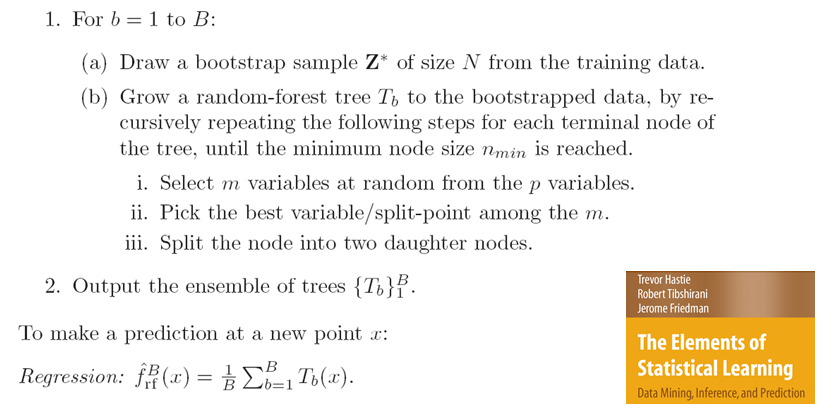
**10.4 - Random Forests**

Random forests are another bootstrap-based method for building trees. In addition to the use of bootstrap samples to build multiple trees that will then be averaged to obtain predictions, random forests also includes randomness in the tree building process for a given bootstrap sample. There are two packages (randomForest and party) to fit random forest models in R and there is also an implementation in JMP, though JMP refers to them at Bootstrap Forests to denote the dependence on bootstrap samples.

The algorithm for random forests from *Elements of Statistical Learning* is presented below.



The advantages of random forests are:

* It handles a very large number of input variables (e.g. QSAR or permeability data)
* It estimates the importance of input variables in the model.
* It returns proximities between cases, which can be useful for clustering, detecting outliers, and visualizing the data. (Unsupervised learning)
* Learning is faster than using the full set of potential predictors.
* Even though bootstrap sample trees will vary some, the predictions from the bootstrap trees will tend to be correlated. For example, the variables used to form the first split in bootstrap trees will tend to be the same and thus the subsequent trees will tend to be similar and thus correlated. Correlations between sums of random variables will inflate the variance, thus bagging will not in some cases decrease the variance part of MSE as much as we might think. In contrast, the trees in a random forest will vary even more than the bootstrap trees due to the random subsets of predictors being considered at each split. This will lead to trees that will tend to be less correlated and thus the benefit of averaging the random forest trees will be more pronounced than in bagging.

**Variable Importance**

To measure variable the importance do the following. For each bootstrap sample we first compute the Out-of-Bag (OOB) error rate, . Next we randomly permute the OOB values on the variable while leaving the data on all other variables unchanged. If is important, permuting its values will reduce our ability to predict the response successfully for all of the OOB observations. Then we make the predictions using the permuted values and all the other predictors unchanged to obtain , which should be larger than the error rate of the unaltered data. The raw score for can be computed by the difference between these two OOB error rates,

Finally, average the raw scores over all the *B* trees in the forest, i.e. expression below is computed

to obtain an overall measure of the importance of . This measure is called the *raw permutation accuracy importance score* for the variable. Assuming the *B* raw scores are independent from tree to tree, we can compute a straightforward estimate of the standard error by computing the standard deviation of the values. Dividing the average raw importance scores from each bootstrap by the standard error gives what is called the *mean decrease in accuracy* for the variable.

**Example 10.4: L.A. Basin Ozone Levels (cont’d)**

> oz.rf = randomForest(tupoz~.,data=Ozdata2,importance=T)

> oz.rf

Call:

randomForest(formula = tupoz ~ ., data = Ozdata2, importance = T)

Type of random forest: regression

Number of trees: 500

No. of variables tried at each split: 3

Mean of squared residuals: 0.05878031

% Var explained: 78.21

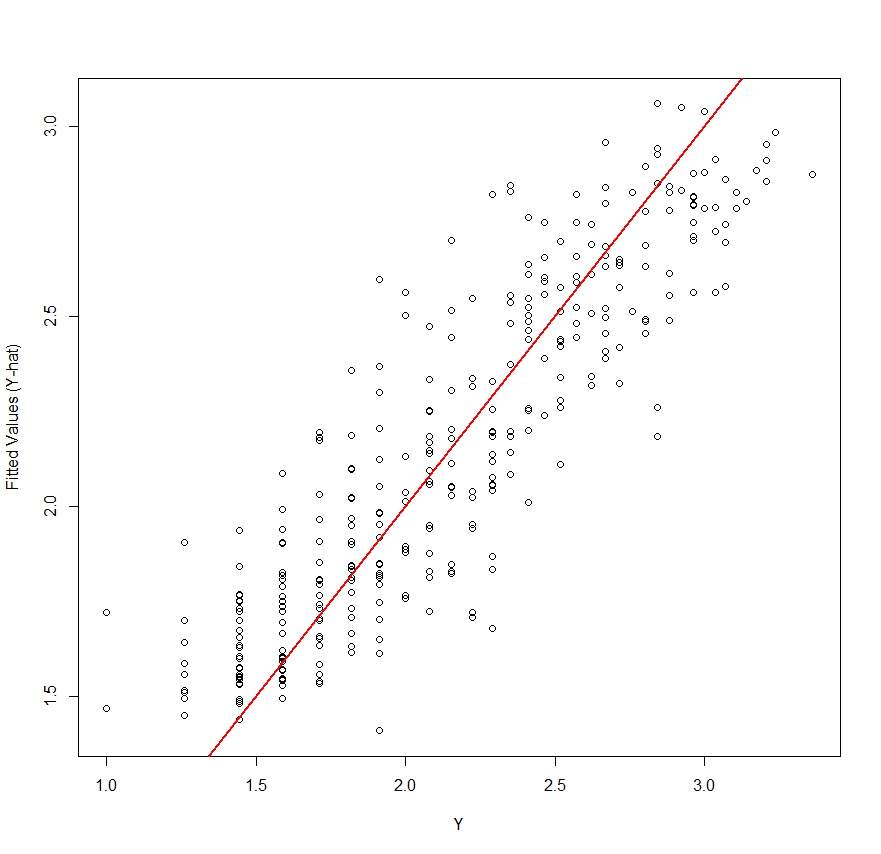
Additional settings in fitting a random forest using randomForest:

ntree – number of trees to grow, like B or nbagg in bagging

mtry – number of predictors to choose randomly for each split (default *= /3* for regression problems and for classification problems.)

nodesize – minimum size of the terminal nodes in terms of the number of observations contained in them, default is 1 for classification problems and 5 for regression problems. Larger values here speed of the fitting process because trees in the forest will not be as big.  
maxnodes – maximum number of terminal nodes a tree can have in the forest. Smaller values will speed up fitting.

