(.mtry=seq(1,mtry))
rf_optimized = train(sale_price~.,
                     data=data.matrix(finalHousingDataImpFeat_subTrain),
                     method='rf',
                     metric='RMSE',
                     tuneGrid=tunegrid,
                     nTree = nTree,
                     trControl=control_rf
                    )
#Stop the cluster
```

```
stopCluster(cluster)
registerDoSEQ()
#Holdout Set Performance
yHats_bgfOpt_Holdout = predict(rf_optimized,data.matrix(X_imp_holdout))
oosRMSE_bgfOpt_Holdout =
 ↪  sqrt(sum((y_imp_holdout-yHats_bgfOpt_Holdout)^2)/length(y_imp_holdout))
SSR_bgfOpt_Holdout = sum((y_imp_holdout - yHats_bgfOpt_Holdout) ^ 2)  ##
 ↪  residual sum of squares
SST_bgfOpt_Holdout = sum((y_imp_holdout - mean(y_imp_holdout)) ^ 2)  ##
 ↪  total sum of squares
Rsq_bgfOpt_Holdout = 1 - SSR_bgfOpt_Holdout/SST_bgfOpt_Holdout
print(rf_optimized)
oosRMSE_bgfOpt_Holdout
Rsq_bgfOpt_Holdout
```


Final Shipped Model Trained On All Data
```{r}
#Hyper-Parameter Tuning
#Setting up parallelization cluster
cluster = makePSOCKcluster(num_of_cores)
registerDoParallel(cluster)
#Lets combine the Train and Test Portion of our important feature data set
 ↪  into a single entity
finalHousingData_ImpFeat =
 ↪  rbind(finalHousingDataImpFeat_Train,finalHousingDataImpFeat_Test)
control_rf = trainControl(method='repeatedcv', number=K, repeats=2,search =
 ↪  'random')
mtry = ncol(finalHousingData_ImpFeat) # Columns in our important feature set
nTree = 500
tunegrid = expand.grid(.mtry=seq(1,mtry))
rf_optimizedFinal = train(sale_price~.,
                      data=data.matrix(finalHousingData_ImpFeat),
                      method='rf',
                      metric='RMSE',
                      tuneGrid=tunegrid,
                      nTree = nTree,
                      trControl=control_rf
                     )
#Stop the cluster
stopCluster(cluster)
registerDoSEQ()
print(rf_optimizedFinal)
```