STAT 4011 project 2

You Xinyu 1155110904

collaboration: Cheng Wing Ryan 1155102964

1. Regression

Import the House dataset

```
House = data.frame(read.csv("/Users/ericyau/Desktop/House.csv"))
```

required packages

```
library(tinytex)
library(mice)
library(caret)
library(MASS)
library(leaps)
library(randomForest)
library(ranger)
library(dplyr)
library(gbm)
library(vip)
```

1.1 Data Cleaning

The first step for starting analysis is always checking the data structure, which provide us a preliminary overview on the dataset. The top priority is to check whether there are any missing values in the dataset, if missing values exisits we have to either fill it in for delete the corresponding observation in the dataset. Inn this study only imputation methods will be implemented.

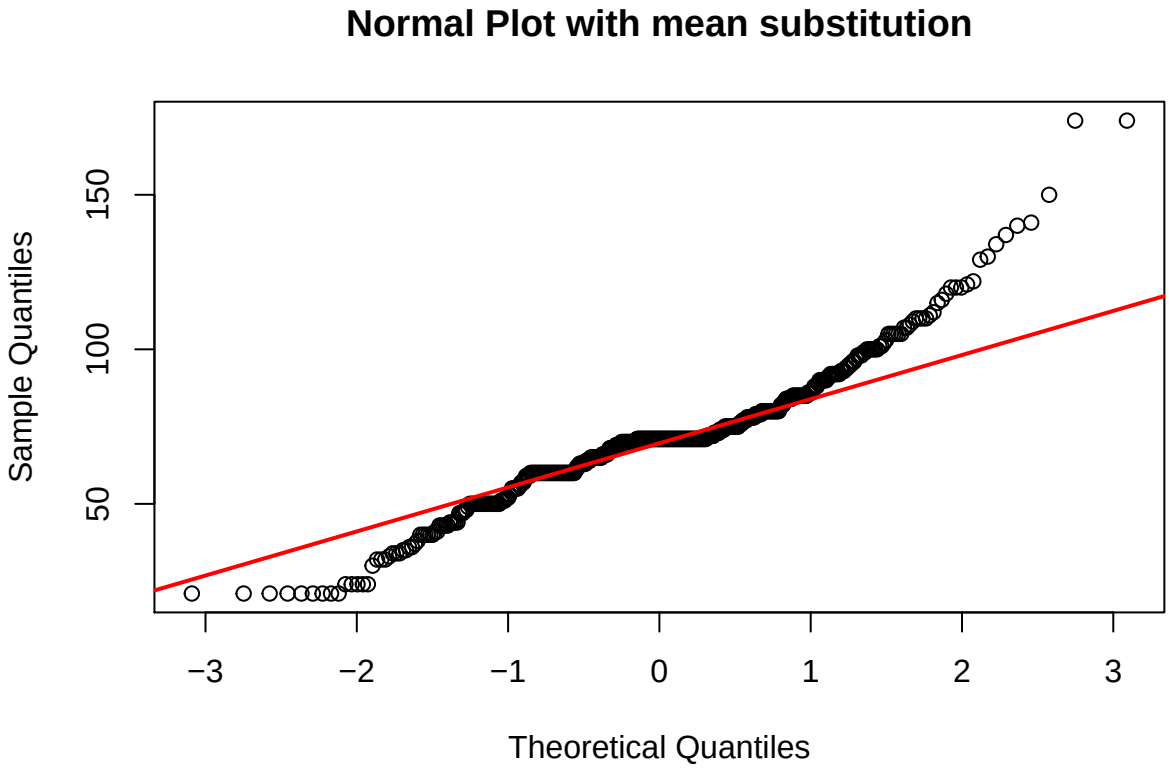
```
summary(House)
```

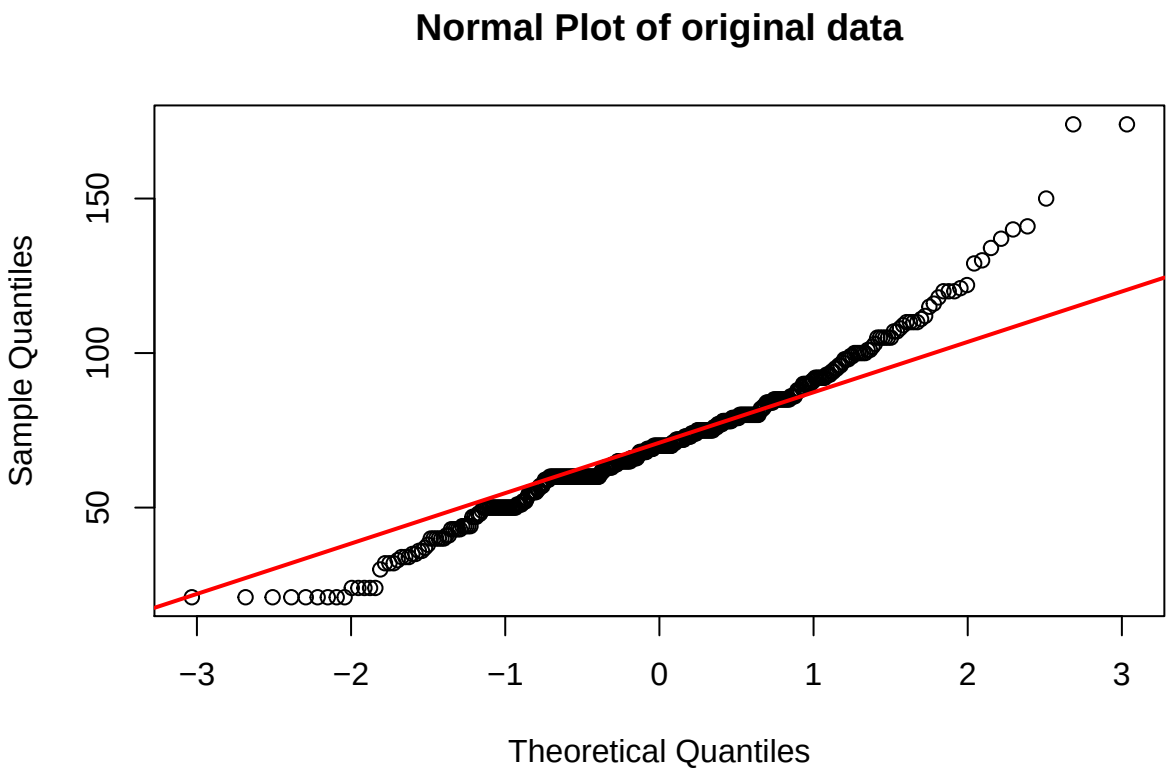
After viewing the summary of the "House" dataset, 87 missing values were found in variable "LotFrontage"(Linear feet of street connected to property) and 1 missing value in was discovered in variable "MasVnrArea"(Masonry veneer area in square feet). We first substitue the only missing value in "MasVnrArea" simply by substitue the missing value by the mean of the other 499 data points.

```
# mean substitution for MasVnrArea
House$MasVnrArea[is.na(House$MasVnrArea)] = round(mean(House$MasVnrArea,na.rm = T))
```

Next the place of the 87 missing values "NA" need to be found so as to do further imputations for the missing values.Seems the missing values were roughly missing at random. Then we are interested in the

consequence of applying mean substitution to 87 missing values, thus we first substitue the missing values with mean and look at the new distribution of the “LotFrontage”