> plot(tupoz,predict(oz.rf),xlab="Y",ylab="Fitted Values (Y-hat)")

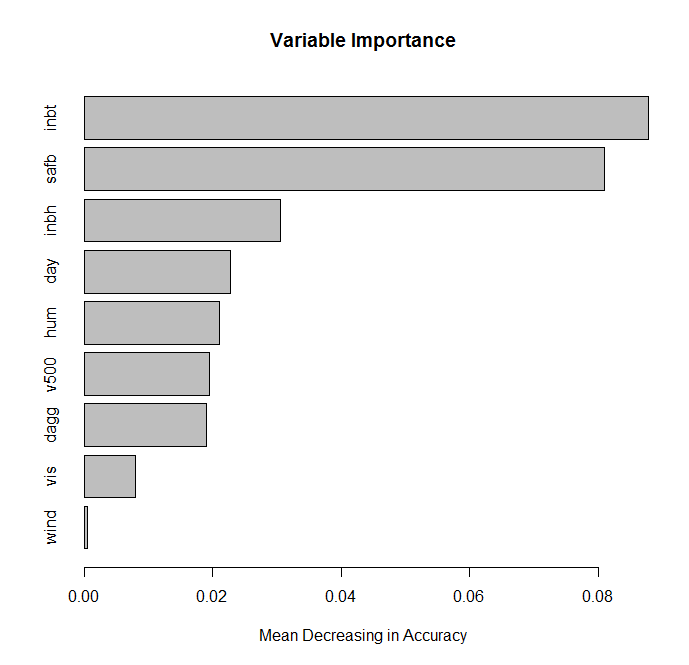


Below is a short function (rfimp) to display mean decrease in accuracy from a random forest fit, thus showing variable importance from top to bottom. Here we see that inversion base temperature (inbt) is the most important, closely followed by Sandburg Air Force Base temp (safb).

rfimp = function(rffit) {barplot(sort(rffit$importance[,1]),horiz=T,

xlab="Mean Decreasing in Accuracy",main="Variable Importance")

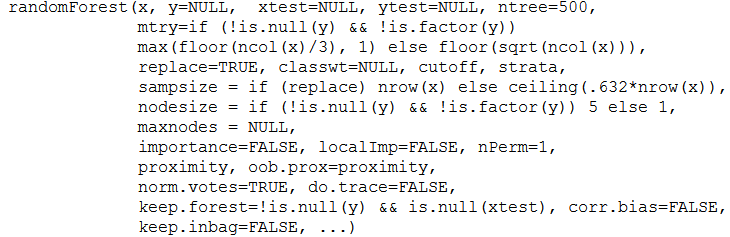
}



The random forest command with options is show below. It should be noted that x and y can be replaced by the usual formula building nomenclature, namely…

randomForest(y ~ . , data=Mydata, etc…)

From the R html help file:



Some important options have been highlighted in the generic function call above and they are summarized below:

* ntree = number of bootstrap trees in the forest
* mtry = number of variables to randomly pick at each stage ( in notes).
* maxnodes = maximum number of terminal nodes to use in the bootstrap trees
* importance = if T means variable importances will be computed.
* proximity = should proximity measures among rows be computed?
* do.trace = report OOB MSE and OOB R-square for each of the ntrees.

Estimating RMSEP for random forest models can be done using the errorest function in the ipred package (i.e. the bagging one) as show below. It is rather slow so doing more than 10 is not advisable. Each replicate does a 10-fold cross-validation B = 25 times per RMSEP estimate by default. It can be used with a variety of other modeling methods, see the errorest help file for examples.

> error.RF = numeric(10)

> for(i in 1:10) error.RF[i]= errorest(tupoz~.,data=Ozdata2,model=randomForest)$error   
  
> summary(error.RF)

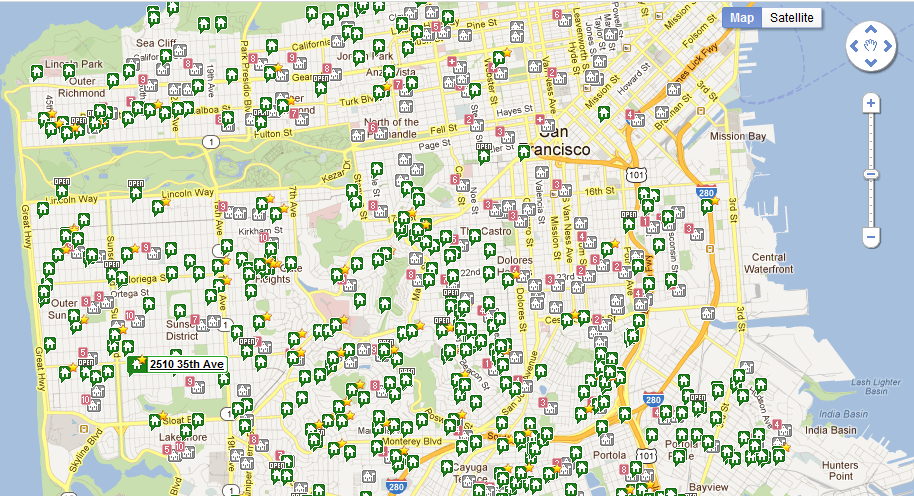
Min. 1st Qu. Median Mean 3rd Qu. Max.

0.2395 0.2424 0.2444 0.2448 0.2466 0.2506

We will consider a split-sample Monte Carlo approach below, which is preferable to the generic errorest function used above.

**Example 10.5: Predicting San Francisco List Prices of Single Family Homes**

Using [www.redfin.com](http://www.redfin.com) it is easy to create interesting data sets regarding the list price of homes and different characteristics of the home. The map below shows all the single-family homes listed in San Francisco just south of downtown and the Golden Gate Bridge. Once you drill down to this level of detail you can download the prices and home characteristics in an Excel file. From there it is easy to load the data in JMP, edit it, and then read it into R.



> SFhomes = read.table(file.choose(),header=T,sep=",")

> names(SFhomes)

[1] "ZIP" "ListPrice" "BEDS" "BATHS" "LOCATION" "SQFT"

[7] "YrBuilt" "Parking" "ParkType" "DaysMarket" "Reduction" "OrigPrice"

[13] "RecentSale" "LastSale" "LATITUDE" "LONGITUDE"  
  
> SFhomes = SFhomes[,-c(1,5,10,11,12,13,14)]  
> str(SFhomes)

'data.frame': 263 obs. of 9 variables:

$ ListPrice: int 749000 499900 579000 1295000 688000 224500 378000   
 140000 530000 399000 ...

