Draft report 2

Asa Ferguson, Louise Xu, Xinran Liu

2022-12-04

Abstract

We will be investigating the prediction of house prices using recent data from Ames Iowa. We are interested in finding which features are the most relevant in predicting house sale price. We implement several methods of analysis, such as linear regression with subset selection, linear regression with LASSO regularization, and various tree-based methods including random forests. We found that well-tuned random forest and LASSO regression on log-transformed data produce the best results on test data, with the LASSO method surprisingly slightly outperforming random forest.

Introduction

The housing market has always been a trending topic; people are sensitive about how tight the current sales market is and whether it is a good time to buy or sell. According to the PD&R, there are 6120 thousand existing home sales and 771 thousand new home sales in 2021. The average sale price is $347 thousand and $397 thousand for each, approximately 20% higher than the price in 2020. With this highly active and fluctuating market, researchers are making lots of forecasts of pricing, supply, and demand every day based on economic conditions and demographic trends. As for individual builders and mortgagees, housing inventory characteristics are crucial in determining the sale price for each specific property. Road access, configuration, and accessories all affect how much people are willing to pay. Uncovering the key features in determining house sale prices is a difficult task. But we are excited to give house price prediction a shot using the skills we have learned in Math 243.

We are excited to be working with a dataset that is more current and expansive than some of the other famous and accessible datasets containing home price information. As an example, the Boston House Prices dataset that is used in many statistics classes (we even used it in our class!) is sourced from data compiled in 1970. In addition to being a relatively recent dataset, the response variable, home sale price, is incredibly relevant to the lives of people all across the country. While none of us are in a position to buy a house, there are millions of people in the US who are either homeowners or are considering buying. Predicting house prices is useful not only for buyers looking for a new home, but also for owners evaluating their assets and deciding whether to sell. Any insight we gain into the factors that lead to an increase in home price will be exciting, in light of the importance of home pricing to so many people.

A precise statement of our research question is: Which features are the most relevant in predicting house sale price, and which type of model best predicts the house price as a function of those features?

Method

Our dataset is the Ames Housing dataset compiled by Dean De Cock for use in data science education. These data were gathered from Ames Iowa during a period spanning 2006 to 2010. The observational unit is a house sold in Ames from 2006 to 2010. The response variable, house sale price, is quantitative. The dataset includes 79 predictor features, with more than half of our predictors categorical. The predictors range from geographical information about the property and the neighborhood, to interior measurements and quality of

amenities, to sales statistics. The dataset has 1460 observations. We found this dataset through the "House Prices - Advanced Regression Techniques" Kaggle competition, (Montoya 2016).

Several important steps were taken in order to prepare this dataframe for our statistical analysis. The first step was recoding NA's. In the raw .csv file, many of the categorical variables took the value NA frequently. Upon reading the codebook, it became clear that in many of the variables, the value NA actually stored meaning. Hence, whenever NA was listed as a possible value storing meaning for a categorical variable in the codebook, we replaced the value NA with the character value "None" so that our algorithms would not treat these NAs as missing data. Next, the continuous variable `LotFrontage` contained a large number of NA values. This variable records the length of street/sidewalk bordering a home. After some exploratory analysis, it became clear that instead of recording "no access" as 0, it was recorded as an NA. Hence, we recoded NA as 0 in LotFrontage. In any case where the codebook indicated that a variable which took numerical values should be interpreted as categorical (variables such as house legal subclass and month sold), we changed the variable type to character.

The next issue to be addressed was rare levels. Some of the categorical features contained levels that were exceedingly rare. In extreme cases, levels were observed only once or twice. This imbalance in levels created issues in data splitting both for cross validation and for the validation set. Since the problem was so acute, even stratified sampling was not an option. Our solution was to combine the rare levels into a single level "other." We set a threshold of 10 observations in a level. Levels with fewer than ten observations were collapsed into the single "other" category. This solution was designed to balance functionality with the desire to preserve the original data structure. Ten observations in each level gives an expected number of 2.5 observations in each level in the test set and 7.5 in the training set, which means that we likely will have at least one observation from each level in the training set and test set. When all levels within the test set are contained in the training set, our models will be able to make predictions on the test data, which will enable us to draw conclusions about relative model performance.

Even when combining rare levels into one "other" category, there were still categorical variables in which the other variable had less than 10 observations. These rare instances created the same issues with CV and in the training test split, so it was decided that these rare "other" values would be overwritten with the mode of the categorical variable. These rare observations reflect less than 0.68% of the observations in each level, so the alteration to the data from imputing was not at all substantial. To prevent data leakage, when we imputed the mode, we imputed the training mode for training observations and the test mode for test observations. If imputation created a variable that only took one single value, then that variable was removed from the whole dataset. This occurred for the variables `Street`, `Utilities`, and `PoolQC`.

After handling the rare values, there were still some variables which had to be cleaned up. The variable `GarageYrBlt` records the year that the home's garage was built. If the home did not have a garage, this value was stored as NA. These 81 NA's in our dataframe were causing issues in our regressions so we performed some exploratory data analysis to see if we could drop `GarageYrBlt`. We found that `GarageYrBlt` had over 0.8 correlation with `YearBuilt`. Further, the variable `GarageType` took the level "none" whenever a home was missing a garage. This meant that a large majority of the information present in `GarageYrBlt` was stored in other variables. Hence, we dropped `GarageYrBlt`. Next, the variables recording Masonry Veneer Type and Masonry Veneer Area had 8 missing values for the same 8 rows. The mode of "none" (across both the test and training sets!) was imputed for Veneer Type and the value corresponding to "none," 0, was imputed for the Veneer Area. Finally the one remaining categorical variable which truly did take NA values (the NA only stored the information that the data were missing), `Electrical`, took only one NA value and the mode value of the set it landed in was imputed here.

One specter looming over all of this data processing is cluster analysis. A more theoretically justified and effective way to deal with the rare levels phenomenon and collapse the dimensions on our dataframe could be PCA for mixed data (or an analogous decomposition through cluster analysis). By isolating principal components, many of the issues with high dimensionality and rare levels would be resolved and our models would likely be more effective. Unfortunately, we have yet to reach these methods in class, and the authors haven't had too much time outside of class to learn about these methods. The "other" recoding method was the most effective strategy we could come up with using the tools that we have learned about in class. If

we learn about cluster analysis and PCA in time for the final presentation, we will test the best performing method on the processed data (as uncovered in this report) on principal components to see if it performs better.

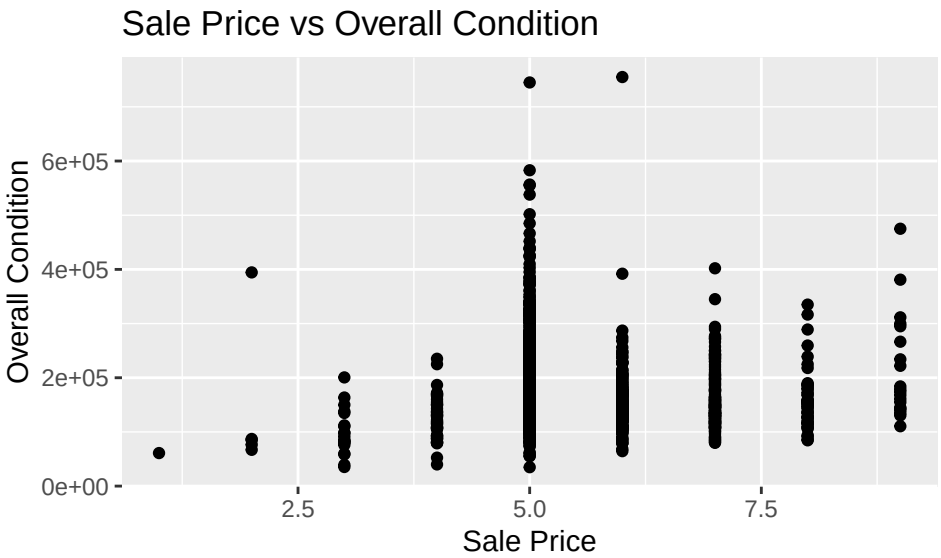
Exploratory

First we can analyze the correlation values between numerical variables. We can see the first few variable pairs with high correlation in the following table, which shows that many predictor variables have a strong linear relation with our response Sale Price. This informs our decision to fit a linear regression model, although we don't anticipate the linear regression model to perform best since the correlation values aren't extremely strong. This also suggests that a more complex model like trees or random forests may perform better at capturing this complex relationship.

```
##      row_name  col_name      corr
## 1 OverallQual SalePrice 0.7899971
## 2   GrLivArea SalePrice 0.7100797
## 3  GarageCars SalePrice 0.6396857
## 4  GarageArea SalePrice 0.6224917
## 5 TotalBsmtSF SalePrice 0.6129709
## 6   X1stFlrSF SalePrice 0.6068494
```

If we look at all of the predictor variables' correlation with Sale Price, we can observe some counter-intuitive trends. For example, the fact that Overall Condition is negatively correlated with Sale Price may seem strange since one would anticipate the price to increase as the overall condition of the house improves. However, if we look closely at the plot between Sale Price vs Overall Condition, we can see that this is a consequence of trying to fit a linear model on a discrete numeric variable. This is another indication that linear regression may not be the best choice, and we need a more complex model to capture these type of complex trends.

```
##        row_name  col_name        corr
## 1  KitchenAbvGr SalePrice -0.13741923
## 2 EnclosedPorch SalePrice -0.12877809
## 3   OverallCond SalePrice -0.07629370
## 4        YrSold SalePrice -0.02618020
## 5  LowQualFinSF SalePrice -0.02526275
## 6       MiscVal SalePrice -0.02095080
```