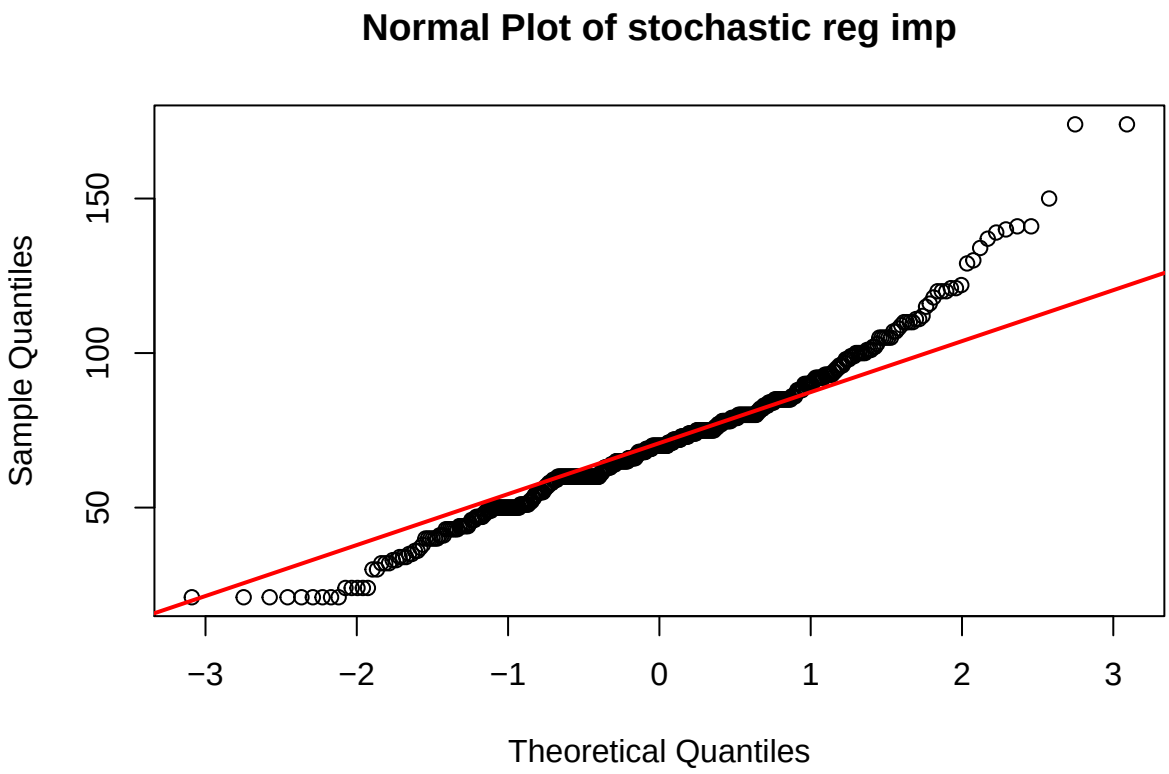


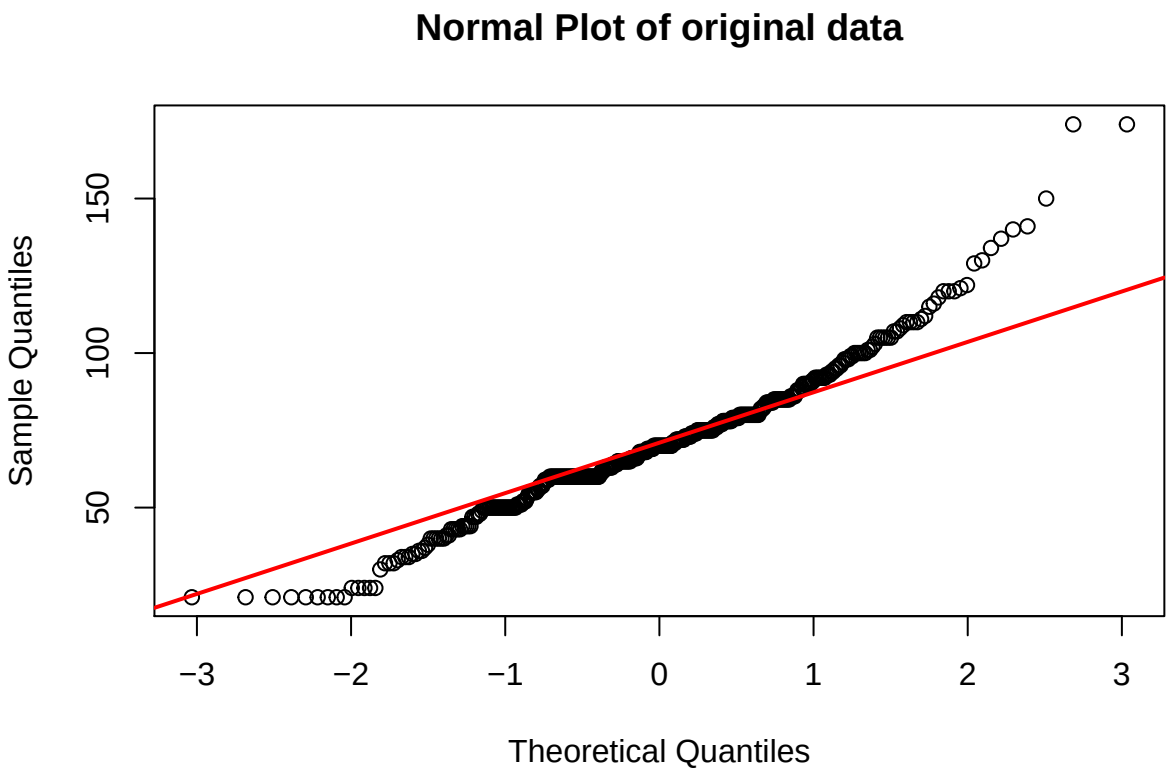


By comparing the distribution of the mean substituition with the original distribution, we found that the distribution is slightly flatter than the original distribution. Therefore 87 missing values in "LotFrontage" may not as simple as just substitue the mean of the other 413 data points. Therefore, we have to discover other imputation methods. In this case, linear regression will be applied to estimate the missing values.

```
#Stocahstic regression imputation
imputation = mice(House,method = "norm.nob",m=1)
House.NoNA = complete(imputation)
House.NoNA$LotFrontage = as.integer(House.NoNA$LotFrontage)
```

By attaching "mice" package for stocahstic regression imputation, a new dataset with filled missing values was generated, the plot of the new distribution as below:


Normal Plot of stochastic reg imp
Sample Quantiles
50
100
150
−3
−2
−1
0
1
2
3
Theoretical Quantiles



We can see that the distribution after stochastic regression is very close to the original normal distribution, so we would use the dataset imputed by stochastic regression in the following analysis methods.

```
House = House.NoNA
House$LotShape = as.factor(House$LotShape)
set.seed(4011)
n = sample(nrow(House),size = nrow(House)*0.8)
#drop the id column
train.H = House[n,-1]
test.H = House[-n,-1]
```

Split the dataset into 2 parts with train set contain 400 random observations and test set contain the remained 100 observations. Then change the type of "LotShape" into factor levels.

1.2 Linear Regression

Linear regression is a statistical technique using the least square method to minimize the residuals between a dependent variable and serveral dependent variables, also it uses several explanatory variables to predict the outcome of a response variable. The goal of this study is to model the linear relationship between the explanatory variables and response (SalePrice) variable. Starting with the fitting the model by linear predictor functions that the unknown coefficients are estimated through linear regression:

```
model.lr = lm(SalePrice~., data = train.H)
summary(model.lr)
```

```
##
## Call:
## lm(formula = SalePrice ~ ., data = train.H)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -109496  -17397    -673   18981  193841
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -4.823e+04  1.420e+04  -3.397 0.000752 ***
## LotFrontage  -9.080e+01  1.001e+02  -0.907 0.365111
## LotArea       2.054e-01  2.319e-01   0.886 0.376407
## LotShapeIR2   2.610e+04  1.104e+04   2.364 0.018561 *
## LotShapeIR3  -1.983e+04  2.594e+04  -0.765 0.445010
## LotShapeReg  -6.935e+03  3.987e+03  -1.739 0.082750 .
## OverallCond   4.565e+03  1.616e+03   2.825 0.004978 **
## MasVnrArea    4.323e+01  1.113e+01   3.884 0.000121 ***
## TotalBsmtSF   5.161e+01  7.362e+00   7.010 1.08e-11 ***
## X1stFlrSF     6.212e+01  2.990e+01   2.078 0.038403 *
## X2ndFlrSF     6.528e+01  2.921e+01   2.235 0.025999 *
## GrLivArea     1.340e+01  2.882e+01   0.465 0.642151
## TotRmsAbvGrd -3.022e+03  2.202e+03  -1.372 0.170749
## GarageArea    1.077e+02  1.092e+01   9.865  < 2e-16 ***
## WoodDeckSF    2.740e+01  1.596e+01   1.717 0.086807 .
## OpenPorchSF   1.102e+02  3.051e+01   3.612 0.000344 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 36010 on 384 degrees of freedom
## Multiple R-squared:  0.8022, Adjusted R-squared:  0.7945
## F-statistic: 103.8 on 15 and 384 DF,  p-value: < 2.2e-16
```

As shown above, several variables are not statistically significant, for instance "LotFrontage", "GrlivArea", "TotRmsAbvGrd" and etc by linear regression. Therefore, we have to remove those inactive terms from our model, in order to select our parsimonious model.