$ BEDS : int 2 3 4 4 3 2 5 1 2 3 ...

$ BATHS : num 1 1 1 3.25 2 1 2 1 1 2 ...

$ SQFT : int 1150 1341 1429 2628 1889 995 1400 772 1240 1702 ...

$ YrBuilt : int 1931 1927 1937 1937 1939 1944 1923 1915 1925 1908 ...

$ ParkSpots: int 1 1 1 3 1 1 1 1 1 1 ...

$ Garage : Factor w/ 2 levels "Garage","No": 1 1 1 1 1 1 1 2 1 2 ...

$ LATITUDE : num 37.8 37.7 37.7 37.8 37.8 ...

$ LONGITUDE: num -122 -122 -122 -122 -122 ...

- attr(\*, "na.action")=Class 'omit' Named int [1:66] 7 11 23 30 32 34 36 43 55 62 ...

.. ..- attr(\*, "names")= chr [1:66] "7" "11" "23" "30" ...  
 **Note:** I omitted missing values by using the command na.omit.

> SFhomes = na.omit(SFhomes)

> head(SFhomes)

ListPrice BEDS BATHS SQFT YrBuilt ParkSpots Garage LATITUDE LONGITUDE

1 749000 2 1.00 1150 1931 1 Garage 37.76329 -122.4029

2 499900 3 1.00 1341 1927 1 Garage 37.72211 -122.4037

3 579000 4 1.00 1429 1937 1 Garage 37.72326 -122.4606

4 1295000 4 3.25 2628 1937 3 Garage 37.77747 -122.4502

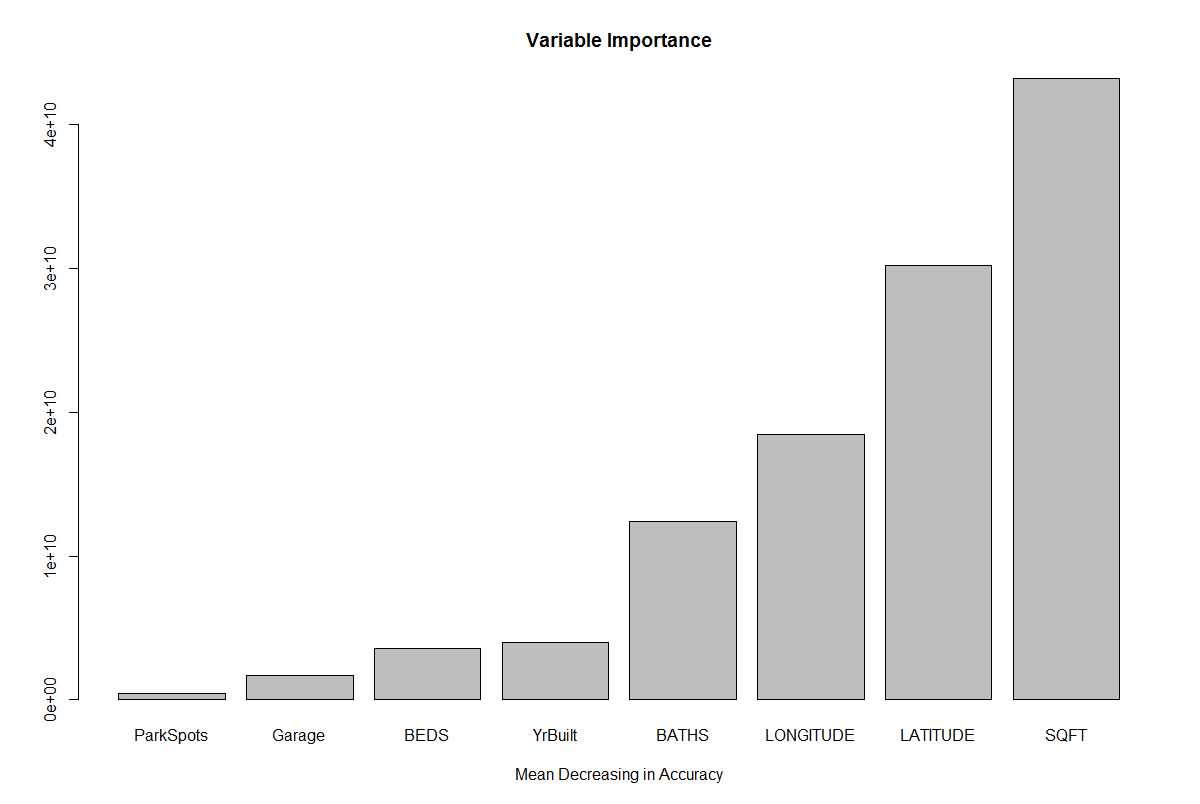
5 688000 3 2.00 1889 1939 1 Garage 37.75287 -122.4857

6 224500 2 1.00 995 1944 1 Garage 37.72778 -122.3838

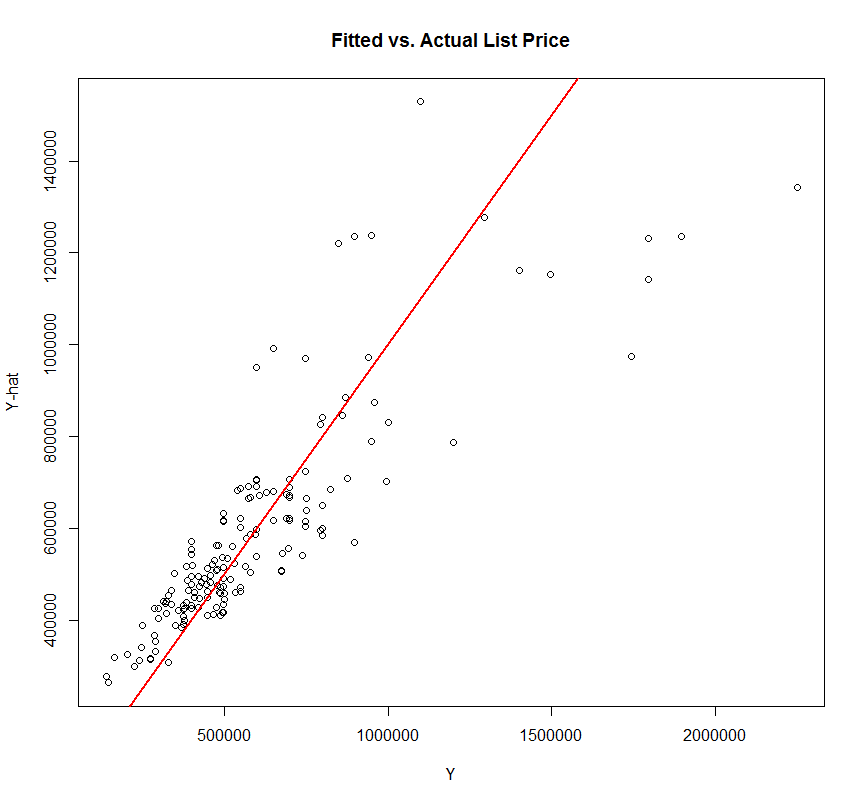
We will now develop a random forest model for the list price of the home as a function of the number of bedrooms, number of bathrooms, square footage, year built, number of parking spots, garage (Garage or No), latitude, and longitude.