Next, if we look at the correlation between numeric predictor variables, we will notice that there are many

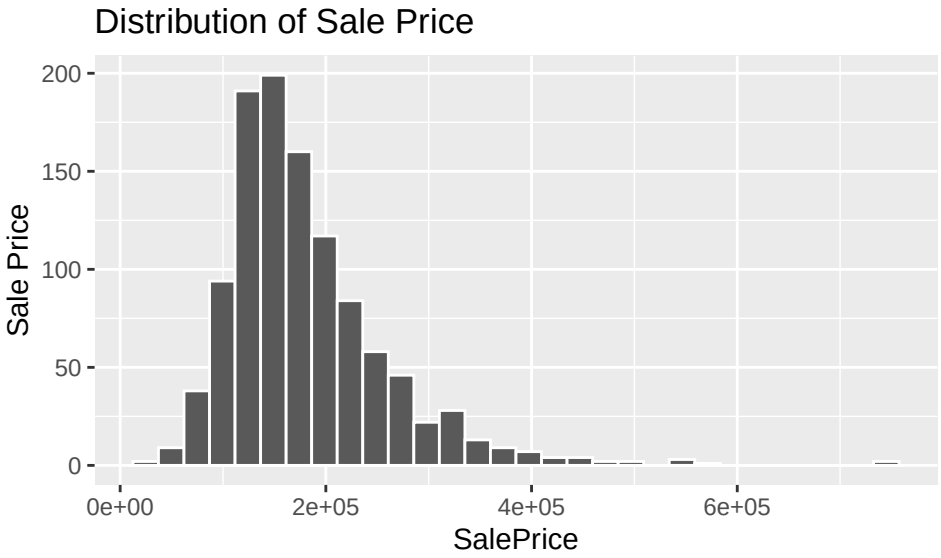
pairs with high correlation. Serious collinearity concerns make fitting a full linear model inadvisable, so we use subset selection methods and LASSO regression methods to address this issue. Tree-based methods, particularly random forests, are also used since they perform better on datasets with correlated predictors. We also try manually removing variables that have over 0.5 correlation and see if that improves the performance of our models.

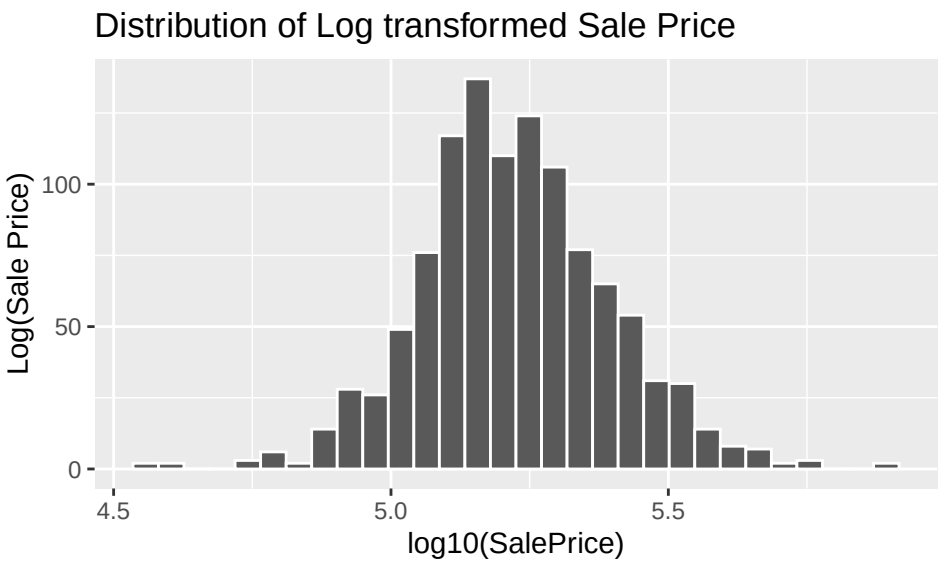
```
##       row_name     col_name      corr
## 1   GarageCars   GarageArea 0.8823316
## 2    GrLivArea TotRmsAbvGrd 0.8254761
## 3  TotalBsmtSF    X1stFlrSF 0.8182463
## 4    X2ndFlrSF    GrLivArea 0.6901776
## 5 BedroomAbvGr TotRmsAbvGrd 0.6755962
## 6   BsmtFinSF1 BsmtFullBath 0.6473462
```

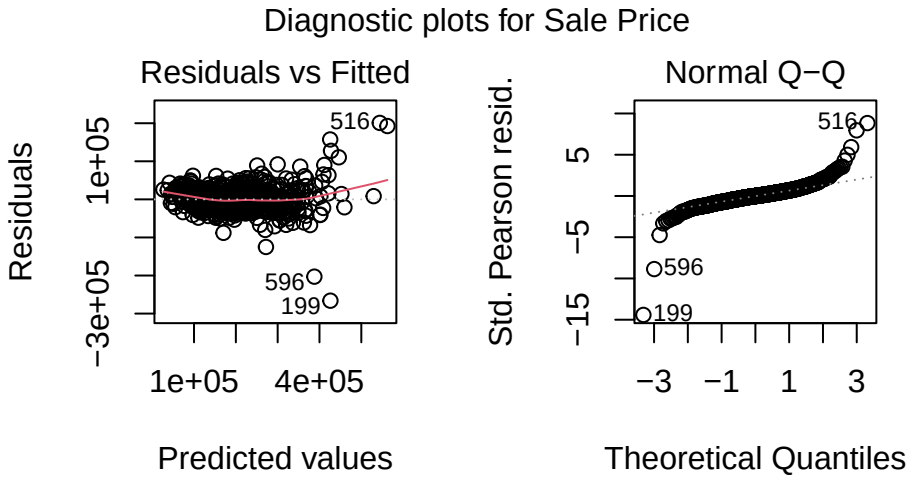
The categorical variables are also problematic for linear regression, especially since some variables like Neighborhood have 24 different levels. Since linear regression creates dummy variables for each level of a categorical variable, linear regression on this data set will soon create numerous amounts of dummy variables. This will make the linear regression models more expensive to fit and harder to interpret, which further certifies the need of model selection methods.

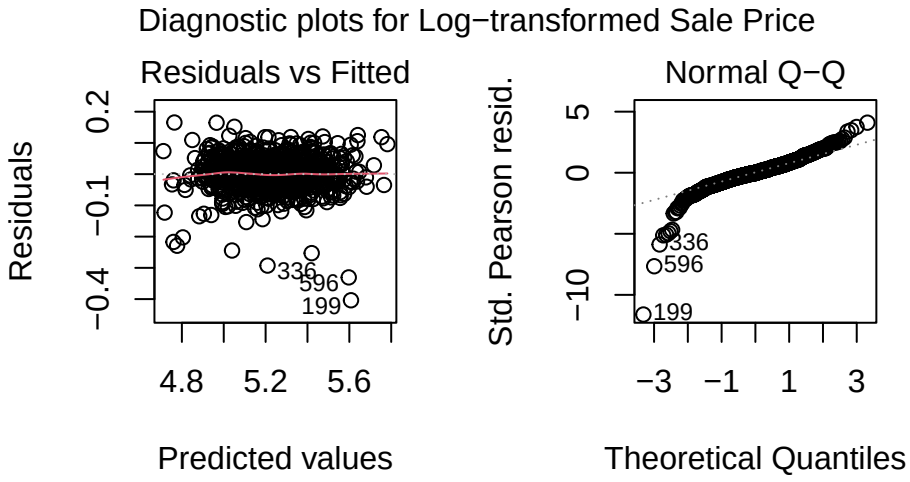
```
##    MSSubClass      MSZoning         Alley      LotShape   LandContour
##            14             5             3             4             4
##     LotConfig     LandSlope  Neighborhood    Condition1    Condition2
##             5             3            24             7             2
##      BldgType    HouseStyle     RoofStyle      RoofMatl   Exterior1st
##             5             8             5             3            11
##   Exterior2nd    MasVnrType     ExterQual     ExterCond    Foundation
##            11             5             4             4             5
##      BsmtQual      BsmtCond  BsmtExposure  BsmtFinType1  BsmtFinType2
##             5             5             5             7             7
##       Heating     HeatingQC    CentralAir    Electrical   KitchenQual
##             3             5             2             4             4
##    Functional   FireplaceQu    GarageType  GarageFinish    GarageQual
##             6             6             6             4             5
##    GarageCond    PavedDrive         Fence   MiscFeature        MoSold
##             4             3             5             3            12
##      SaleType SaleCondition
##             4             6
```

Finally, we can investigate the distribution of the response variable Sale Price. The distribution is clearly right-skewed with outliers that have extremely high sale price values, so linear regression methods may suffer. Hence we consider a log transformation on the response variable to de-emphasize outliers and obtain a more normally-distributed response variable. Moreover, the residual plots confirm that doing a log transformation produces a residual plot with more constant variance and less skewness. This suggests that taking the log might improve linear model performance. However, we should also beware that the residuals still don't follow a normal distribution even after log transformation, as shown by the Normal QQ plot.