Model tuning

We will implement stepwise selection in this case, which is a combination of forward and backward selections. We start with no predictors, then sequentially add the most contributive predictors (like forward selection). After adding each new variable, remove any variables that no longer provide an improvement in the model fit (like backward selection). Only 5 (keeping the 5 variables with significance >0.01 from previous result) most contributive predictor will be specify to incorporate in the model.

```
model.stepwise = stepAIC(model.lr, direction = "both",trace = FALSE)
summary(model.stepwise)
```

```
##
```

```
## Call:
## lm(formula = SalePrice ~ LotShape + OverallCond + MasVnrArea +
##     TotalBsmtSF + X1stFlrSF + X2ndFlrSF + TotRmsAbvGrd + GarageArea +
##     WoodDeckSF + OpenPorchSF, data = train.H)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -111669  -17538     623   18141  193097
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -51853.395  13502.962  -3.840 0.000144 ***
## LotShapeIR2   28840.766  10622.154   2.715 0.006921 **
## LotShapeIR3  -17673.254  25763.282  -0.686 0.493133
## LotShapeReg   -7045.715   3964.467  -1.777 0.076318 .
## OverallCond    4749.692   1604.058   2.961 0.003255 **
## MasVnrArea       42.543     11.050   3.850 0.000138 ***
## TotalBsmtSF      51.520      7.344   7.015 1.03e-11 ***
## X1stFlrSF        75.343     10.364   7.270 2.01e-12 ***
## X2ndFlrSF        78.677      7.723  10.187  < 2e-16 ***
## TotRmsAbvGrd  -3114.136   2162.691  -1.440 0.150694
## GarageArea      106.354     10.751   9.893  < 2e-16 ***
## WoodDeckSF       28.428     15.861   1.792 0.073869 .
## OpenPorchSF     108.886     30.378   3.584 0.000381 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 35930 on 387 degrees of freedom
## Multiple R-squared:  0.8015, Adjusted R-squared:  0.7953
## F-statistic: 130.2 on 12 and 387 DF,  p-value: < 2.2e-16
```

```
model_6 = regsubsets(SalePrice~., data = train.H, nvmax = 5,method = "seqrep")
summary(model_6)
```

Then estimate the average prediction error(RMSE) of each of the 5 models select in the previous part by using 10-fold cross validation. The best model with the smallest RMSE will be regarded as the final best model.

```
# Set up repeated 10-fold cross-validation
train.control <- trainControl(method = "cv", number = 10)
# Train the model
model.train <- train(SalePrice ~.,
                     data =train.H,method = "leapBackward",
                     tuneGrid = data.frame(nvmax = 1:5),
                     trControl = train.control)
model.train$results
```

```
##   nvmax     RMSE  Rsquared      MAE   RMSESD RsquaredSD    MAESD
## 1     1 62537.71 0.3840155 48311.19 6731.629 0.14819297 5016.981
## 2     2 47755.35 0.6412343 35028.37 8243.594 0.10385101 6567.821
## 3     3 42389.28 0.7193646 30729.35 8349.216 0.08117616 4655.867
## 4     4 38195.74 0.7675663 27788.62 7304.897 0.07170693 4083.460
## 5     5 38851.42 0.7611599 28212.90 6807.508 0.07261818 3925.068
```

```
model.train$bestTune
```

```
##   nvmax
## 4     4
```

The best tuning value nvmax= 4 variables would be our best model, then check the final model selected with its coefficients

```
coef(model.train$finalModel, 4)
```

```
##  (Intercept)  TotalBsmtSF    X1stFlrSF    X2ndFlrSF   GarageArea
## -45468.47220     58.26186     70.88679     81.05283    117.80380
```

```
final.model= lm(SalePrice~TotalBsmtSF+X1stFlrSF+X2ndFlrSF+GarageArea,data=train.H)
```

Then our final best 4-variables model contains **"TotalBsmtSF", "X1stFlrSF", "X2ndFlrSF", "GarageArea"**. We will calculate the cross validation error for this model and compare with the original full model.

```
#cv error
nsample = nrow(train.H)
fold_label = sample(10, nsample, replace=T)
all_MSE <- c()
all_n_k <- c()
for (k in 1:10){
  test_label = which(fold_label==k)
  train_label = which(fold_label!=k)
  train_data = train.H[train_label,]
  test_data = train.H[test_label,]
  model.final <- lm(SalePrice ~ TotalBsmtSF+X1stFlrSF+X2ndFlrSF+GarageArea,
                    data = train_data)
  test_data_y_hat <- predict(model.final,newdata=test_data)
  MSE_k <- mean((test_data$SalePrice - test_data_y_hat)^2)
  n_k <- length(test_data_y_hat)
  all_MSE <- c(all_MSE, MSE_k)
  all_n_k <- c(all_n_k, n_k)
}
CV.lr <- sum(all_MSE*all_n_k/nsample)
print(paste("CVerror for final model is",CV.lr))
```

```
## [1] "CVerror for final model is 1498836610.76567"
```

As shown above, we can see the cross validation error of our final model is nearly half of the original one, but it is still very large which around 1.5billions, we believe that the data complexity and diverse data type contribute to the high prediction error.

Interpretation

```
pred = predict(final.model,test.H)
RMSE.lr=sqrt(mean((pred - test.H$SalePrice)^2))
print(paste("RMSE =",sqrt(mean((pred - test.H$SalePrice)^2))))
```

```
## [1] "RMSE = 34100.058060633"
```

In view of huge cross-validation error and reletively great root-mean-square error (**RMSE= 34100.06**) in prediction. Linear regression would not be an accurate and precise prediction method on predicting the sale-price in this study due to collinearity between all explanatory variables and the dependent variable SalePrice is not collinear is all situations.

1.3 Random Forest

Random Forest approach is a supervised learning algorithm (tree-based). A random forest is a collection of decision trees and thus, it does not rely on a single feature. Therefore, by combining multiple predictions from each decision tree (or regression tree), a more accurate and stable prediction will be urged. The random forest is similar to Bagging (an ensemble technique). In this approach, multiple trees are generated by bootstrap samples from training dataset and then we simply reduce the correlation between the trees. With multiple decision trees, each tree draws a sample random data giving the random forest more randomness to produce better accuracy than decision trees. Performing this approach increases the performance of decision trees and helps in avoiding overfitting.

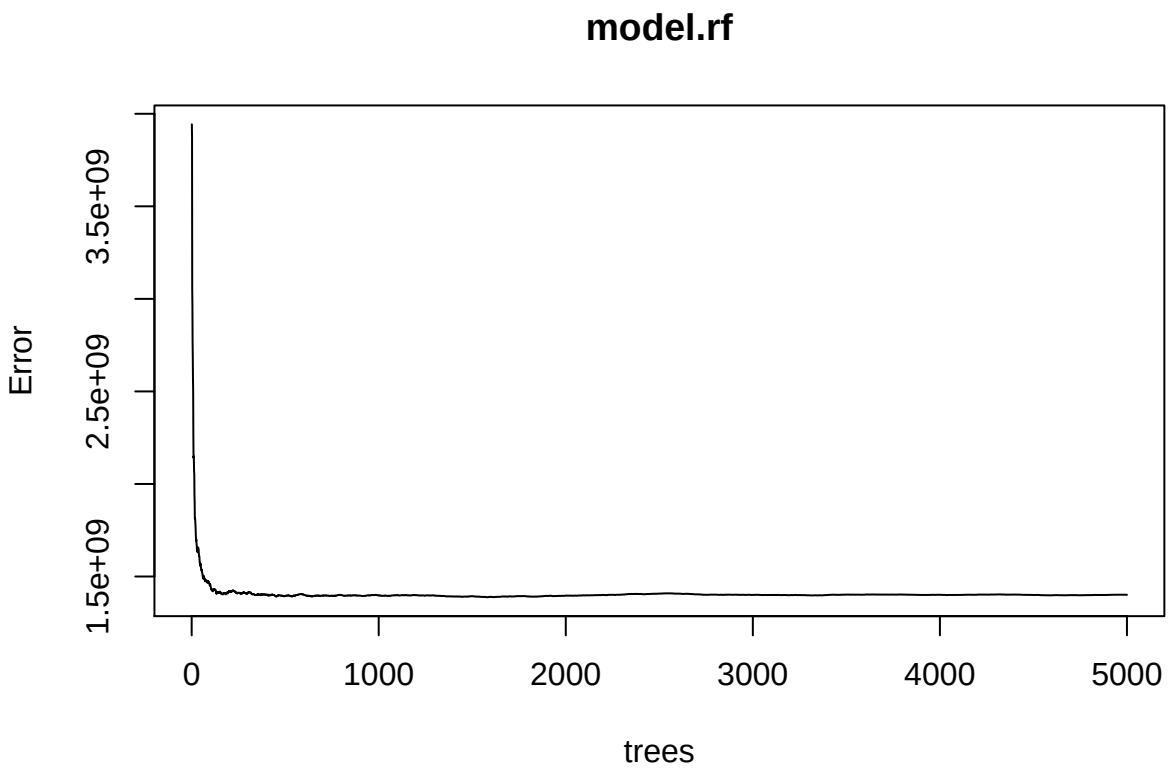
Create Random Forest Model

In this case study, we will stick to tuning parameters: **number of variables randomly sampled as candidates at each split("mtry")** and **number of trees to grow("ntree")** and **maximum depth of the tree("max.depth")** and **minimum number of instance in node("node.size")** that perhaps are the parameters most likely to have the biggest effect on our final accuracy.

```
model.rf = randomForest(SalePrice~.,data=train.H,ntree=5000)
print(model.rf)
```

```
##
## Call:
##  randomForest(formula = SalePrice ~ ., data = train.H, ntree = 5000)
##                Type of random forest: regression
##                      Number of trees: 5000
## No. of variables tried at each split: 4
##
##           Mean of squared residuals: 1401497005
##                     % Var explained: 77.73
```

The default value of "mtry" is 4(around [no.of variables-1]/3) while we set the "ntree" to 5000 for our regression model. Then plot the error against number of trees to see the effect of adding more trees on the error



Obviously the sizable amounts of trees grown, the lower the error would be.Therefore, the ntree parameter can be as large as possible, and continues to increases the accuracy up to some point around 1000 trees, then we will set **"ntree"=1000** in the following parameter tuning.