> sf.rf = randomForest(ListPrice~.,data=SFhomes,importance=T)

> rfimp(sf.rf,horiz=F)

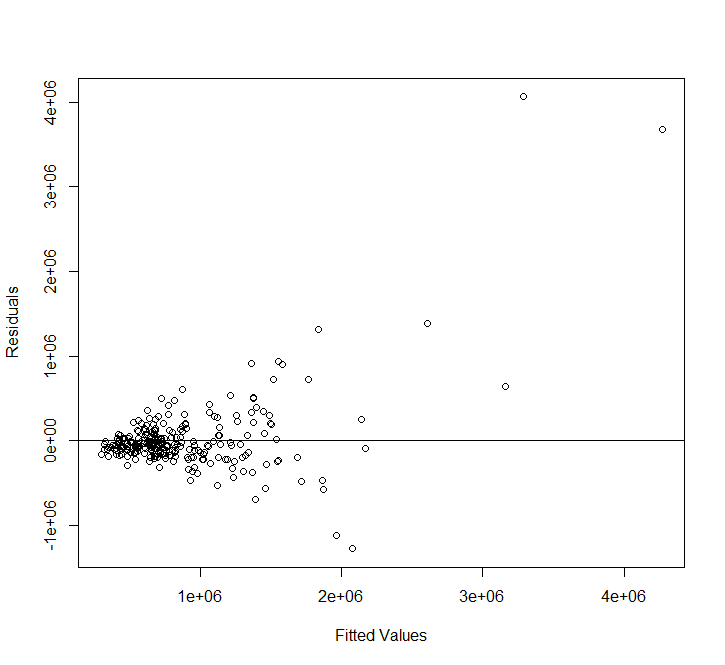
  
> plot(SFhomes$ListPrice,predict(sf.rf),xlab="Y",ylab="Y-hat",  
 main="Fitted vs. Actual List Price")

> abline(0,1,col=”red”,lwd=2)



> plot(predict(sf.rf),SFhomes$ListPrice - predict(sf.rf),xlab="Fitted Values",ylab="Residuals")

> abline(h=0,col=”red”,lwd=2)

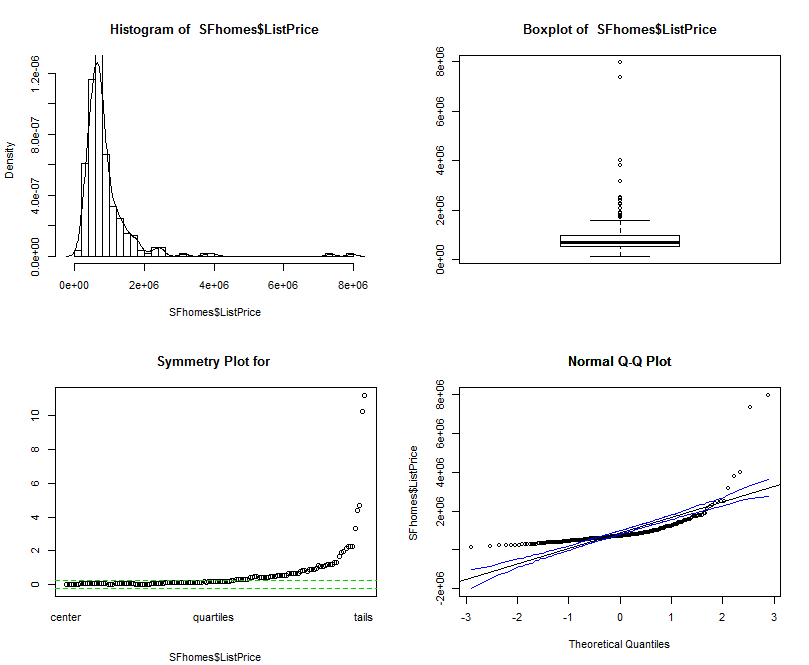
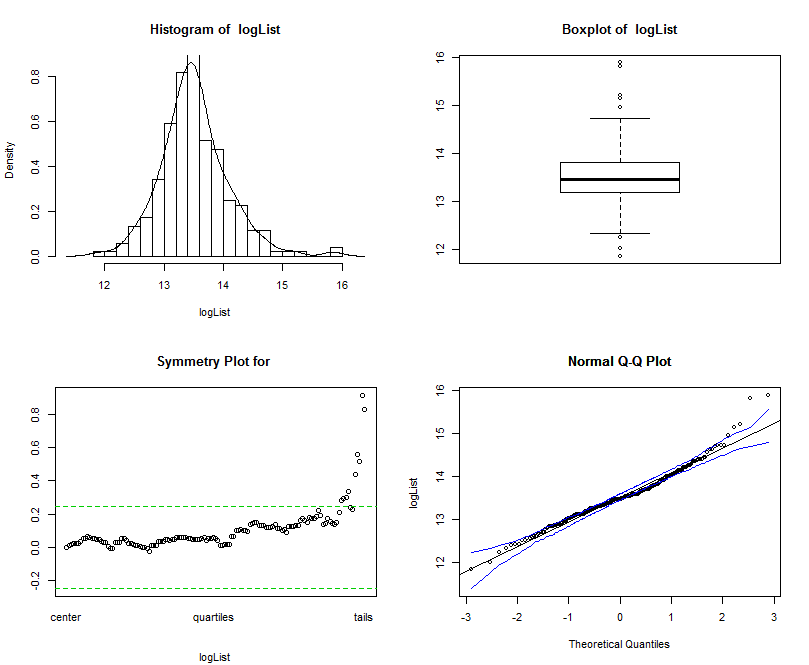


This does not look good, there is severe heteroscedasticity and outliers.