Distribution of Sale Price
Sale Price
200
150
100
50
0
0e+00
2e+05
4e+05
6e+05
SalePrice


Distribution of Log transformed Sale Price
Log(Sale Price)
100
50
0
4.5
5.0
5.5
log10(SalePrice)





LR with Subset selection

Reduce data

We compute the correlation matrix and use the findCorrelation function to drop variables that have over 0.5 pair-wise correlation. The way this function chooses which of the pair to drop is by computing its mean correlation value with all other variables, and dropping the one with a higher mean correlation value. We drop these variables in to create reduced train and test sets. The variables dropped are shown below.

```
## [1] "GrLivArea"    "OverallQual"  "X1stFlrSF"    "TotalBsmtSF"  "GarageCars"
## [6] "TotRmsAbvGrd" "YearBuilt"    "X2ndFlrSF"    "BsmtFinSF1"
```

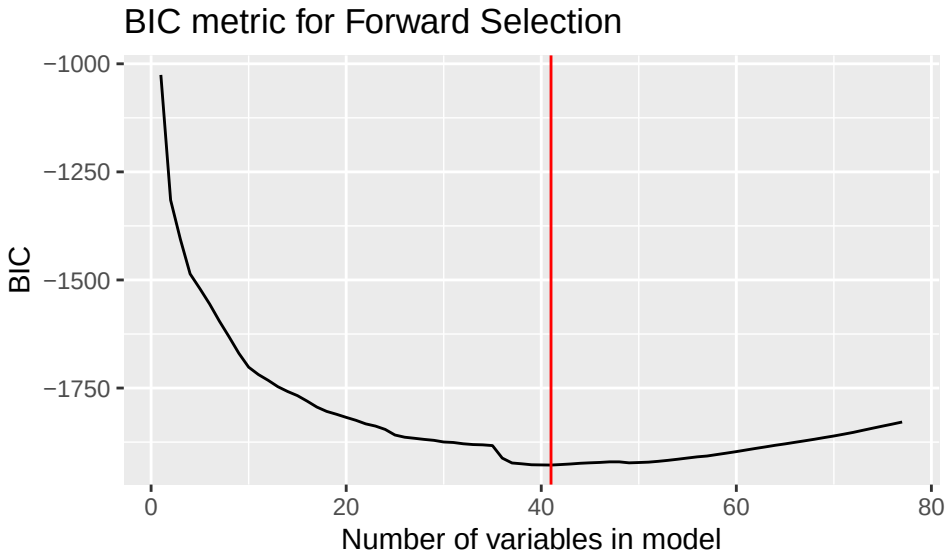
Forward Selection

We use forward selection on usual data, reduced data, log-transformed data, and reduced log-transformed data.

We choose the model with minimum BIC metric value, since that metric gives the simplest model. For example, the forward selection on the usual data gives the following models as the best according to different metrics.

```
##   adjr2.max rss.min cp.min bic.min
## 1        77      77     72      41
```

Here is a plot of the BIC metric as model complexity increases. We can see that the minimum occurs at 41



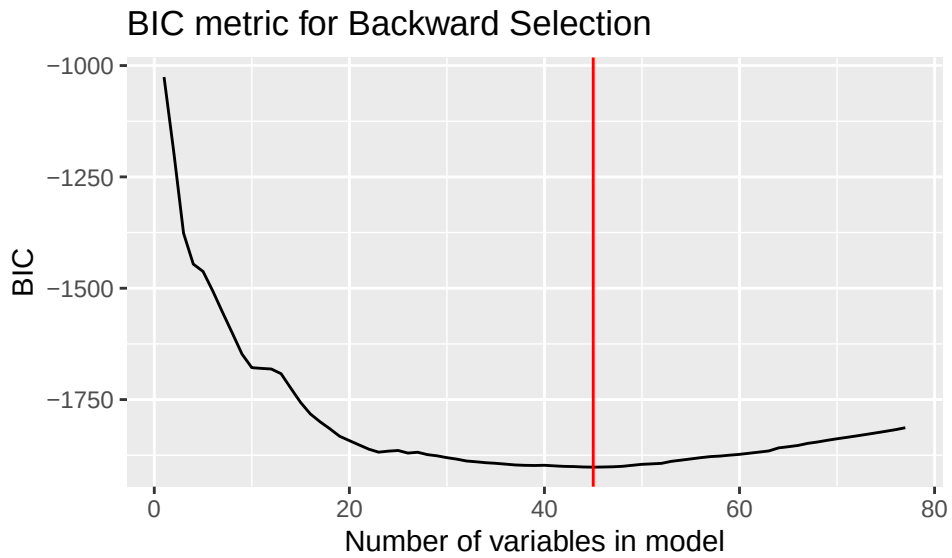
variables.

Backward Selection

Similarly, we use backward selection on four types of data: whether or not it is reduced, and whether or not it is log-transformed.

We continue to use bic.min as the metric for model selection, since that gives the simplest model.

```
##   b_adjr2.max b_rss.min b_cp.min b_bic.min
## 1          77        77       77        45
```



Predict

Next we make predictions and compute the test MSE values for each model.

We can see that the log transformation reduces test MSE value but reducing the dataset doesn't will instead increase test MSE. This confirms our conjecture in the exploratory state that log transformation will improve model performance. It also reveals that manually removing highly correlated variables don't actually improve model performance, and it is better to let the subset selection methods to their job.

Overall, backward selection seems to outperform forward selection. Hence according to test MSE, R^2, and adjusted R^2, backward selection on log-transformed data performs best. However, the BIC metric is minimized for backward selection on reduced data. Nevertheless, we decide to use backward selection on log-transformed data since the majority of the metrics agree on this algorithm.

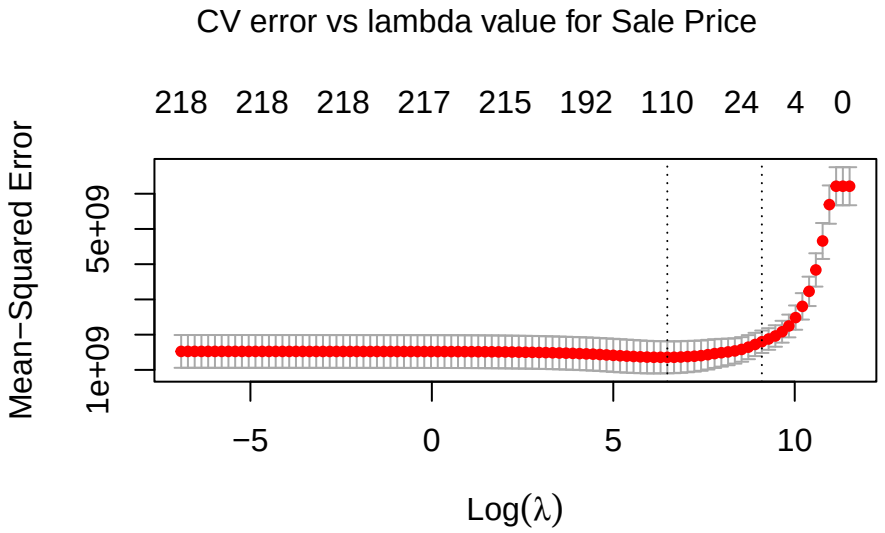
Algorithm	Test_MSE	Test_RMSE	BIC	R^2	Adjusted R^2
Forward	1.616e+09	40198	-1928	0.8686	0.8635
Forward Reduced	1.738e+09	41686	-1731	0.8427	0.8366
Forward Log	1.428e+09	37787	-2218	0.8972	0.8935
Forward Log Reduced	1.816e+09	42614	-1962	0.8758	0.8705
Backward	1.532e+09	39145	-1902	0.8688	0.8631
Backward Reduced	1.646e+09	40569	-1700	0.8392	0.8328
Backward Log	1.356e+09	36820	-2202	0.8989	0.8948
Backward Log Reduced	1.647e+09	40585	-1990	0.8765	0.8716