Parameters Tuning

After select the number of trees of our model, we are interested in the optimal value of the mtry, minimum node size and corresponding maximum tree depth. Similar to bagging, a natural benefit of the bootstrap resampling process is that random forests have an **out-of-bag (OOB)** sample that provides an efficient and reasonable approximation of the test error. This provides a built-in validation set without any extra work on our part, and we do not need to sacrifice any of our training data to use for validation. This makes identifying the number of trees required to stablize the error rate during tuning more efficient.

First, we use OOB error to check the effects of value of minimum node size and maximum tree depth on the accuracy in prediction with mtry set to default value as previous(**mtry=4**).

```
#pick best 5 node size
set.seed(4011)
hyper_grid <- expand.grid(
  node_size = seq(1, 10, by = 1),
  OOB_RMSE = 0
)
for (i in 1:nrow(hyper_grid)) {
  # train model
  model <- ranger(
    formula = SalePrice ~ .,
```

```
    data = train.H,
    num.trees = 1000,
    mtry = 4,
    min.node.size = hyper_grid$node_size[i],
    max.depth = 10,
    seed = 4011
    )
  hyper_grid$OOB_RMSE[i] <- sqrt(model$prediction.error)
}
hyper_grid=hyper_grid[order(hyper_grid$OOB_RMSE),]
hyper_grid %>%
  dplyr::arrange(OOB_RMSE) %>%
  head(5)
```

```
##   node_size OOB_RMSE
## 1         1 37625.55
## 2         2 37674.18
## 3         3 37688.11
## 4         4 37775.47
## 5         5 37855.28
```

The next step is to train the top 5 models from above with the 10-fold cross-validation method to obtain the cross validation error, so as to select our model with the lowest cross-validation error.

```
#cv error
nsample = nrow(train.H)
fold_label = sample(10, nsample, replace=T)
all_MSE <- c()
all_n_k <- c()
for (i in 1:5) {
  for (k in 1:10){
  test_label = which(fold_label==k)
  train_label = which(fold_label!=k)
  train_data = train.H[train_label,]
  test_data = train.H[test_label,]
  model.final = randomForest(SalePrice~.,data=train.H,ntree=1000,
                             mtry=4,
                             nodesize=hyper_grid[i,1])
  test_data_y_hat = predict(model.final,newdata=test_data)
  MSE_k = mean((test_data$SalePrice - test_data_y_hat)^2)
  n_k = length(test_data_y_hat)
  all_MSE = c(all_MSE, MSE_k)
  all_n_k = c(all_n_k, n_k)
  }
  CV = sum(all_MSE*all_n_k/nsample)
print(paste("CVerror for",i,"th model is",CV))
}
```

```
## [1] "CVerror for 1 th model is 191474513.900795"
## [1] "CVerror for 2 th model is 396299290.94502"
## [1] "CVerror for 3 th model is 618339714.428445"
## [1] "CVerror for 4 th model is 857740957.886868"
## [1] "CVerror for 5 th model is 1123464838.94839"
```

The fist model has the lowest cross-validation error, thus we will regard the value of "mtry" and minimum node size from the first model as our prediction model's parameters.

```
## [1] "RMSE of original model is 29754.4933002814"
```

```
## [1] "RMSE of final model is 29986.395115252"
```

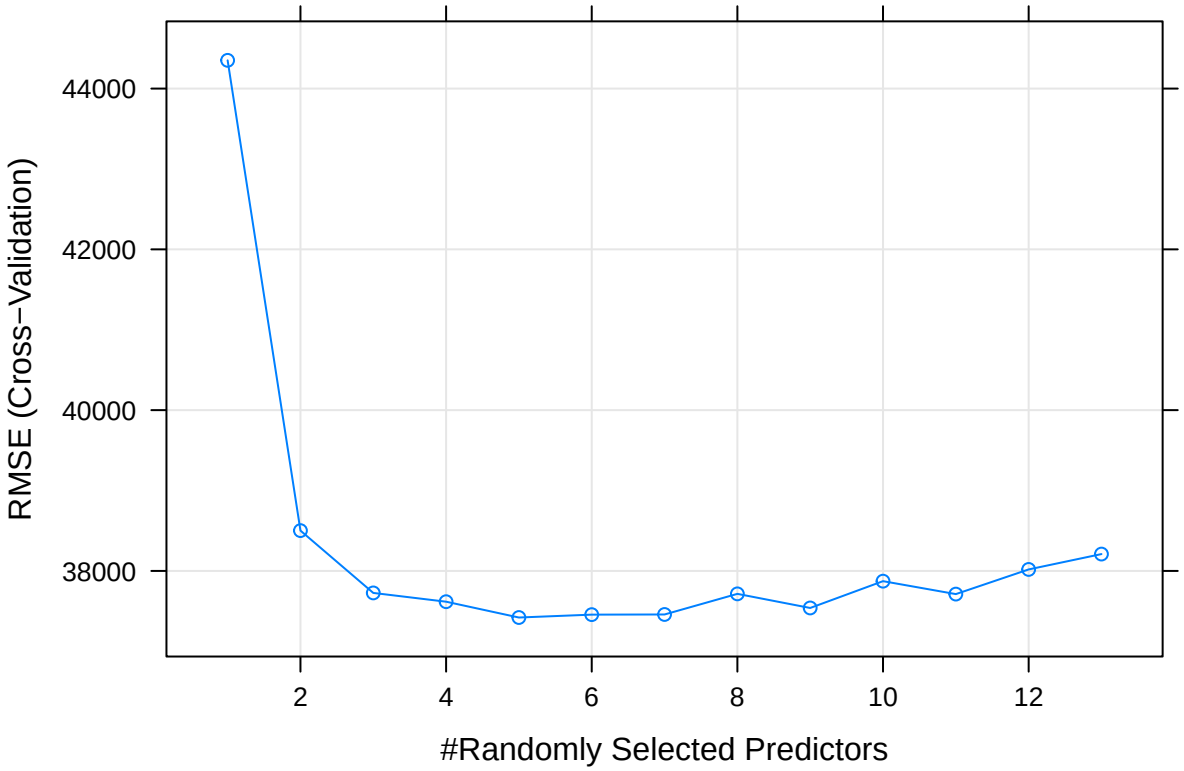
Shown as above, we can see the RMSE of the original model and our selected model is very close, thus it means the minimum node size and tree depth would hold a significant effect on our model accuracy much, thus, only **mtry** would possibly have the biggest effect on our model.

Then, we will perform grid search to identify the optimal "mtry" value. It define a grid of algorithm parameters to try, each axis of the grid is an algorithm parameter, and points in the grid are specific combinations of parameters. Because we are only tuning one parameter, the grid search is a linear search through a vector of candidate values.

```
control = trainControl(method = 'cv',number=10,search = "grid")
tune_Grid = expand.grid(.mtry=c(1:13))
rf=train(SalePrice~.,data=train.H,
         method="rf",metric="RMSE",
         ntree=1000,
         trControl=control,
         tuneGrid=tune_Grid)
print(rf$finalModel)
```

```
##
## Call:
##  randomForest(x = x, y = y, ntree = 1000, mtry = param$mtry)
##                Type of random forest: regression
##                      Number of trees: 1000
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 1403698098
##                     % Var explained: 77.69
```

```
plot(rf)
```



From the plot, we can observe the RMSE at mtry in range 3 to 8 is very close which is resonable, then the prediction error is calculate as below:

```
#cv error
all_MSE <- c()
all_n_k <- c()
pred= predict(rf,test.H)
RMSE.rf=sqrt(mean((pred - test.H$SalePrice)^2))
print(paste("The RMSE of final model is ",sqrt(mean((pred - test.H$SalePrice)^2))))
```