> Statplot(SFhomes$ListPrice)

> logList = log(SFhomes$ListPrice)

> Statplot(logList)

> SFhomes.log = data.frame(logList,SFhomes[,-1])

> names(SFhomes.log)

[1] "logList" "BEDS" "BATHS" "SQFT" "YrBuilt" "ParkSpots" "Garage"

[8] "LATITUDE" "LONGITUDE"

We now resume the model building process using the log of the list price as the response. To develop models we consider different choices for (m) the number predictors chosen in each random subset.

> attributes(sf.rf)

$names

[1] "call" "type" "predicted" "mse" "rsq" "oob.times"

[7] "importance" "importanceSD" "localImportance" "proximity" "ntree" "mtry"

[13] "forest" "coefs" "y" "test" "inbag" "terms"

$class

[1] "randomForest.formula" "randomForest"

> sf.rf$mtry

[1] 2

***mtry = 2***

> myforest = function(formula,data){randomForest(formula,data,mtry=2)}

> error.RF = numeric(10)

> for (i in 1:10) error.RF[i] = errorest(logList~.,data=SFhomes.log,model=myforest)$error

> mean(error.RF)

[1] 0.2214782

> summary(error.RF)

Min. 1st Qu. Median Mean 3rd Qu. Max.

0.2176 0.2193 0.2218 0.2215 0.2225 0.2270

***mtry = 3***

> myforest = function(formula,data){randomForest(formula,data,mtry=3)}

> error.RF = numeric(10)

> for (i in 1:10) error.RF[i] = errorest(logList~.,data=SFhomes.log,model=myforest)$error

> mean(error.RF)

[1] 0.2063236

> summary(error.RF)

Min. 1st Qu. Median Mean 3rd Qu. Max.

0.2025 0.2054 0.2071 0.2063 0.2079 0.2087

***mtry = 4***  
> myforest = function(formula,data){randomForest(formula,data,mtry=4)}

> error.RF = numeric(10)

> for (i in 1:10) error.RF[i] = errorest(logList~.,data=SFhomes.log,model=myforest)$error

> mean(error.RF)

[1] 0.202133

> summary(error.RF)

Min. 1st Qu. Median Mean 3rd Qu. Max.

0.1953 0.2005 0.2018 0.2021 0.2042 0.2110

***mtry = 5***

> myforest = function(formula,data){randomForest(formula,data,mtry=5)}

> error.RF = numeric(10)

> for (i in 1:10) error.RF[i] = errorest(logList~.,data=SFhomes.log,model=myforest)$error

> mean(error.RF)

[1] 0.2014575

> summary(error.RF)

Min. 1st Qu. Median Mean 3rd Qu. Max.

0.1933 0.1987 0.2015 0.2015 0.2040 0.2094

***mtry = 6***

> myforest = function(formula,data){randomForest(formula,data,mtry=6)}

> error.RF = numeric(10)

> for (i in 1:10) error.RF[i] = errorest(logList~.,data=SFhomes.log,model=myforest)$error

> mean(error.RF)

[1] 0.1979085

> summary(error.RF)

Min. 1st Qu. Median Mean 3rd Qu. Max.

0.1943 0.1972 0.1978 0.1979 0.1986 0.2015

***mtry = 7***  
> myforest = function(formula,data){randomForest(formula,data,mtry=7)}

> error.RF = numeric(10)

> for (i in 1:10) error.RF[i] = errorest(logList~.,data=SFhomes.log,model=myforest)$error

> mean(error.RF)

[1] 0.2034014

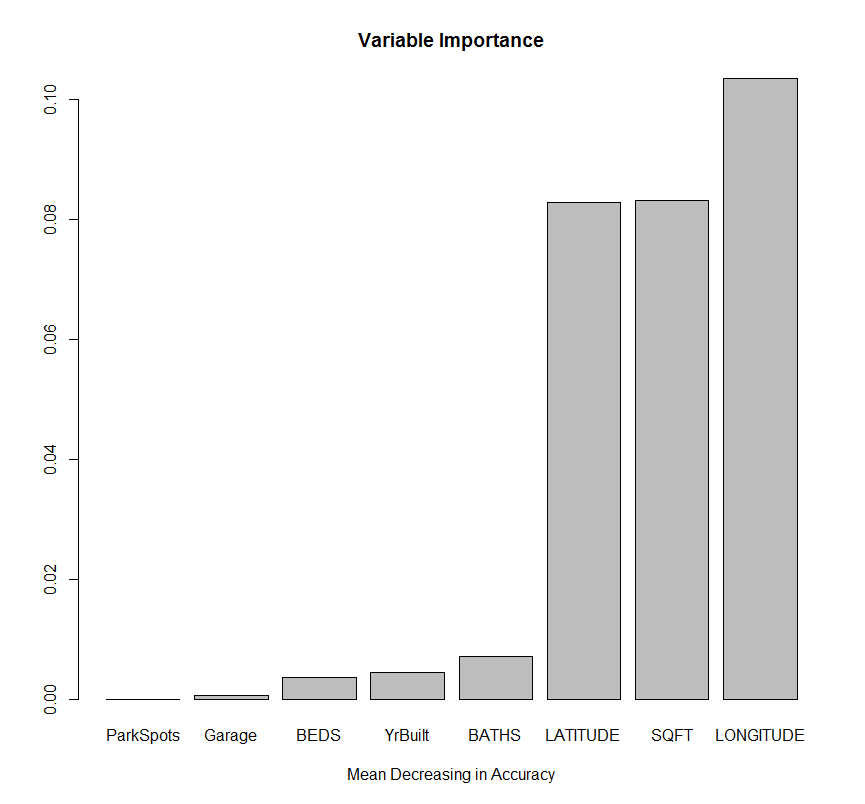
> summary(error.RF)

Min. 1st Qu. Median Mean 3rd Qu. Max.