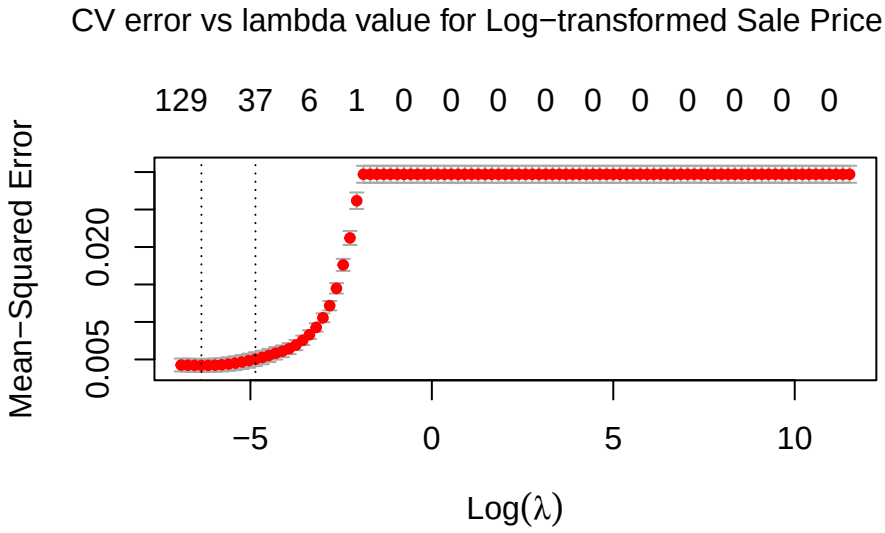
Here are the 43 variables included in the best model under backward selection on log-transformed model. We can see that many variables are related to measurements and qualities of various parts of the house, such as the bathroom, kitchen, and basement; as well as quality of amenities such as the garage and the fireplace. Interestingly, a typical or fair quality kitchen would also affect the house price negatively (`KitchenQualFa`, `KitchenQualTA`), although a good or excellent quality kitchen doesn't seem to improve house price significantly since these levels are not included in the model. House price is also impacted if the house is located in nine specific neighborhoods.

```
##          (Intercept)       MSSubClass160        MSSubClass20      LandContourHLS
##        3.002648e+00       -9.167436e-02        4.191079e-02        4.493870e-02
##      LandContourLow      LandContourLvl    LotConfigCulDSac NeighborhoodCrawfor
##        5.623819e-02        2.092294e-02        2.465208e-02        5.398163e-02
## NeighborhoodEdwards  NeighborhoodIDOTRR NeighborhoodMeadowV NeighborhoodNoRidge
##       -2.199180e-02       -8.943644e-02       -8.438757e-02        1.200563e-01
## NeighborhoodNridgHt NeighborhoodOldTown NeighborhoodSomerst NeighborhoodStoneBr
##        1.180869e-01       -1.917074e-02        6.642237e-02        1.471782e-01
##      Condition1Norm         OverallCond           YearBuilt  Exterior1stVinylSd
##        1.893365e-02        2.031170e-02        8.912774e-04       -2.725562e-03
##         ExterQualTA      FoundationSlab          BsmtQualTA      BsmtExposureNo
##       -2.407953e-02       -6.484061e-03       -1.809758e-02       -3.455209e-02
##     BsmtFinType2GLQ         CentralAirY           X2ndFlrSF         LowQualFinSF
##        5.499900e-02        3.158237e-02        2.367195e-05        1.102315e-04
##            FullBath            HalfBath       KitchenQualFa       KitchenQualTA
##        4.368798e-02        2.610351e-02       -4.779949e-02       -2.957632e-02
##        TotRmsAbvGrd      FunctionalMod       FunctionalTyp          Fireplaces
##        2.469710e-02       -2.007107e-02        1.072455e-02        4.359761e-02
##       FireplaceQuFa     GarageTypeNone        GarageQualTA         PavedDriveY
##       -9.963798e-03       -2.899871e-02        2.624620e-02        1.145108e-02
##           FenceNone            MoSold11    BsmtFinType1None      GarageQualNone
##        7.086545e-03       -3.812883e-03       -1.290368e-01        0.000000e+00
```

Lasso regression

We apply lasso regression on both usual and logged data.





For our optimal λ values, we take the λ values that minimizes CV error:

```
##      min_L   log_min_L
## 1 657.9332 0.001747528
```

We fit our LASSO and LASSO Log models with the optimal parameter:

We see that for LASSO, the log transformation also improves test MSE. We thus choose the logged model for LASSO.

Algorithm	Test_MSE	Test_RMSE
LASSO	755826419	27492
LASSO Log	657485003	25641

The Log LASSO model selects 220 levels out of 238 levels. We print out the first 15 levels selected below:

```
##                           coef
## NeighborhoodStoneBr 57174.115
## NeighborhoodNoRidge 56837.478
## NeighborhoodNridgHt 35936.339
## NeighborhoodVeenker 18753.465
## BsmtExposureGd      18167.109
## NeighborhoodCrawfor 17650.369
## NeighborhoodSomerst 16111.750
## LandContourHLS      13443.277
## SaleTypeNew         12991.132
## Exterior1stBrkFace  11802.124
## GarageCars          11787.280
## LotShapeIR2         11307.724
## OverallQual         10282.388
## LandContourLvl       9922.915
## RoofMatlother        8784.723
```

Since the coefficients of each level do not indicate its significance, we can not do much variable analysis here.

Tree-based methods

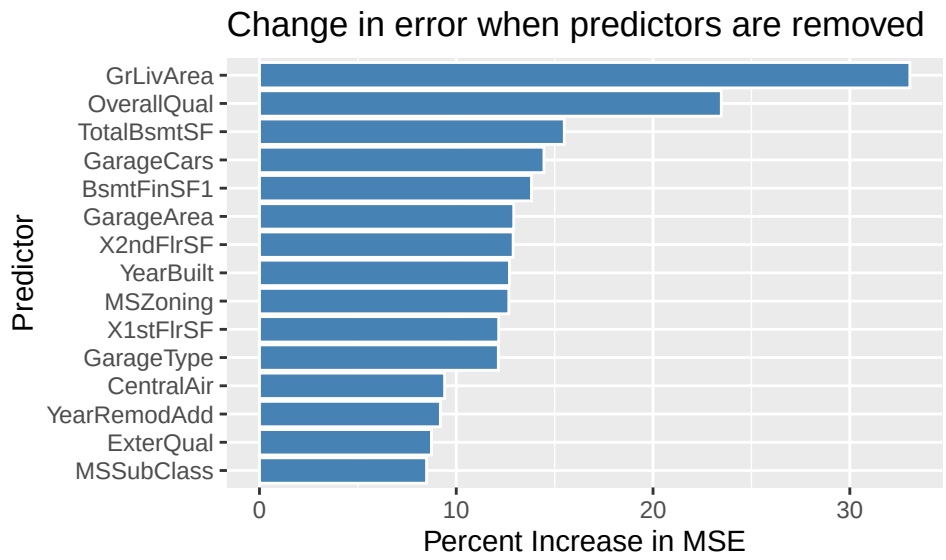
Training

We start with a single pruned decision tree. First we train a full tree. Next, we pick the maximum value of CP that has X-relative error within one standard deviation of the mean. This value balances performance and complexity. Then we prune our tree with the optimal CP value to make our final tree. With the pruned regression tree has been fit, we then fit bagged and random forests. First we fit the bagged tree by letting `mtry` be equal to the number of predictors. Then we fit the random forest. Here we set `mtry` equal to the default number for a random forest for regression ($p/3$). Then we extract predictions on test data and create a table of the test MSEs for each model to get a preliminary idea of the relative strengths of each model.

Algorithm	Test_MSE	Test_RMSE
Pruned Tree	1.436e+09	37893
Bagged Forest	640414104	25306
Bagged Forest Log	808708558	28438
Random Forest	614926331	24798
Random Forest Log	718890028	26812

Among trees, random forest performs the best and the pruned tree performs the worst. Models with log transforms also tend to perform worse on average than those without. The fact that the pruned tree performs worse than the other two trees makes sense since pruned trees are notorious for overfitting. The random forest also performs about 1% better than the bagged tree. Whether this increase in performance is significant is subject to debate, but we would expect a random forest to outperform a bagged tree on these data. This is because, returning to our EDA, many of the predictors in this dataset are highly correlated, which means that a bagged tree, which has access to all of the predictors at every step, might systematically choose non-optimal trees due to its greedy selection algorithm. In addition to predictions, the random forest also gives us access to variable importance data which we investigate now.

Variable Importance



Here the metric printed in the table is the percent increase in out of bag MSE when the values of a predictor are permuted. This metric provides a fairly robust estimate of the relative importance of predictors in our dataframe. Far and away the most important predictor is General Living Area. After that the next most important predictor is Overall Quality, a rating of the overall material and finish of the house on a scale one to ten. Year Built comes in 4th place, and many of the remaining important predictors have to do with the floor area and quality of various parts of the house. In general, it seems that the key factors in determining house price seem to be size, quality of material and finish, and the age of the home. Two other interesting important predictors are `Neighborhood` and `MSZoning`. Neighborhood is self explanatory, but `MSZoning` pertains to the general zoning classification of the sale (ie Commercial, Agricultural, Residential High Density, and so on). The importance of this variable seems to indicate that differently zoned transactions have different Sale Prices.

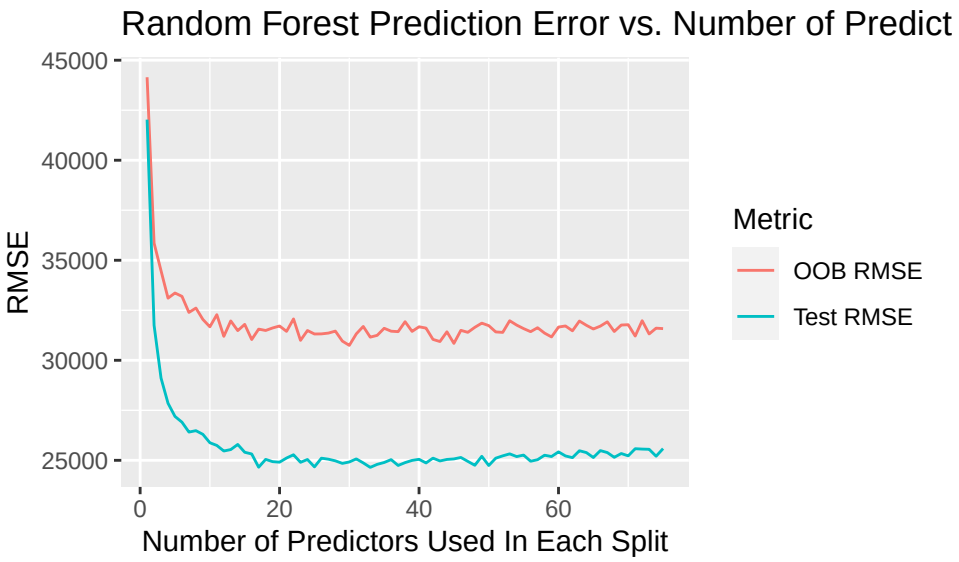
Further tuning

Now that we have a good image of the output of one random forest, we continue tuning to see if we can eek out more performance by tuning some of the hyperparameters that are used when fitting random forests. We tune the number of predictors used when splitting (`mtry`), the minimum node size for terminal nodes (`nodesize`), and the number of trees in the forest (`ntree`). In order to minimize computation time, we perform a guided and greedy search, first optimizing `mtry`, then optimizing `nodesize` for the optimal value of `mtry`, then optimizing `ntree` for using the optimal values of `mtry` and `nodesize`. Other less greedy approaches were attempted, but they took several hours to converge and ultimately led to very similar parameter values. Thus the greedy approach is presented in this report.