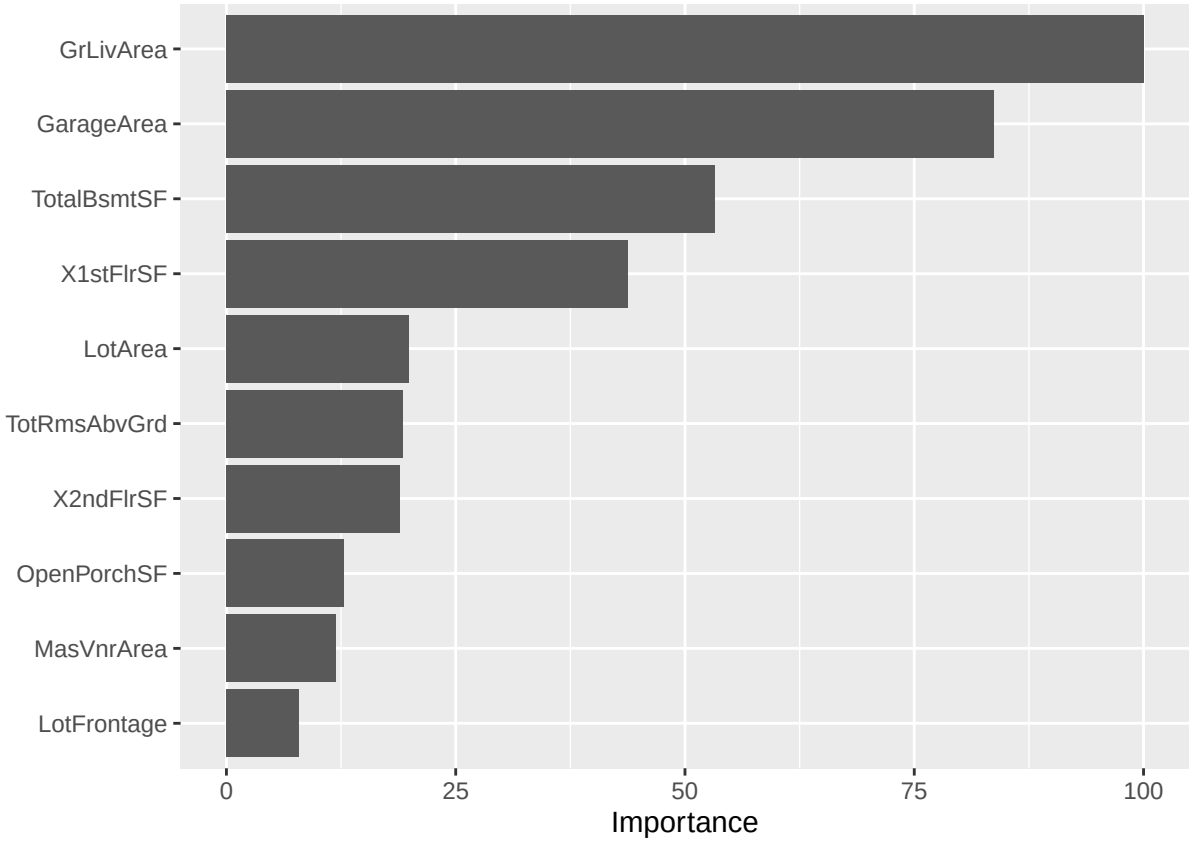
```
## [1] "The RMSE of final model is  29630.429704031"
```

```
for (k in 1:10){
  test_label = which(fold_label==k)
  train_label = which(fold_label!=k)
  train_data = train.H[train_label,]
  test_data = train.H[test_label,]
  model.final = randomForest(SalePrice~.,data= train_data,
                             ntree=1000,
                             mrty=rf$bestTune)
  test_data_y_hat <- predict(model.final,newdata=test_data)
  MSE_k <- mean((test_data$SalePrice - test_data_y_hat)^2)
  n_k <- length(test_data_y_hat)
  all_MSE <- c(all_MSE, MSE_k)
  all_n_k <- c(all_n_k, n_k)
}
```

```r
CV.rf <- sum(all_MSE*all_n_k/nsample)
print(paste("CVerror for final model is",CV.rf))
```

```
## [1] "CVerror for final model is 1413431923.37317"
```

Variable Importance Plot



1.4 Boosting

Unlike many machine learning algorithms which focus on high quality prediction done by single model, boosting is a generic algorithm rather than a specific model that improve the prediction power by training a sequence of weaker prediction models. Therefore, we need to specify a weak model then improves it by using boosting. There are two famous boosting algorithms: **Adaptive Boosting(Adaboost)** and **Gradient Boosting Machine(GBM)**.

The difference between the algorithms is **Adaboost** identifies miss-classified data points, then increase their weights (decrease the weights of correct points, in a sense) so that the next classifier will pay extra attention to get them right, in each iteration. The weakness is identified by the estimator's error rate(usually develop for classification problems).

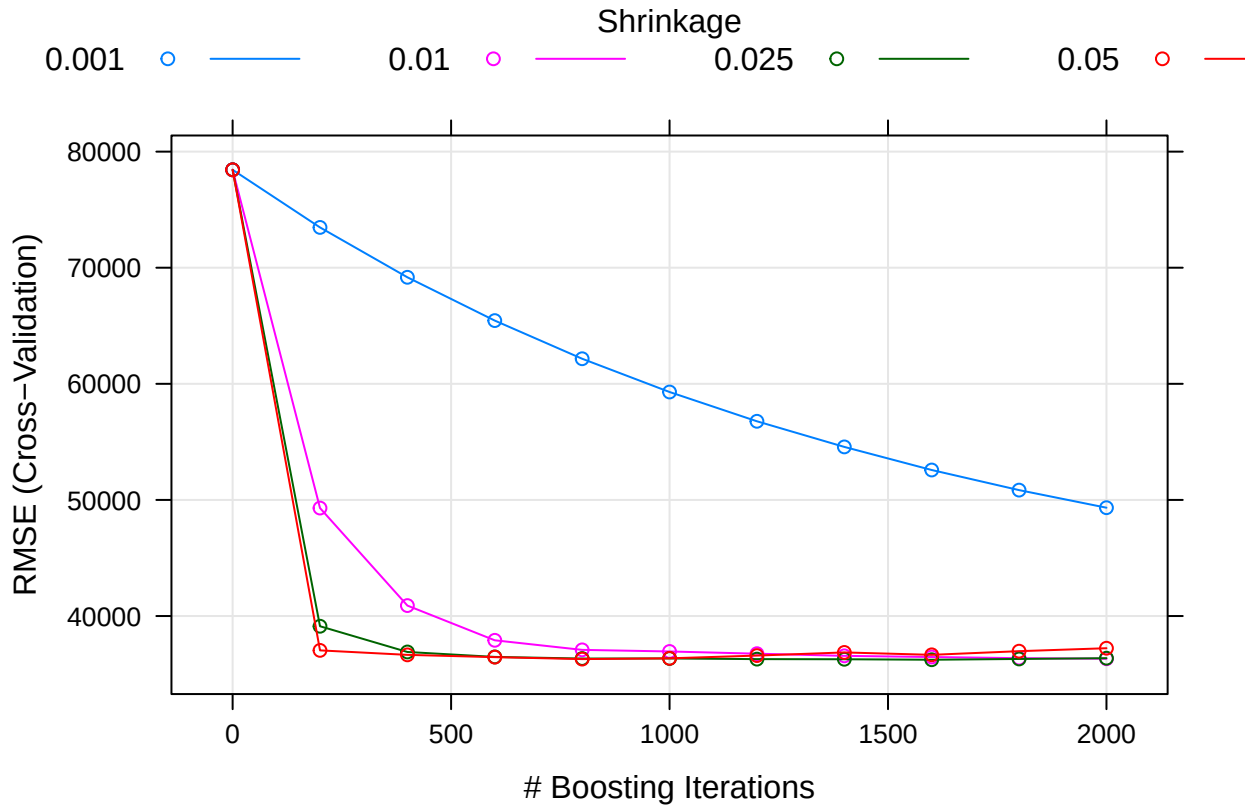
In stead of adjusting the weights of data points, **GBM(method focused in this study)** focus on the difference between prediction and truth (residuals) in regression approach. Besides, GBM uses gradient descent to obtain the optimize solution. In practice, boosted algorithms almost always use decision trees (or regression tree) as the weaker model.