0.1919 0.1967 0.2031 0.2034 0.2072 0.2190

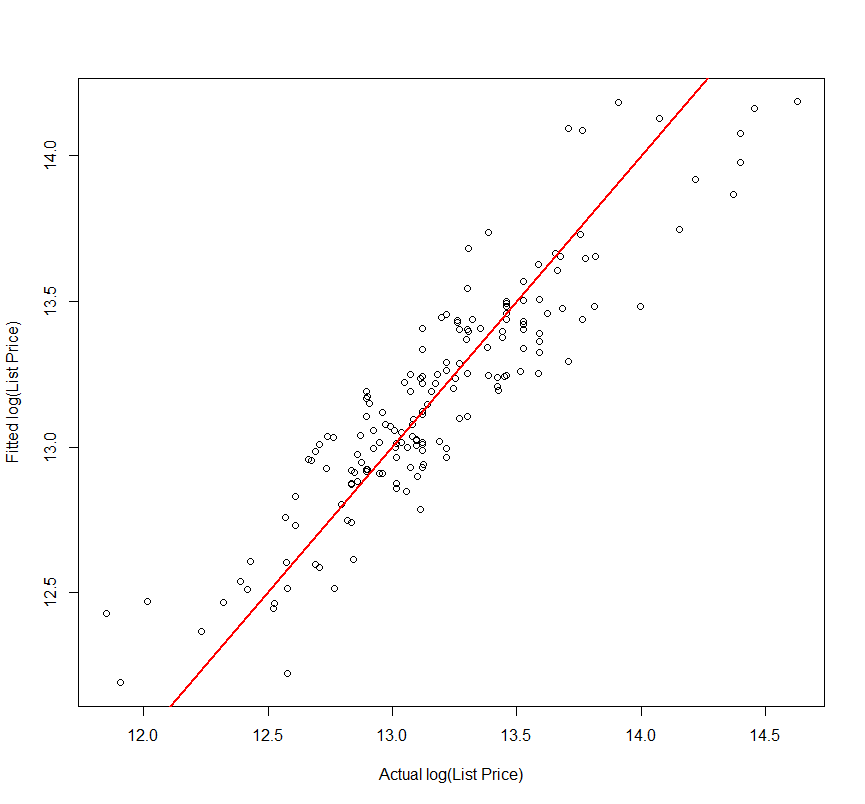
> sf.final = randomForest(logList~.,data=SFhomes.log,importance=T,mtry=6)

> rfimp(sf.final,horiz=F)



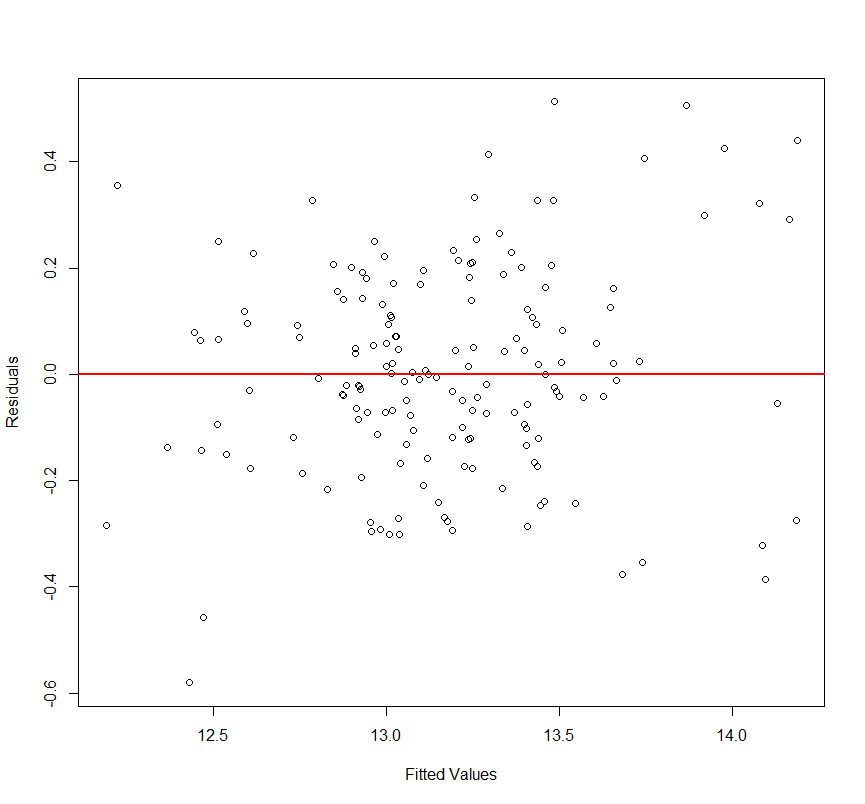
> plot(logList,predict(sf.final),xlab="log(List Price)",ylab="Fitted log(List Price)")  
> cor(SFhomes.log$logList,predict(sf.final))^2 🡨

[1] 0.821722



> plot(predict(sf.final),logList - predict(sf.final),  
 xlab="Fitted Values",ylab="Residuals")

> abline(h=0)



As with earlier methods, it is not hard to write our own MCCV function. This function actually runs faster than the using the errorest function from ipred and is estimating the prediction errors using random pairs of training/validation sets.

> rf.sscv = function(fit,data,p=.667,B=100,mtry=fit$mtry,ntree=fit$ntree) {

MSE = rep(0,B)

MAE = rep(0,B)

MAPE = rep(0,B)

y = fit$y

n = nrow(data)

ss <- floor(n\*p)

for (i in 1:B) {

sam = sample(1:n,ss,replace=F)

fit2 = randomForest(formula(fit),data=data[sam,],mtry=mtry,ntree=ntree)

ynew = predict(fit2,newdata=data[-sam,])

MSE[i] = mean((y[-sam]-ynew)^2)

MAE[i] = mean(abs(y[-sam]-ynew))

MAPE[i] = mean((abs(y[-sam]-ynew)/y[-sam]))

}

RMSEP = sqrt(mean(MSE))

MAEP = mean(MAE)

MAPEP = mean(MAPE)

cat("RMSEP\n")

cat("===============\n")

cat(RMSEP,"\n\n")

cat("MAE\n")

cat("===============\n")

cat(MAEP,"\n\n")

cat("MAPE\n")

cat("===============\n")

cat(MAPEP,"\n\n")

temp = data.frame(MSEP=MSE,MAEP=MAE,MAPEP=MAPE)

return(temp)

}

This function also allows you to experiment with different values for (mtry) & (ntree).  
  
> results = rf.sscv(sf.final,SFhomes.log,mtry=2)

RMSEP

===============

0.2308656

MAE

===============

0.1784265

MAPE

===============

0.01351952

> results = rf.sscv(sf.final,SFhomes.log,mtry=3)

RMSEP

===============

0.2184711

MAE

===============

0.1688001

MAPE

===============

0.01279754

> results = rf.sscv(sf.final,SFhomes.log,mtry=4)

RMSEP

===============

0.2172428

MAE

===============

0.1685127

MAPE

===============

0.01272567

> results = rf.sscv(sf.final,SFhomes.log,mtry=5) 🡨 mtry = 5 appears optimal

RMSEP

===============

0.2070135

MAE

===============

0.1615488

MAPE

===============

0.0122046

> results = rf.sscv(sf.final,SFhomes.log,mtry=6)

RMSEP

===============

0.2079716

MAE

===============

0.1616868

MAPE

===============

0.01222138

We can also examine the impact of increasing the number of trees in the random forest. Below we consider increasing the number of trees beyond the default ntree=500.