We first estimate test error and OOB error as a function the `mtry` parameter (the number of predictors used in splitting). The test MSE and OOB MSE minimizing values for `mtry` are printed below:

Error_Type	Minimal_mtry_value
Test MSE	33
OOB Error	30

To get a better picture of the relationship between the error and `mtry`, we plot the errors below:



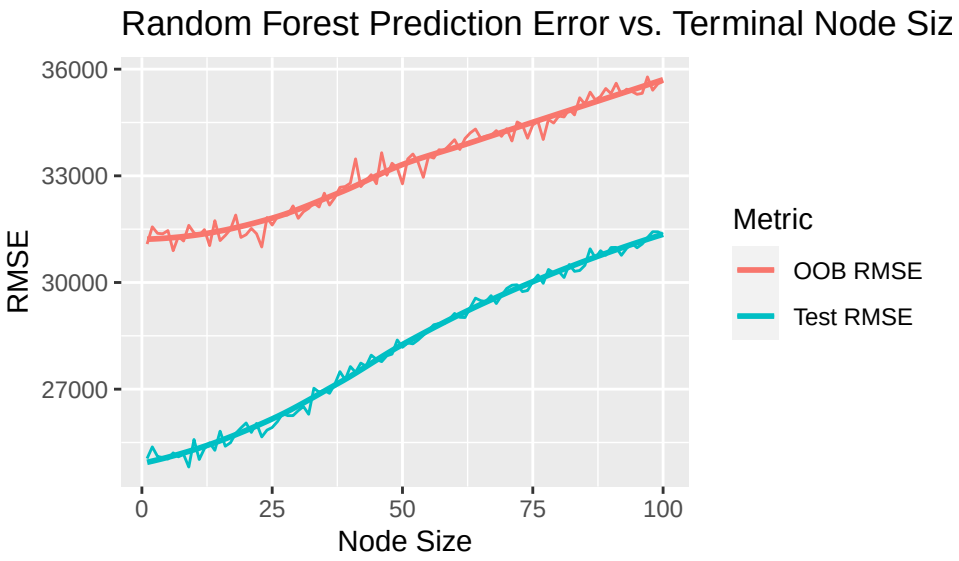
The OOB plot is more noisy, and achieves its minimum due to a sudden spike. This spike reaching the minimum looks more like noise than the general trend so we take the Test MSE minimizing value of 33 instead of the OOB minimizing value. Looking at the plots, this test MSE value also seems to be the minimum of the broader u-shaped trend.

Next, we use our optimal number for `mtry` and tune the minimum terminal node size for our trees. The test MSE and OOB MSE minimizing values for `nodesize` are printed below:

Error_Type	Minimal_nodesize_value
Test MSE	9
OOB Error	6

To get a better picture of the relationship between the error and `nodesize`, we plot the errors below:

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



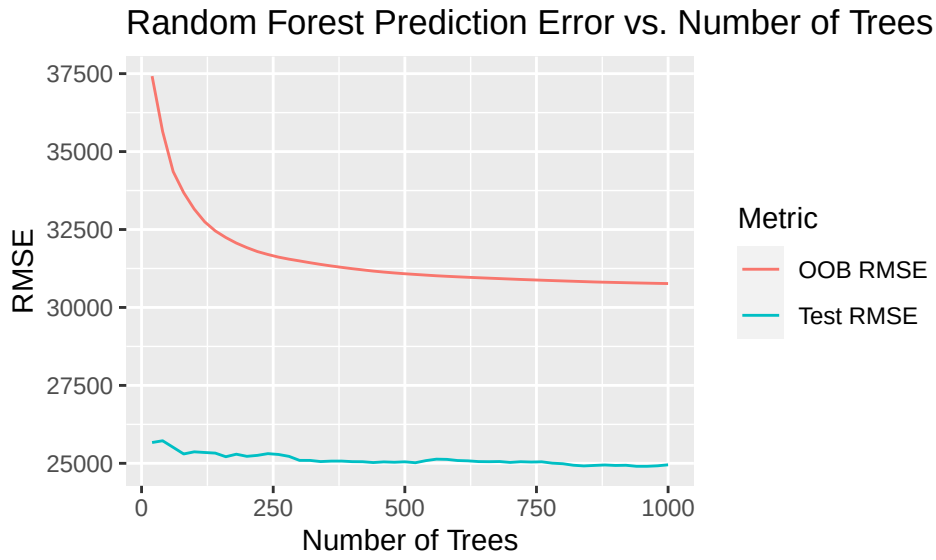
The OOB error minimizing value is 6 but the Test MSE minimizing value is nine. Based on the plot, the OOB error seems more volatile at the for small values of `nodesize`, but the test MSE achieves its minimum

due to a noisy spike that seems to be a deviation from the trend. We fit a `geom_smooth` to the data in order with `method = loess` to get a better idea of the general trends in the data. The curve made it easier to see that both errors were varying fairly evenly around an increasing trend. Hence we take the value one for our optimal parameter value. Lending credence to this choice is the fact that a value of 1 is the default value that `randomForest` uses for regression.

Finally we use all of our optimized parameters while fitting a forest and tune the number of trees. Really we are checking whether the `randomForest` default of 500 trees is enough for the test and OOB MSE to converge for this dataset. Again we print the test MSE and OOB MSE minimizing values for `nodesize` are in a table below:

Error_Type	Minimal_ntree_value
Test MSE	960
OOB Error	1000

Here we see that the Test MSE is actually minimized for an `ntree` value of 960. But with `ntree`, we are less interested in the precise number of trees which minimizes error, and more interested in the number of trees beyond which the values begin to level off. In order to find this value, we must look to the plot.



The OOB-Error seems to asymptotically decrease with the number of trees. The Test MSE is more variable at the beginning but then begins to level out around 750 trees. Following the conventions discussed ISLR, we want to pick a number of trees threshold past which the plot MSE seems to converge. There is a hard elbow in the OOB MSE plot around `ntree = 200`, after which the decrease in MSE seems to level off. In earlier tuning, we tested ntree values up to 2000 and found that doubling the number of trees from 1000 to 2000 decreased the OOB RMSE by about 0.59%. This showed that there are serious diminishing returns at play for more than 1000 trees. Fitting more than 1000 trees led to computaion times exceeding two hours and RAM issues, so the very slight increase in performance was not worth it for the authors. Since ISLR says that overfitting is not a concern with the number of trees, and there is a slight decrease in test MSE around 800 trees we take the test MSE minimizing value of 960 as our final number of trees.

Now we can fit the final random forest, with `mtry = 34`, `nodesize = 1`, and `ntree = 960`. We can compare the performance this final random forest to the untuned forest using the table below:

Metric	Untuned	Tuned	Percent_Change
OOB RMSE	31010	30761	-0.803
Test RMSE	24798	24759	-0.1556

We see that all of that time tuning resulted in around a one point seven percent decrease in OOB RMSE and an INCREASE in Test RMSE of 0.9%. This marginal increase in test MSE is not as alarming as it seems. Due to computational concerns, the test MSE estimate is not cross-validated. This means that the test MSE estimate is highly vulnerable to the observations included in the test set. The OOB estimate for test RMSE averages test MSE over all observations in the training set. Hence, a percent decrease in OOB error two times larger than the percent increase in test error indicates that our model likely will perform better when we train it on the full dataset to make predictions.

Conclusion

Algorithm	Test_RMSE	Test_MSE
Tuned Random Forest	24759	6.13e+08
LASSO Log	25641	657485003
Backward Log	36820	1.356e+09

Now we compare all the models that perform the best in each model type. From the table we see that the Tuned Random Forest model outperforms all other models with RMSE 24759, we thus select this model to perform prediction on the test set. Noted that the LASSO Log model has only a 3.4% higher RMSE than the Random Forest. Given a large number of predictors, it's surprising how well the LASSO Log model performs against the random forest model. If the problem is inference instead of prediction, LASSO Log is preferred to perform variable selection.

To answer our research question on which features are the most relevant in predicting house sale price, we compare the variables/levels selected by all three models, where the backward selection on log-transformed model selects 43 variables, the lasso regression on log-transformed model selects 220 out of 238 levels, and the tuned random forest model sets `mtry = 34` and provides an estimate of the relative importance of predictors. We use the variable importance estimate given by the tree model as a reference and check whether the most important variables in the chart are also selected by the backward selection and the lasso model. The three models share similar variable selection sets. In general, the key features in determining house price are the size and quality of each area and amenities, the age of the house, and the type of neighborhood.