Exploring GBM model

If we train a GBM model with all default parameters values at the very beginning like other algorithms, it never suffices. With default very small **learning rate(shrinkage)= 0.001** the model typically reqiures enormous number of trees(since GBM can be regarded as a tree-based algorithm like random forest) to minimize the MSE, thus it would be extremely time consuming. Then we start optimize the boosting parameters including: **learning rate("shrinkage")**, **number of trees("n.tree")**,**tree depth("interaction.depth")**,**number of observations allowed in tree terminal nodes("n.minobsinnode")** by fitting a model with relative large learning rate in order to, get an overview on how one parameters would affect another.

```
set.seed(1234)
nfold = invisible(trainControl( method = "cv", number = 10))
grid.eg = expand.grid(interaction.depth=1,
                          n.trees = 200*(0:10),
                          shrinkage=c(0.01,0.05, 0.025, 0.001),
                          n.minobsinnode=10)
gbm.trail = train(SalePrice ~ .,
                  data = train.H,
                  method = 'gbm',
                  trControl = nfold,
                  tuneGrid=grid.eg,
                  metric = "RMSE")
```

```
plot(gbm.trail)
```



The general idea of gradient descent is to tune parameters iteratively in order to minimize a cost function, which similar to find the smallest area under the curve(a model corresponding set of parameters value) from the start the the point with lowest RMSE. For illustration, the model with shrinkage= 0.001 needs more tress to hit the minimum MSE value. On the other hand model with shrinkage=0.05 takes less than 500 tress to retain the lowest RMSE.

Tuning Parameters

After knowing what the criterion of the optimal model should have, we start to tune the parameters with different combination sets

```
nfold = invisible(trainControl( method = "cv", number = 10))
tune.Grid = expand.grid(interaction.depth=seq(1,10,by=2),
                        n.trees = 100*(1:20),
                       #0.001 perform worst from previous part,rejected
                        shrinkage=c(0.01,0.25,0.5,0.75,0.1),
                        n.minobsinnode=c(5,10,15))
gbm.fit = train(SalePrice ~ .,
                data = train.H,
                method = 'gbm',
                trControl = nfold,
                tuneGrid=tune.Grid,
                metric = "RMSE")
```

Model summary

The optimal value of parameters are as below:

```
#RMSE
gbm.fit$bestTune
```

```
##     n.trees interaction.depth shrinkage n.minobsinnode
## 134    1400                 5      0.01              5
```

```
pred=predict(gbm.fit,test.H)
RMSE.gbm=sqrt(mean((pred- test.H$SalePrice)^2))
print(paste("RMSE=",RMSE.gbm))
```

```
## [1] "RMSE= 30517.1397737013"
```

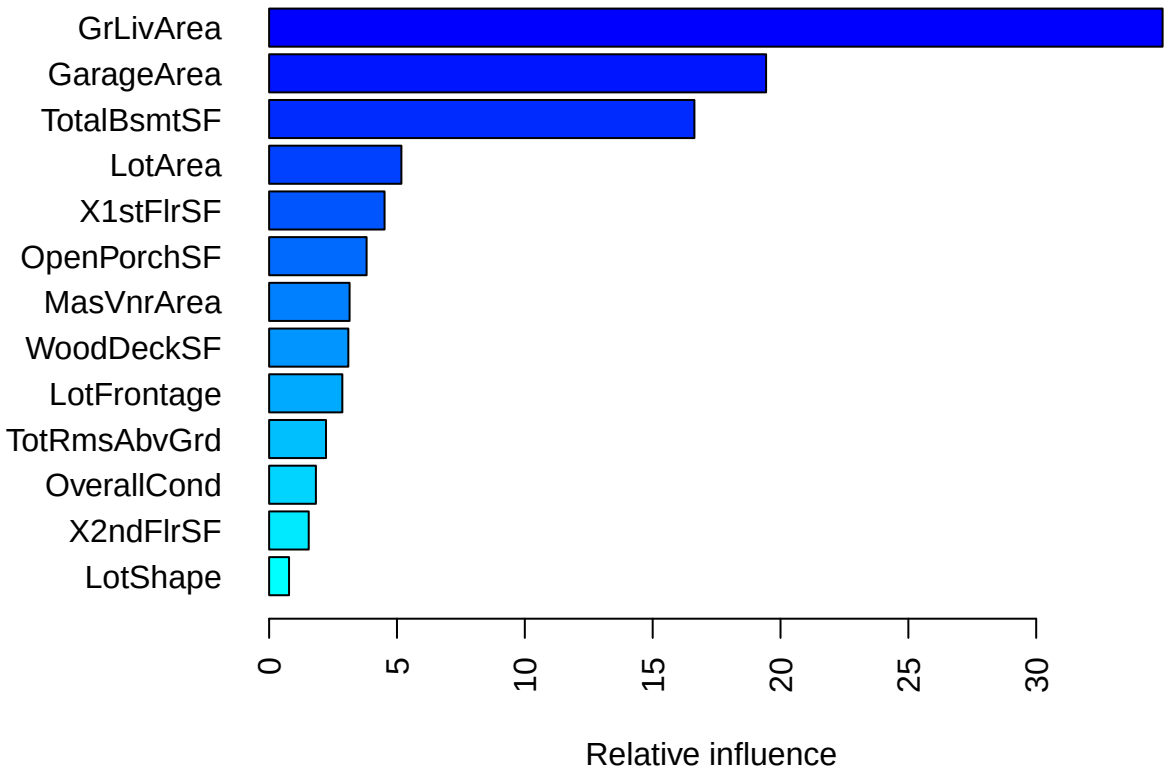
```
#CV error
all_MSE <- c()
all_n_k <- c()
for (k in 1:10){
  test_label = which(fold_label==k)
  train_label = which(fold_label!=k)
  train_data = train.H[train_label,]
  test_data = train.H[test_label,]
  model.final = gbm(formula = SalePrice ~ .,
  distribution = "gaussian",
  data = train_data,
```

```
  n.trees = gbm.fit$bestTune[1],
  interaction.depth = gbm.fit$bestTune[2],
  shrinkage = gbm.fit$bestTune[3],
  n.minobsinnode = gbm.fit$bestTune[4],
  n.cores = NULL,
  verbose = FALSE
  )
  test_data_y_hat <- predict(model.final,newdata=test_data)
  MSE_k <- mean((test_data$SalePrice - test_data_y_hat)^2)
  n_k <- length(test_data_y_hat)
  all_MSE <- c(all_MSE, MSE_k)
  all_n_k <- c(all_n_k, n_k)
}
CV.gbm <- sum(all_MSE*all_n_k/nsample)
print(paste("CVerror for final model is",CV.gbm))
```

```
## [1] "CVerror for final model is 1317208032.15892"
```

The best model with **number of trees equals to 2000**, **maximum depth of trees equals to 5** and the **learning rate is equal to 0.1**,**5** observations allowed in tree terminal nodes.

Variable Importance plot




```
##                       var    rel.inf
```

```
## GrLivArea       GrLivArea 34.9430645
## GarageArea     GarageArea 19.4362348
## TotalBsmtSF   TotalBsmtSF 16.6325220
## LotArea           LotArea  5.1728901
## X1stFlrSF       X1stFlrSF  4.5145087
## OpenPorchSF   OpenPorchSF  3.8115594
## MasVnrArea     MasVnrArea  3.1469794
## WoodDeckSF     WoodDeckSF  3.0977511
## LotFrontage   LotFrontage  2.8629033
## TotRmsAbvGrd TotRmsAbvGrd  2.2243203
## OverallCond   OverallCond  1.8288783
## X2ndFlrSF       X2ndFlrSF  1.5490409
## LotShape         LotShape  0.7793472
```

The vairable importance plot of GBM suggest obvious influnce of ground living area toward saleprice (compare to other variables) than random forest does.

1.5 Conclusion

```
## [1] "RMSE of Liner Regression= 34100.058060633 CV error= 1498836610.76567"
```

```
## [1] "RMSE of Random Forest= 29630.429704031 CV error= 1413431923.37317"
```

```
## [1] "RMSE of GBM= 30517.1397737013 CV error= 1317208032.15892"
```

Linear regression is the simplest method to implement among 3 method, but this is double edge sword, which scarifice the accuracy so it has the lowest RMSE. Suprisingly, the RMSE of random forest and GBM is close to each other, probably as they are all tree based methods which both sacrifice the training time and rise the model compexity in order to obtain higher accuray. However, if we take a look on the cross validation error, GBM would be much better than others. Thus, the best model for prediction in this study would be **Gradient Boosting Machine**.