> results = rf.sscv(sf.final,SFhomes.log,mtry=5,ntree=500,B=250)

RMSEP

===============

0.2125801

MAE

===============

0.1640355

MAPE

===============

0.01240571

> results = rf.sscv(sf.final,SFhomes.log,mtry=5,ntree=750,B=250) 🡨 slight improvement on all

RMSEP

===============

0.2106317

MAE

===============

0.163657

MAPE

===============

0.01238443

> results = rf.sscv(sf.final,SFhomes.log,mtry=5,ntree=1000,B=250)🡨 worse on all

RMSEP

===============

0.2121543

MAE

===============

0.1645516

MAPE

===============

0.01245262

**Partial Plots to Visualize Predictor Effects**

> names(SFhomes.log)

[1] "logList" "BEDS" "BATHS" "SQFT" "YrBuilt"   
[8] "ParkSpots" "Garage" "LATITUDE" "LONGITUDE"

> partialPlot(sf.final,SFhomes.log,SQFT)

> partialPlot(sf.final,SFhomes.log,LONGITUDE)

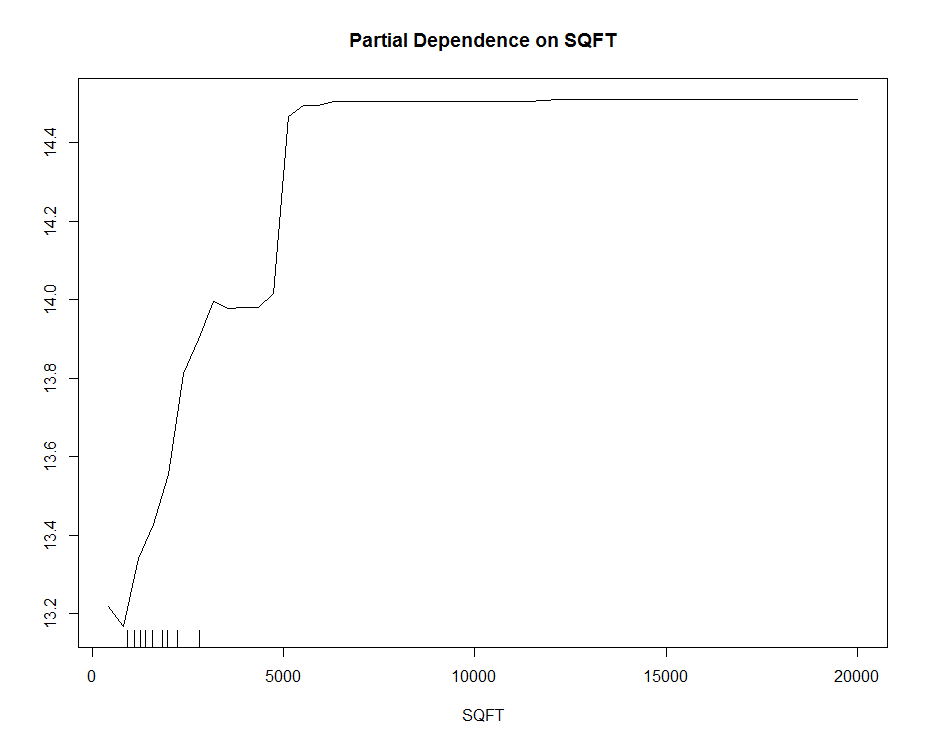
> partialPlot(sf.final,SFhomes.log,LATITUDE)

> partialPlot(sf.final,SFhomes.log,BEDS)

> partialPlot(sf.final,SFhomes.log,BATHS)

> partialPlot(sf.final,SFhomes.log,YrBuilt)

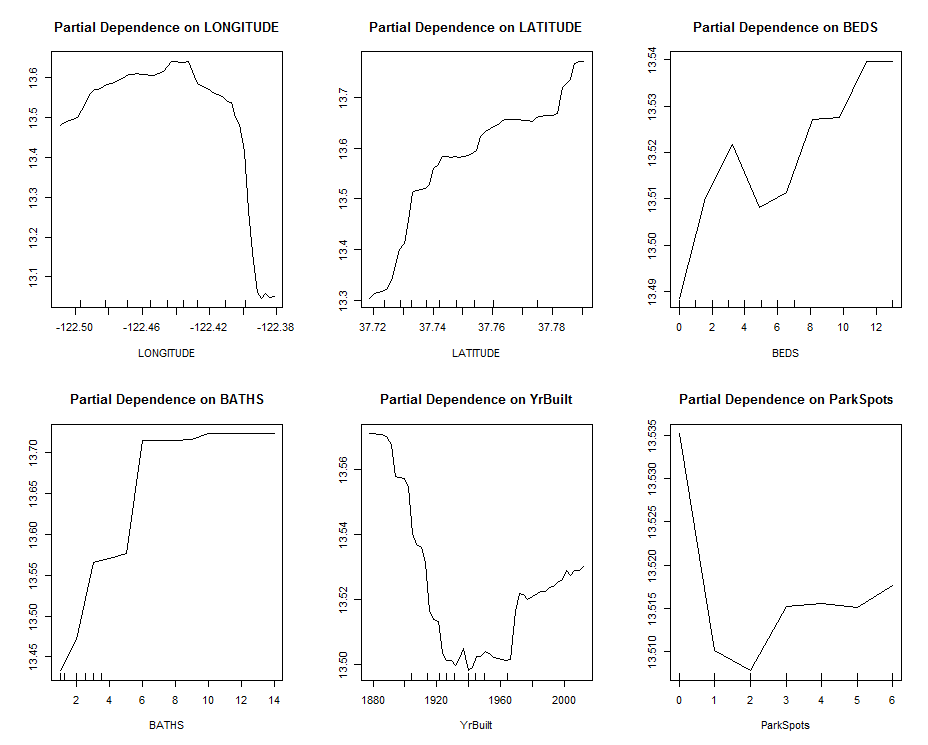
> partialPlot(sf.final,SFhomes.log,ParkSpots)