References

Montoya, Anna. 2016. "House Prices - Advanced Regression Techniques." Kaggle. Retrieved December 5, 2022 (https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques/). # Code Appendix

```
knitr::opts_chunk$set(echo = FALSE, warning = FALSE)
library(tidyverse)
library(GGally)
library(skimr)
library(leaps) # regsubsets: subset selection
library(caret)
library(yardstick)
library(rpart) # trees
library(randomForest)
```

```
library(gglasso)
library(glmnet)
train_raw <- read_csv("train.csv", show_col_types = FALSE)
#Recode NA's as character values
train_df <- train_raw %>%
  select(-Id) %>%
  mutate(Alley = ifelse(is.na(Alley), "None", Alley),
         FireplaceQu = ifelse(is.na(FireplaceQu), "None", FireplaceQu),
         PoolQC = ifelse(is.na(PoolQC),  "None", PoolQC),
         Fence = ifelse(is.na(Fence), "None", Fence),
         MiscFeature = ifelse(is.na(MiscFeature), "None", MiscFeature),
         BsmtQual = ifelse(is.na(BsmtQual), "None", BsmtQual),
         BsmtCond = ifelse(is.na(BsmtCond), "None", BsmtCond),
         BsmtExposure = ifelse(is.na(BsmtExposure), "None", BsmtExposure),
         BsmtFinType1 = ifelse(is.na(BsmtFinType1), "None", BsmtFinType1),
         BsmtFinType2 = ifelse(is.na(BsmtFinType2), "None", BsmtFinType2),
         GarageType = ifelse(is.na(GarageType), "None", GarageType),
         GarageFinish = ifelse(is.na(GarageFinish), "None", GarageFinish),
         GarageQual = ifelse(is.na(GarageQual), "None", GarageQual),
         GarageCond = ifelse(is.na(GarageCond), "None", GarageCond),
         # YrSold = as.character(YrSold),
          MoSold = as.character(MoSold),
         # YearBuilt = as.character(YearBuilt),
         # YearRemodAdd = as.character(YearRemodAdd)
         MSSubClass = as.character(MSSubClass)
) %>%
  rename("X1stFlrSF"= `1stFlrSF`) %>%
  rename("X2ndFlrSF"= `2ndFlrSF`) %>%
  rename("X3SsnPorch"= `3SsnPorch`)

# 0.8 correlation beweetn Garage Year built and House year build
# train_df %>% select(GarageYrBlt, YearBuilt) %>% drop_na() %>%  cor()

#Change NAs in LotFrontage to zeroes
train_df<-train_df %>%
  mutate(LotFrontage = ifelse(is.na(LotFrontage), 0,LotFrontage)
) %>%
  select(-GarageYrBlt)
Mode <- function(x) {
ux <- unique(x)
ux[which.max(tabulate(match(x, ux)))]
}

otherify<-function(df, k, mode_rep){
  names<-df %>%
    select_if(negate(is.numeric)) %>%
    names()
  # d: each dummy counts number of obs in each level
  # names-vecs: each entry is a list of the levels with count <=k
  d<-list()
  names_vecs<-list()
  for (i in 1:length(names)){
    dummy<-df %>%
```

```
      group_by(.[[paste(names[i])]]) %>%
      summarise(count=n())
  names(dummy)=c(names[i],"count")

  d[[i]]<-dummy

  names_vecs[[i]]<-dummy %>%
    filter(count<=k) %>%
    select(-count) %>%
    pull()
  }

  for (i in 1:length(names)){
    df[[names[i]]]=ifelse(df[[names[i]]]%in%names_vecs[[i]],"other",df[[names[i]]])
  }

  if (mode_rep==T){
    for (i in 1:length(names)){
      s<-sum(df[,names[i]]=="other")
      if ((s!=0)&(s<k)){
        df[[names[i]]]=ifelse(df[[names[i]]]=="other",
                        Mode(df[[names[i]]]),
                        df[[names[i]]])
      }
    }
  }

  return(df)
}

modeify<-function(df, k){
  names<-df %>%
    select_if(negate(is.numeric)) %>%
    names()
  for (i in 1:length(names)){
      s<-sum(df[,names[i]]=="other")
      if ((s!=0)&(s<=k)){
        df[[names[i]]]=ifelse(df[[names[i]]]=="other",
                        Mode(df[[names[i]]]),
                        df[[names[i]]])
      }
  }
  return(df)
}

train_df<-train_df %>%
  otherify(k=10, mode_rep = F)

sample_size = floor(0.75*nrow(train_df))
set.seed(1)
picked = sample(seq_len(nrow(train_df)),size = sample_size)
train =train_df[picked,]
test =train_df[-picked,]
```

```r
#Impute the mode value in train and test for those "other" categories with
# less than 10 overall predictors
names<-train_df %>%
    select_if(negate(is.numeric)) %>%
    names()


for (i in 1:length(names)){
    s<-sum(train_df[[names[i]]]=="other")
    if ((s!=0)&(s<=10)){
      train[[names[i]]]=ifelse(train[[names[i]]]=="other",
                      Mode(train[[names[i]]]),
                      train[[names[i]]])
      test[[names[i]]]=ifelse(test[[names[i]]]=="other",
                      Mode(test[[names[i]]]),
                      test[[names[i]]])
    }
  }

#remove singleton categoricals
singleton<-character()
t=1
# s = number of others in train_df
# l = number of unique values in train_df
for (i in 1:length(names)){
  s<-sum(train_df[[names[i]]]=="other")
  l<-length(train_df[[names[i]]] %>% unique())
  if (((s!=0)&(s<=10)&(l==2))|| (l==1)  ){
    singleton[t]=names[i]
    t=t+1
  }
}

train_df<-train_df %>%
  select(-all_of(singleton))
train<-train %>%
  select(-all_of(singleton))
test<-test %>%
  select(-all_of(singleton))


#Impute missing values for MasVnr and Electrical and perform rough fix for GarageYrBlt
train<-train %>%
  mutate(MasVnrType=ifelse(is.na(MasVnrType), "None",MasVnrType)) %>%
  mutate(MasVnrArea=ifelse(is.na(MasVnrArea), 0,MasVnrArea)) %>%
  mutate(Electrical=ifelse(is.na(Electrical),"SBrkr",Electrical))

#Again for the test set, where we checked that the mode is the same
test<-test %>%
  mutate(MasVnrType=ifelse(is.na(MasVnrType), "None",MasVnrType)) %>%
  mutate(MasVnrArea=ifelse(is.na(MasVnrArea), 0,MasVnrArea)) %>%
  mutate(Electrical=ifelse(is.na(Electrical),"SBrkr",Electrical))
```

```r
build_cor_mat <- function(df){
  cor_mat<-df %>%
  select_if(is.numeric) %>%
  drop_na() %>%
  cor()

  n<-nrow(cor_mat)
  names<-row.names(cor_mat)
  correlation_tidy<-data.frame(row_name=character(),col_name=character(),corr=character())
  for(i in 1:n){

    for (l in i:n){
      corr<-cor_mat[i,l]
      row_name<-names[i]
      col_name<-names[l]

      df=data.frame(row_name,col_name,corr)
      correlation_tidy=rbind(correlation_tidy,df)
    }
  }
  return(list(cor_mat, correlation_tidy))
}

return_list <- build_cor_mat(train_df)
cor_mat <- return_list[[1]]
correlation_tidy <- return_list[[2]]
# Check what has highest (absolute) correlation with sale price
correlation_tidy %>%
  filter(col_name=="SalePrice") %>%
  mutate(abs_corr=abs(corr)) %>%
  arrange(desc(abs_corr))%>%
  select(-abs_corr) %>%
  filter(row_name != col_name) %>%
  head(6)
# Check which has negative correlation
# What Overall Cond has negative correlation?
correlation_tidy %>%
  filter(col_name=="SalePrice",
         corr<=0) %>%
  mutate(abs_corr=abs(corr)) %>%
  arrange(desc(abs_corr))%>%
  select(-abs_corr) %>%
  head()

# SalePrice vs OverallCond
ggplot(train, aes(y = SalePrice, x = OverallCond)) +
  geom_point() +
  labs(title = "Sale Price vs Overall Condition",
       y = "Overall Condition",
       x = "Sale Price")
# Correlation between predictors
correlation_tidy %>%
  filter(col_name!="SalePrice",
```

```
         col_name!=row_name) %>%
  mutate(abs_corr=abs(corr)) %>%
  arrange(desc(abs_corr))%>%
  select(-abs_corr) %>%
  head()
sapply(lapply(train_df %>% select_if(negate(is.numeric)), unique), length)
# SalePrice distribution is right skewed
ggplot(train, aes(x = SalePrice)) +
  geom_histogram(color = "white", bins=30)+
  labs(title = "Distribution of Sale Price",
       y = "Sale Price")
# SalePrice distribution is right skewed
ggplot(train, aes(x = log10(SalePrice))) +
  geom_histogram(color = "white", bins=30) +
  labs(title = "Distribution of Log transformed Sale Price",
       y = "Log(Sale Price)")
# Taking log solves residual pattern
full_mod <- glm(SalePrice ~ ., data = train)
par(mfrow = c(1,2))
plot(full_mod, which = c(1,2))
mtext("Diagnostic plots for Sale Price",
      side = 3,
      line = -2.5,
      outer = TRUE)
par(mfrow = c(1,2))
log_mod <- glm(log10(SalePrice) ~ ., data = train)
plot(log_mod, which = c(1,2))
mtext("Diagnostic plots for Log-transformed Sale Price",
      side = 3,
      line = -2.5,