2.Classification

Import the House dataset

```
Titanic = data.frame(read.csv("/Users/ericyau/Desktop/Titanic.csv"))
```

required packages

```
library(caret)
library(class)
library(e1071)
library(kernlab)
```

2.1Data Cleaning

```
summary(Titanic)
```

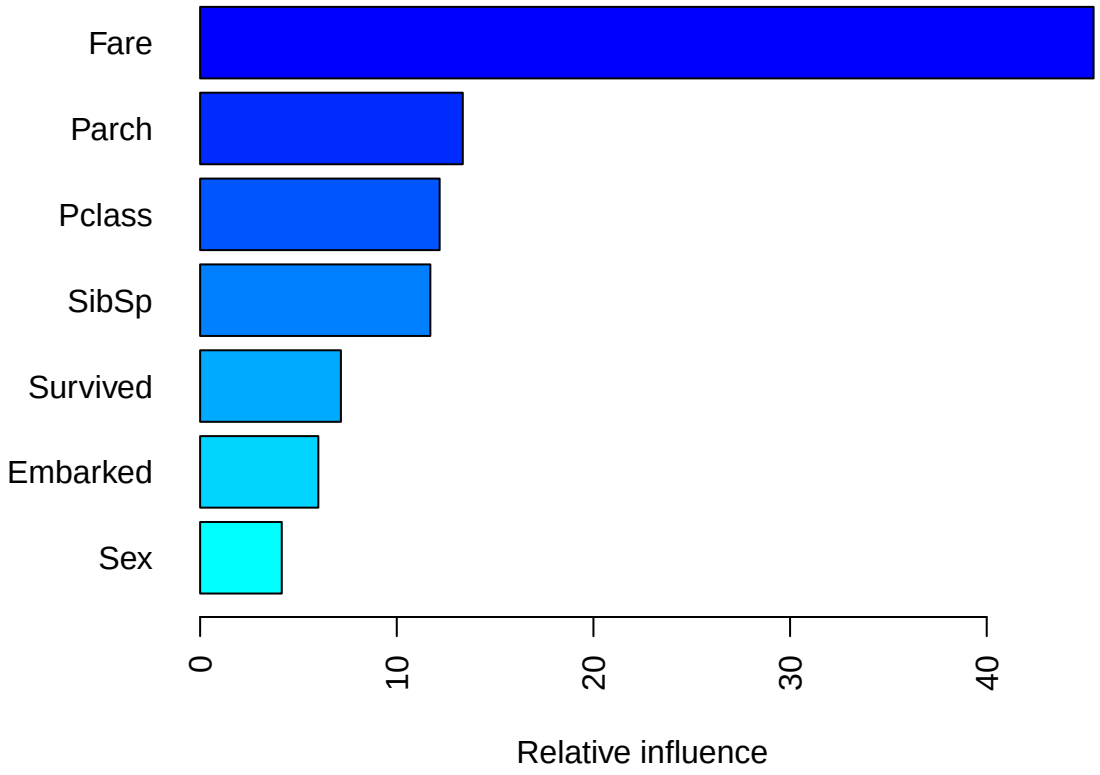
Like the previous part, there are some missing values in the dataset(**102 random missing ages**), however due to the previous part's study (accuracy in prediction), gradient boosting machine will be implement for imputation instead of regression imputation. The dataset will be divided to training set with no missing value and testing set with all missing age observation. Then perform the same process as previous.

```
Titanic$Sex = as.factor(Titanic$Sex)
Titanic$Embarked = as.factor(Titanic$Embarked)
Titanic = Titanic[,-1]
test.age = Titanic[is.na(Titanic$Age), ]
train.age = Titanic[!is.na(Titanic$Age),]
nfold = invisible(trainControl( method = "cv", number = 10))
tune.Grid = expand.grid(interaction.depth=seq(1,10,by=2),
                        n.trees = 100*(1:20),
                        shrinkage=c(0.01,0.25,0.5,0.1),
                        n.minobsinnode=c(5,10,15))
gbm.age = train(Age ~ .,
                data = train.age,
                method = 'gbm',
                trControl = nfold,
                tuneGrid=tune.Grid,)
```

```
##     n.trees interaction.depth shrinkage n.minobsinnode
## 244     400                 9      0.01              5
```

The optimal parameters value for the model is shown above. Then study the variable imporatnce plot to check the contribution of different features to age.

```
gbm.age.final = gbm(
          formula = Age ~ .,
          distribution = "gaussian",
          data = train.age,
          n.trees = 800,
          interaction.depth = 9,
          shrinkage = 0.01,
          n.minobsinnode = 5)
par(mar = c(5, 8, 1, 1))
summary(gbm.age.final,las=2)
```




```
##                 var   rel.inf
## Fare           Fare 45.434644
## Parch         Parch 13.357279
## Pclass       Pclass 12.180076
## SibSp         SibSp 11.706800
## Survived   Survived  7.158109
## Embarked Embarked  6.011562
## Sex             Sex  4.151529
```

After checking the variable importance plot, we substitue the predicted age value into those missing value.

```
pred.age = predict(gbm.age.final,test.age)
test.age$Age = round(as.numeric(pred.age))
data = rbind(train.age, test.age)
data[,1] = as.factor(data[,1])
data[,2] = as.factor(data[,2])
```

Last we split the dataset into training set and testing set

```
sample = sample(nrow(Titanic), nrow(Titanic)*0.8)
train.T = data[sample,]
test.T = data[-sample,]
```

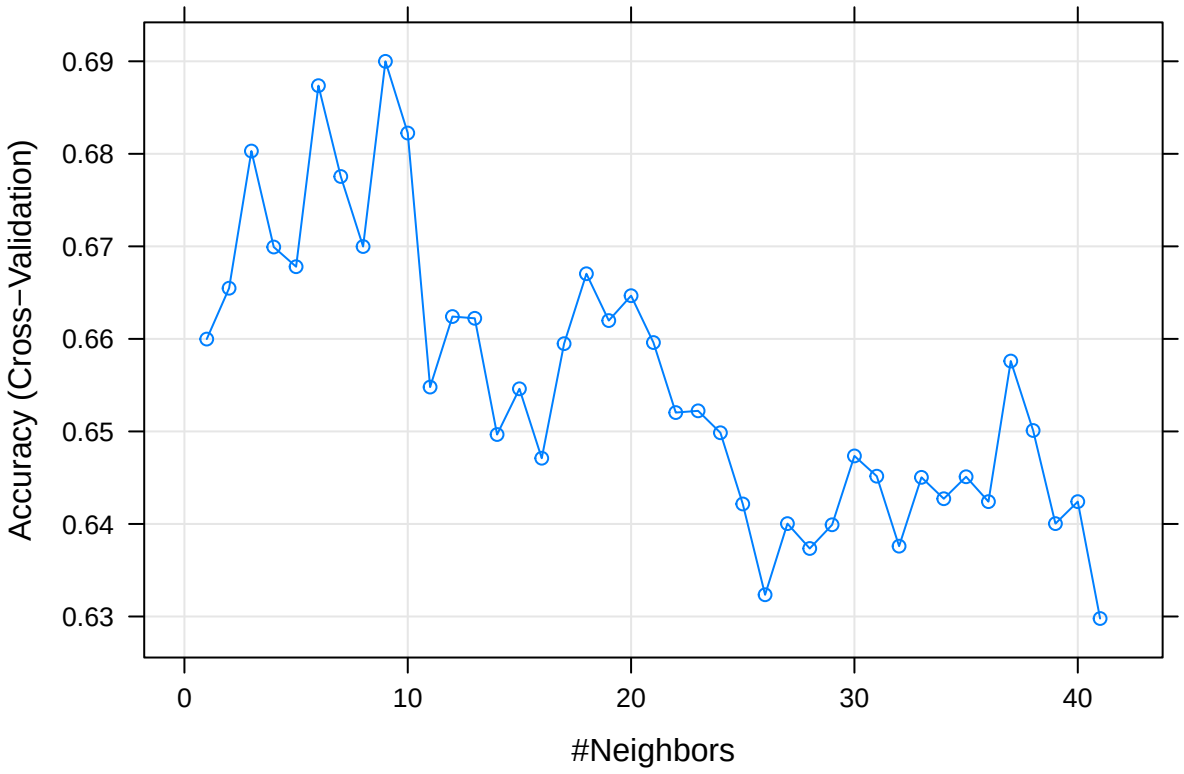
2.2 K-nearest neighbours (KNN)

K Nearest Neighbor is a Supervised Machine Learning algorithm that classifies a new data point into the target class, depending on features of its neighboring data points . For instance, we input datasets of men images and women images and our model is trained to detect the gender based on some features, the the knn algorithm would classify a new data input to either male or female depends on the similarity in their features.