      outer = TRUE)

# correlation_tidy %>%
#   filter(row_name != col_name) %>%
#   filter(col_name != "SalePrice") %>%
#   filter(abs(corr) > 0.5) %>%
#   arrange(desc(abs(corr)))

# remove correlations with response variable
new_cor_mat <-cor_mat[rownames(cor_mat) != "SalePrice", colnames(cor_mat) != "SalePrice"]

# Finds pair-wise correlation over 0.5
# removes the one with higher mean correlation with other variables
rm_names <- findCorrelation(new_cor_mat, 0.5, names=T)
rm_names
rm_idx <- which(names(train) %in% rm_names)

# reduce train and test data
red_train = train[, -rm_idx]
red_test = test[, -rm_idx]

# # Check that reduced train_df has no variables with corr > 0.5
# red_return_list <- build_cor_mat(red_train_df)
```

```r
# red_cor_mat <- red_return_list[[1]]
# red_correlation_tidy <- red_return_list[[2]]
#
# red_correlation_tidy %>%
#   filter(row_name != col_name) %>%
#   filter(col_name != "SalePrice") %>%
#   arrange(desc(abs(corr)))

# # Check that using train will result in removing the same variables
# tmp <- train %>% select_if(is.numeric) %>% cor()
# tmp_cor_mat <-cor_mat[rownames(tmp) != "SalePrice", colnames(tmp) != "SalePrice"]
# tmp_names <- findCorrelation(tmp_cor_mat, 0.5, names=T, verbose=T)
# sort(tmp_names) == sort(rm_names)
## R code: Model testing
n = dim(train)[2]
forward_mod <- regsubsets(SalePrice ~ ., data=train, nvmax=n, method="forward")
forward_mod_red <- regsubsets(SalePrice ~ ., data=red_train, nvmax=n, method="forward")
forward_mod_log <- regsubsets(log10(SalePrice) ~ ., data=train, nvmax=n, method="forward")
forward_mod_log_red <- regsubsets(log10(SalePrice) ~ ., data=red_train, nvmax=n, method="forward")
# Absolute metrics give full models as best, except bic.min which gives 64-model as best
adjr2.max <- which.max(summary(forward_mod)$adjr2)
rss.min <- which.min(summary(forward_mod)$rss)
cp.min <- which.min(summary(forward_mod)$cp)
bic.min <- which.min(summary(forward_mod)$bic)
data.frame(adjr2.max, rss.min, cp.min, bic.min)
d <- data.frame(model = 1:(dim(train)[2]+1),
adjr2 = summary(forward_mod)$adjr2,
rss = summary(forward_mod)$rss,
cp = summary(forward_mod)$cp,
bic = summary(forward_mod)$bic)

ggplot(d, aes(x = model, y = bic)) +
  geom_line() +
  labs(title = "BIC metric for Forward Selection",
       x = "Number of variables in model",
       y = "BIC") +
  geom_vline(aes(xintercept = bic.min),color = "red")
backward_mod <- regsubsets(SalePrice ~ ., data=train, nvmax=n, method="backward")
backward_mod_red <- regsubsets(SalePrice ~ ., data=red_train, nvmax=n, method="backward")
backward_mod_log <- regsubsets(log10(SalePrice) ~ ., data=train, nvmax=n, method="backward")
backward_mod_log_red <- regsubsets(log10(SalePrice) ~ ., data=red_train, nvmax=n, method="backward")
b_adjr2.max <- which.max(summary(backward_mod)$adjr2)
b_rss.min <- which.min(summary(backward_mod)$rss)
b_cp.min <- which.min(summary(backward_mod)$cp)
b_bic.min <- which.min(summary(backward_mod)$bic)
data.frame(b_adjr2.max, b_rss.min, b_cp.min, b_bic.min)
b_d <- data.frame(model = 1:(dim(train)[2]+1),
adjr2 = summary(backward_mod)$adjr2,
rss = summary(backward_mod)$rss,
cp = summary(backward_mod)$cp,
bic = summary(backward_mod)$bic)

ggplot(b_d, aes(x = model, y = bic)) +
```

```
  geom_line() +
  labs(title = "BIC metric for Backward Selection",
       x = "Number of variables in model",
       y = "BIC") +
  geom_vline(aes(xintercept = b_bic.min),color = "red")
predict.regsubsets = function(object, newdata, id, ...) {
    form = as.formula(object$call[[2]])
    mat = model.matrix(form, newdata)
    coefi = coef(object, id = id)
    mat[, names(coefi)] %*% coefi
}

lr_results <- function(model, reduce=F, log=F){
  id = which.min(summary(model)$bic)
  preds <- predict.regsubsets(model, test, id= id)
  if(log){
    preds <- 10^preds
  }
  if(reduce){
    mse <- mean((red_test$SalePrice - preds)^2)
  } else{
    mse <- mean((test$SalePrice - preds)^2)
  }
  bic <- summary(model)$bic[id]
  r2 <- summary(model)$rsq[id]
  adjr2 <- summary(model)$adjr2[id]
  df <- data.frame(mse, bic, r2, adjr2)
  return(df)
}

lr_out <- rbind(
  lr_results(forward_mod),
  lr_results(forward_mod_red, reduce = T),
  lr_results(forward_mod_log, log=T),
  lr_results(forward_mod_log_red, reduce = T, log=T),
  lr_results(backward_mod),
  lr_results(backward_mod_red, reduce = T),
  lr_results(backward_mod_log, log=T),
  lr_results(backward_mod_log_red, reduce = T, log=T)
) %>% mutate(
  Algorithm = c("Forward", "Forward Reduced", "Forward Log", "Forward Log Reduced",
                "Backward", "Backward Reduced", "Backward Log", "Backward Log Reduced")
) %>%
   rename(
     Test_MSE = mse,
     BIC = bic,
     R_squared = r2,
    Adjusted_R_squared = adjr2) %>%
   mutate(Test_RMSE = sqrt(Test_MSE)) %>%
   select(Algorithm, Test_MSE, Test_RMSE, BIC, R_squared, Adjusted_R_squared)
names(lr_out)[5] = "$R^2$"
names(lr_out)[6] = "Adjusted $R^2$"
lr_out %>%
```

```
  pander::pander()
coef_back <- coef(backward_mod_log, id = which.min(summary(backward_mod_log)$bic))
coef_back
set.seed(1)

x<-model.matrix(SalePrice ~., data = train)[,-1]
y<-train$SalePrice

logx<-model.matrix(log10(SalePrice) ~., data = train)[,-1]
logy<-log10(train$SalePrice)

grid = 10^(seq( -3, 5, length = 100))
lasso_cv<-cv.glmnet(x,y,alpha=1,lambda=grid,nfolds=10)
log_lasso_cv<-cv.glmnet(logx,logy,alpha=1,lambda=grid,nfolds=10)
plot(lasso_cv)
mtext("CV error vs lambda value for Sale Price",
      side = 3,
      line = -1,
      outer = TRUE)
plot(log_lasso_cv)
mtext("CV error vs lambda value for Log-transformed Sale Price",
      side = 3,
      line = -1,
      outer = TRUE)
min_L= lasso_cv$lambda.min
log_min_L = log_lasso_cv$lambda.min
data.frame(min_L, log_min_L)
lasso_mod<-glmnet(x,y,alpha=1,lambda=grid,nfolds=10)
test_mat<-model.matrix(SalePrice ~., data = test)[,-1]
lasso_pred<-predict(lasso_mod,s=min_L, newx=test_mat)

log_lasso_mod<-glmnet(logx,logy,alpha=1,lambda=grid,nfolds=10)
log_test_mat<-model.matrix(log10(SalePrice) ~., data = test)[,-1]
log_lasso_pred<-predict(log_lasso_mod,s=log_min_L, newx=log_test_mat)
test_TV<-test$SalePrice
lasso_MSE<-mean((test_TV-lasso_pred)^2)
log_lasso_MSE<-mean((test_TV-10^(log_lasso_pred))^2)

lasso_out <- data.frame(Algorithm=c("LASSO", "LASSO Log"),Test_MSE=c(lasso_MSE, log_lasso_MSE)) %>%
  mutate(Test_RMSE=sqrt(Test_MSE))%>%
  arrange(desc(Test_MSE))

lasso_out %>% pander::pander()
level <- data.frame(matrix(ncol = 76, nrow = 1))
for (i in 1:76){
  if(is.na(as.numeric(train[[i]])[[1]])){
    level[[i]] = as.factor(train[[i]]) %>%
    nlevels()
  }
  if(!is.na(as.numeric(train[[i]])[[1]])){
    level[[i]] = 1
  }
}
```

```
total_level <- rowSums(level)

s <- which(lasso_mod$lambda==min_L)
coef <- data.frame(coef = coef(lasso_mod)[,s]) %>%
  arrange(desc(coef))
coef_print <- coef %>%
  head(15)
coef_print
# First we train a full tree.
set.seed(1)
T0<-rpart(SalePrice~., data=train,
             control = rpart.control(cp=0))

# Next, we pick the maximum value of CP that has
# $X$-relative error within one standard deviation of the mean.
# This value balances performance and complexity
cptable<-T0$cptable %>% as.data.frame()


p.rpart <- T0$cptable
xstd <- p.rpart[, 5L]
    xerror <- p.rpart[, 4L]
minpos <- min(seq_along(xerror)[xerror == min(xerror)])
thresh<-(xerror + xstd)[minpos]


best_cp<-cptable %>%
  filter(xerror<=thresh) %>%
  filter(CP==max(CP)) %>%
  select(CP) %>%
  pull()


pruned<-prune(T0,best_cp)
#First we fit the bagged tree, letting `mtry` be equal to the number of predictors
set.seed(1)

bag<-randomForest(SalePrice~.,
                        mtry=ncol(train)-1,
                        data=train,
                        importance=T)

set.seed(1)

log_bag<-randomForest(log10(SalePrice)~.,
                        mtry=ncol(train)-1,
                        data=train,
                        importance=T)
#Now we fit the random forest. Here we set `mtry` equal to the