Parameter Tuning

The parameter **K** in KNN stands for the number of Nearest Neighbors, thus we have to opitimze the value of K inorder to optimize our model. Typically, we starting the K with $K = \sqrt{n}+1$ (n= number of obseravtions in training set) thus we are interested the effect of K varys from 1 to K on the accuracy(10-fold cross validation result).

```
train.control = trainControl(method  = "CV", number = 10)
#set starting k as meadian
grid= expand.grid(k= seq(1,2*sqrt(nrow(train.T))+1,by=1))
KNN = train(Survived ~ .,
            method     = "knn",
            tuneGrid   = grid,
            trControl  = train.control,
            metric     = "Accuracy",
            data       = train.T)
plot(KNN)
```



```r
print(paste("the optimal value K is",KNN$bestTune))
```

```
## [1] "the optimal value K is 9"
```

From the plot we can see the accuracy of $K < 21$ holds the best accuracy interval and optimal k value is around the range 5 to 9

Result

```r
#accuracy
train.knn=train.T
train.knn$Survived=as.factor(train.knn$Survived)
train.knn$Pclass=as.numeric(train.knn$Pclass)
train.knn$Sex=as.numeric(train.knn$Sex)
train.knn$Embarked=as.numeric(train.knn$Embarked)
pred= predict(KNN, test.T)
CM.knn=confusionMatrix(pred, test.T$Survived)
print(paste("Accuracy of KNN is",CM.knn$overall[1]))
```

```
## [1] "Accuracy of KNN is 0.7"
```

```r
acc=c()
#cv error
for (k in 1:10){
  test_label = which(fold_label==k)
  train_label = which(fold_label!=k)
  train_data = train.knn[train_label,]
  test_data = train.knn[test_label,]
  model.final = knn(train = train_data,test= test_data,
                    cl=train_data$Survived,
                    k=KNN$bestTune )
  acc[k] =1- sum(test_data$Survived == model.final)/NROW(test_data$Survived)
}
acc.knn= mean(acc)
print(paste("CVerror for final model is",acc.knn))
```

```
## [1] "CVerror for final model is 0.300737868367906"
```

2.3 Support Vector Machine

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples.The goal of an SVM is to take groups of observations and construct boundaries to predict which group future observations belong to based on their measurements. The different groups that must be separated will be called “classes”. SVMs can handle any number of classes, as well as observations of any dimension. SVMs can take almost any shape (including linear, radial, and polynomial,

among others), and are generally flexible enough to be used in almost any classification endeavor that the user chooses to undertake.

There are two main situations for determining the hyperplane which are 1. When the hyperplanes segregate the classes well, the hyperplane having the maximum distance between nearest data point will be considered as the right hyperplane to classify the classes better. 2. When the hyperplanes do not segregate the classes well, the thumb rule to be known, before finding the right hyper plane, to classify star and circle is that the hyper plane should be selected which segregate two classes better.

Tuning

We start the tuning with the kernel selection,a total of 3 models are implement with their own paramters combinations(10-fold cross validation was done to optimize the parameter in each kernel), inorder to determine the kernel with the highest accuracy.

```
linear.tune = train(Survived ~., data = train.T,
                    method = "svmLinear",
                    trControl = trainControl("cv", number = 10),
                    tuneGrid = expand.grid(C = c(0.1,0.5,1,2,5,10 )))
rad.tune = train(Survived ~., data = train.T,
                    method = "svmRadial",
                    trControl = trainControl("cv", number = 10),
                    tuneLength=10)
poly.tune = train(Survived ~., data = train.T,
                    method = "svmPoly",
                    trControl = trainControl("cv", number = 10),
                    tuneLength=4)
```

```
## [1] "parameter for linear kernel: Cost= 0.1"
```

```
## [1] "parameter for radial kernel: Cost= 4 , sigma= 0.138339829289346"
```

```
## [1] "parameter for polynomial kernel: Cost= 1 , degree= 2 , Scale= 0.1"
```

```
linear.test = predict(linear.tune, test.T)
CM.lin= confusionMatrix(linear.test, test.T$Survived)
print(paste("the accuracy of linear kernel is",CM.lin$overall[1]))
```

```
## [1] "the accuracy of linear kernel is 0.85"
```

```
rad.test = predict(rad.tune, newdata=test.T)
CM.rad= confusionMatrix(rad.test, test.T$Survived)
print(paste("the accuracy of radial kernel is",CM.rad$overall[1]))
```

```
## [1] "the accuracy of radial kernel is 0.8"
```

```
poly.test = predict(poly.tune, newdata=test.T)
CM.ploy= confusionMatrix(poly.test, test.T$Survived)
print(paste("the accuracy of polynomial kernel is",CM.ploy$overall[1]))
```

```
## [1] "the accuracy of polynomial kernel is 0.84"
```

then our final model would be a made by 10-fold cross validation the corss validation error:

```
acc=c()
#cv error for linear kernel
for (k in 1:10){
  test_label = which(fold_label==k)
  train_label = which(fold_label!=k)
  train_data = train.T[train_label,]
  test_data = train.T[test_label,]
  model.final = svm(Survived ~., data = train_data,
                    type = 'C-classification',
                    kernel = 'linear',
                    cost=linear.tune$bestTune[1])
  pred= predict(model.final,test_data)
  cm= confusionMatrix(pred, test_data$Survived)
  acc[k] = 1-cm$overall[1]
}
acc.svm1= mean(acc)
print(paste("CVerror for linear kernel model is",acc.svm1))
```

```
## [1] "CVerror for linear kernel model is 0.210000698208907"
```

```
acc=c()
#cv error for radial kernel
for (k in 1:10){
  test_label = which(fold_label==k)
  train_label = which(fold_label!=k)
  train_data = train.T[train_label,]
  test_data = train.T[test_label,]
  model.final = svm(Survived ~., data = train_data,
                    type = 'C-classification',
                    kernel = 'radial',
                    cost=rad.tune$bestTune[2],
                    sigma=rad.tune$bestTune[1])
  pred= predict(model.final,test_data)
  cm= confusionMatrix(pred, test_data$Survived)
  acc[k] = 1-cm$overall[1]
}
acc.svm2= mean(acc)
print(paste("CVerror for radial kernel model is",acc.svm2))
```

```
## [1] "CVerror for radial kernel model is 0.211595236867447"
```

```
acc=c()
#cv error for polynomial kernel
for (k in 1:10){
  test_label = which(fold_label==k)
  train_label = which(fold_label!=k)
  train_data = train.T[train_label,]
  test_data = train.T[test_label,]
  model.final = svm(Survived ~., data = train_data,
                    type = 'C-classification',
```

```
                    kernel = 'polynomial',
                    cost=poly.tune$bestTune[3],
                    degree=poly.tune$bestTune[1],
                    scale=T)
  pred= predict(model.final,test_data)
  cm= confusionMatrix(pred, test_data$Survived)
  acc[k] = 1-cm$overall[1]
}
acc.svm3= mean(acc)
print(paste("CVerror for polunomial kernel model is",acc.svm3))
```

```
## [1] "CVerror for polunomial kernel model is 0.201737666111288"
```

Then the parameters and performance of our final model would be :

```
# create matrix saves all parameters
M=matrix(nrow=3,ncol=6)
M[,1]=c("linear kernel","radial kernel","polynomial kernel")
M[,2]=c(paste("Cost=",linear.tune$bestTune[1]),
        paste("Cost=",rad.tune$bestTune[2]),
        paste("Cost=",poly.tune$bestTune[3]))
M[,3]=c(paste(""),
        paste("sigma=",rad.tune$bestTune[1]),
        paste("degree=",poly.tune$bestTune[1]))
M[,4]=c(paste(""),
        paste(""),
        paste("scale=",poly.tune$bestTune[2]))
M[,5]=c(paste("accuracy=",CM.lin$overall[1]),
        paste("accuracy=",CM.rad$overall[1]),
        paste("accuracy=",CM.ploy$overall[1]))
M[,6]=c(paste("CV error=",acc.svm1),
        paste("Cv error=",acc.svm2),
        paste("Cv error=",acc.svm3))

number= max(c(CM.lin$overall[1],CM.ploy$overall[1],CM.rad$overall[1]))

if(number==CM.lin$overall[1]){
  print(M[1,])
}else{
  if(number==CM.rad$overall[1]){
    print(M[2,])
  }else{
    if(number==CM.ploy$overall[1]){
      print(M[3,])
    }
  }
}
```

```
## [1] "linear kernel"              "Cost= 0.1"
## [3] ""                           ""
## [5] "accuracy= 0.85"             "CV error= 0.210000698208907"
```

2.4 Conclusion

```
cve=min(c(acc.svm1,acc.svm2,acc.svm3))
print(paste("Accuracy of K-nearest neighbours is",CM.knn$overall[1],"CV error=",acc.knn))
```

```
## [1] "Accuracy of K-nearest neighbours is 0.7 CV error= 0.300737868367906"
```

```
print(paste("Accuracy of Support Vector Machine is",max(c(CM.lin$overall[1],CM.ploy$overall[1],CM.rad$o
```

```
## [1] "Accuracy of Support Vector Machine is 0.85 CV error= 0.201737666111288"
```

As the result indicate, the **support vector machine** perform better than KNN in both accuracy and minimized cross validation error, possible reason would be the model complexity of SVM is much complicate than KNN does(only one hyperparameter), due to serveral hyperparameters which optimize the accuracy. Thus SVM would be a better classification method in this study, though KNN is a time saving algorithm.