# default number for a random forest for regression ($p/3$).
set.seed(1)
rf<-randomForest(SalePrice~.,
                        mtry=(ncol(train)-1)/3,
```

```
                        data=train,
                        importance=T)

set.seed(1)
log_rf<-randomForest(log10(SalePrice)~.,
                        mtry=(ncol(train)-1)/3,
                        data=train,
                        importance=T)
## Now we extract predictions on test data and create a table of MSEs for each model.

#Tree preds
test_Tree_pred<-predict(pruned,newdata = test
                          )

#RF
test_rf_preds<-predict(rf,test)

#Bagged
test_bag_preds<-predict(bag,test)

#RF
log_rf_preds<-10^predict(log_rf,test)

#Bagged
log_bag_preds<-10^predict(log_bag,test)

#True values
test_TV<-test$SalePrice
# Get MSEs for Trees
tree_MSE<-mean((test_TV-test_Tree_pred)^2)
rf_MSE<-mean((test_TV-test_rf_preds)^2)
bag_MSE<-mean((test_TV-test_bag_preds)^2)
log_rf_MSE<-mean((test_TV-log_rf_preds)^2)
log_bag_MSE<-mean((test_TV-log_bag_preds)^2)

data.frame(Algorithm=c("Pruned Tree","Bagged Forest","Bagged Forest Log",
                        "Random Forest","Random Forest Log"),
           Test_MSE=c(tree_MSE,bag_MSE,log_bag_MSE,rf_MSE,log_rf_MSE)) %>%

mutate(Test_RMSE=sqrt(Test_MSE))%>%
  pander::pander()
imp<-importance(rf) %>%
  as.data.frame() %>%
  rownames_to_column() %>%
  rename("Predictor"=rowname) %>%
  arrange(desc(`%IncMSE`))

imp[1:15,] %>%
  mutate(Predictor = fct_reorder(Predictor, `%IncMSE`)) %>%
  ggplot( mapping=aes(x=Predictor,y=`%IncMSE`))+
  geom_col(color="white",fill="steelblue")+
  coord_flip()+
  labs(title="Change in error when predictors are removed",y="Percent Increase in MSE")
```

```
m_try_tune<-readRDS("m_try_tune.rds")

#This Allows us to compute the tuning grid directly (Takes about 10 minutes)


# library(tictoc)
# tic()
# m_try_tune<-list()
#
# for (i in 1:(ncol(train)-1)){
#   set.seed(1)
#   m_try_tune[[i]]<-randomForest(SalePrice~.,
#                         mtry=i,
#                         data=train,
#                         importance=T)
# }
# saveRDS(m_try_tune,file="m_try_tune.rds")
# toc()
mtry_oob<-double(length=ncol(train)-1)
mtry_tMSE<-double(length=ncol(train)-1)
mtry<-1:(ncol(train)-1)

for (i in 1:(ncol(train)-1)){
  mtry_oob[i]=mean(m_try_tune[[i]]$mse)

  mtry_tMSE[i]<-mean((test_TV-predict(m_try_tune[[i]],test))^2)
}
mtry_df<-data.frame(mtry,mtry_tMSE,mtry_oob)

data.frame(`Error_Type`=c("Test MSE", "OOB Error"),
           `Minimal_mtry_value`=c(which.min(mtry_df$mtry_tMSE),
                                  which.min(mtry_df$mtry_oob))) %>%
  pander::pander()
ggplot(mtry_df %>%
  pivot_longer(2:3,names_to = "Metric",values_to = "Error") %>%
    mutate(Metric=case_when(Metric=="mtry_oob"~"OOB RMSE",
                            Metric=="mtry_tMSE"~"Test RMSE")),
  aes(x=mtry,y=sqrt(Error),color=Metric))+
  geom_line()+
  labs(y="RMSE",
       x="Number of Predictors Used In Each Split",
       title="Random Forest Prediction Error vs. Number of Predictors Used in Each Split")

node_size_tune<-readRDS("node_size_tune.rds")

#This Allows us to compute the tuning grid directly (Takes about 10 minutes)

 node_size<-1:100
#
#  tic()
#  node_size_tune<-list()
#
```

```
#  for (i in 1:length(node_size)){
#    set.seed(1)
#    node_size_tune[[i]]<-randomForest(SalePrice~.,
#                          mtry=33,
#                          data=train,
#                          importance=F,
#                          nodesize=node_size[i])
#  }
#  saveRDS(node_size_tune,file="node_size_tune.rds")
#  toc()
node_size_oob<-double(length=length(node_size))
node_size_tMSE<-double(length=length(node_size))


for (i in 1:length(node_size)){
  node_size_oob[i]=mean(node_size_tune[[i]]$mse)

  node_size_tMSE[i]<-mean((test_TV-predict(node_size_tune[[i]],test))^2)
}

node_size_df<-data.frame(node_size,node_size_tMSE,node_size_oob)
data.frame(`Error_Type`=c("Test MSE", "OOB Error"),
           `Minimal_nodesize_value`=c(which.min(node_size_df$node_size_tMSE),
                                      which.min(node_size_df$node_size_oob))) %>%
  pander::pander()
ggplot(node_size_df %>%
  pivot_longer(2:3,names_to = "Metric",values_to = "Error")%>%
    mutate(Metric=case_when(Metric=="node_size_oob"~"OOB RMSE",
                            Metric=="node_size_tMSE"~"Test RMSE")),
  aes(x=node_size,y=sqrt(Error),color=Metric))+
  geom_line()+
  geom_smooth(se=F)+
  labs(y="RMSE",x='Node Size',
       title="Random Forest Prediction Error vs. Terminal Node Size")

n_tree_tune<-readRDS("n_tree_tune.rds")

#This Allows us to compute the tuning grid directly (Takes about an hour)


 tree_num<-seq(20,1000,by=20)

# tic()
# n_tree_tune<-list()
#
# for (i in 1:length(tree_num)){
#   set.seed(1)
#   n_tree_tune[[i]]<-randomForest(SalePrice~.,
#                          mtry=33,
#                          data=train,
#                          nodesize=1,
#                          ntree=tree_num[i],
#                          importance=F)
```

```
# }
#  saveRDS(n_tree_tune,file="n_tree_tune.rds")
# toc()
ntree_oob<-double(length=length(tree_num))
ntree_tMSE<-double(length=length(tree_num))


for (i in 1:length(tree_num)){
  ntree_oob[i]=mean(n_tree_tune[[i]]$mse)

  ntree_tMSE[i]<-mean((test_TV-predict(n_tree_tune[[i]],test))^2)
}
ntree_df<-data.frame(tree_num,ntree_tMSE,ntree_oob)
data.frame(`Error_Type`=c("Test MSE", "OOB Error"),
           `Minimal_ntree_value`=c(which.min(ntree_df$ntree_tMSE)*20,
                                   which.min(ntree_df$ntree_oob)*20)) %>%
pander::pander()
ggplot(ntree_df %>%
  pivot_longer(2:3,names_to = "Metric",values_to = "Error")%>%
    mutate(Metric=case_when(Metric=="ntree_oob"~"OOB RMSE",
                            Metric=="ntree_tMSE"~"Test RMSE")),
  aes(x=tree_num,y=sqrt(Error),color=Metric))+
  geom_line()+
  labs(y="RMSE",x='Number of Trees',
       title="Random Forest Prediction Error vs. Number of Trees")
set.seed(1)
final_rf<-randomForest(SalePrice~.,
                       mtry=33,
                       data=train,
                       nodesize=1,
                       ntree=960,
                       importance=F)


rf_oob=mean(rf$mse)
rf_MSE<-mean((test_TV-test_rf_preds)^2)
final_rf_oob= mean(final_rf$mse)
final_rf_tMSE<-mean((test_TV-predict(final_rf,test))^2)

final_rf_sum<-data.frame(final_rf_oob,final_rf_tMSE) %>%
  pivot_longer(1:2,names_to = "Metric",values_to = "MSE") %>%
  mutate(Random_Forest="Tuned")

table<-data.frame(rf_oob,rf_MSE) %>%
  pivot_longer(1:2,names_to = "Metric",values_to = "MSE") %>%
  mutate(Random_Forest="Untuned")
rbind(final_rf_sum,table) %>%
  mutate(RMSE=sqrt(MSE)) %>%
  select(-MSE) %>%
  mutate(Metric=case_when(Metric=="final_rf_oob"~"OOB RMSE",
                          Metric=="rf_oob"~"OOB RMSE",
                          Metric=="final_rf_tMSE"~"Test RMSE",
                          Metric=="rf_MSE"~"Test RMSE")) %>%
```

```
  pivot_wider(names_from = "Random_Forest",values_from = "RMSE") %>%
  select(Metric,Untuned,Tuned) %>%
  mutate(Percent_Change=100*(Tuned-Untuned)/Untuned) %>%
  pander::pander()

#Here are the final test MSE's if someone wants to use them in a table
#Original RF
rf_oob=mean(rf$mse)
rf_MSE<-mean((test_TV-test_rf_preds)^2)

#Tuned RF
final_rf_oob= mean(final_rf$mse)
final_rf_MSE<-mean((test_TV-predict(final_rf,test))^2)

tree_out <- data.frame(Algorithm=c("Pruned Tree","Bagged Forest",
                                   "Random Forest", "Tuned Random Forest"),
                       Test_MSE=c(tree_MSE,bag_MSE,rf_MSE, final_rf_MSE)) %>%
mutate(Test_RMSE=sqrt(Test_MSE))
rbind(
lr_out %>% filter(Algorithm == "Backward Log") %>% select(Algorithm, Test_RMSE, Test_MSE),
lasso_out  %>% filter(Algorithm == "LASSO Log"),
tree_out  %>% filter(Algorithm == "Tuned Random Forest")) %>%
  arrange(Test_RMSE) %>% pander::pander()
```