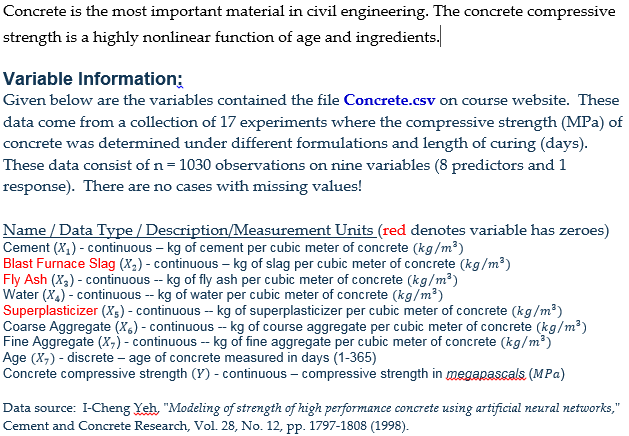


> par(mfrow=c(2,3))

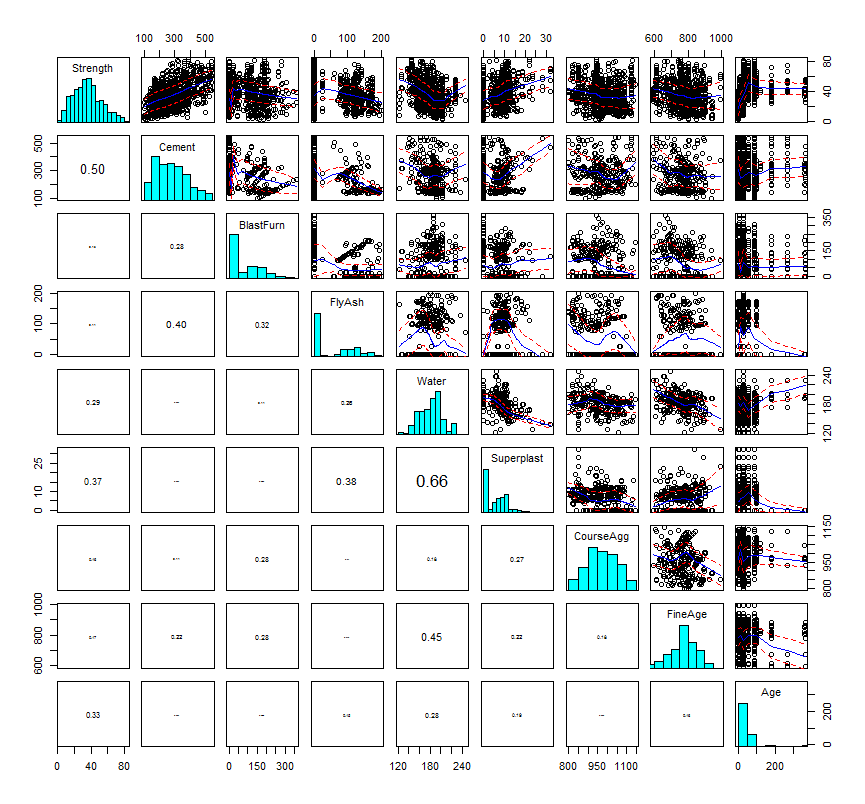
Sets up plot consisting of 2 rows and 3 columns of plots.

> par(mfrow=c(1,1)) 🡨 restore default



**Example 10.6 – Concrete Strength**

We will now consider fitting a random forest to the concrete strength data from Assignment 2.



> dim(Concrete)

[1] 1030 9

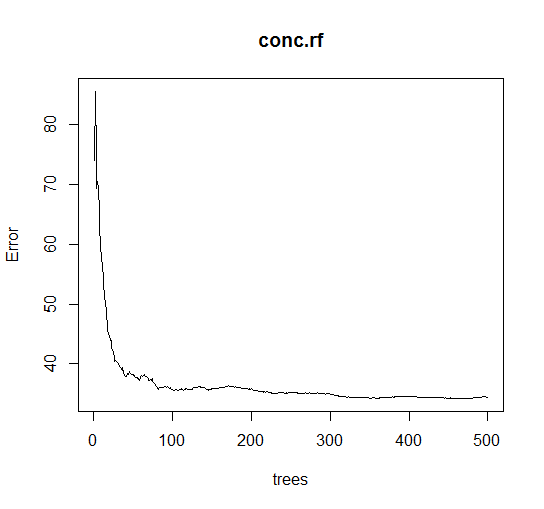
> sam = sample(1:1030,size=floor(1030\*.6667),replace=F)

> Concrete.train = Concrete[sam,] 🡨 form training dataset

> Concrete.valid = Concrete[-sam,] 🡨 form validation dataset

> conc.rf = randomForest(Strength~.,data=Concrete.train

> plot(conc.rf)



> conc.rf = randomForest(Strength~.,data=Concrete.train,ntree=100)

> results = rf.sscv(conc.rf,Concrete.train)

RMSEP

===============

6.620847

MAE

===============

5.019157

MAPE

===============

0.1883238  
  
> conc.rf = randomForest(Strength~.,data=Concrete.train,ntree=350)  
> results = rf.sscv(conc.rf,Concrete.train)

RMSEP

===============

6.600895

MAE

===============

4.99328

MAPE

===============

0.1881917

> results = rf.sscv(conc.rf,Concrete.train,mtry=2,ntree=350)

RMSEP

===============

6.508817

MAE

===============

4.926981

MAPE

===============

0.1843898

> results = rf.sscv(conc.rf,Concrete.train,mtry=3,ntree=350)

RMSEP

===============

6.079046

MAE

===============

4.521543

MAPE

===============

0.163179

> results = rf.sscv(conc.rf,Concrete.train,mtry=4,ntree=350)

RMSEP

===============

5.966902

MAE

===============

4.361965

MAPE

===============

0.153899

> results = rf.sscv(conc.rf,Concrete.train,mtry=5,ntree=350)

RMSEP

===============

5.937936

MAE

===============

4.319596

MAPE

===============

0.1514776

> results = rf.sscv(conc.rf,Concrete.train,mtry=6,ntree=350)

RMSEP

===============

5.909492

MAE

“Final” Model

===============

4.27028

MAPE

===============

0.148233

> results = rf.sscv(conc.rf,Concrete.train,mtry=7,ntree=350)

RMSEP

===============

5.942979

MAE

===============

4.299485

MAPE

===============

0.1476808

> conc.final = randomForest(Strength~.,data=Concrete.train,mtry=6,ntree=350)

> ypred = predict(conc.final,newdata=Concrete.valid)

> PredAcc(ypred,Concrete.valid$Strength)

RMSEP

===============

4.925258

MAE

===============

3.579164

MAPE

===============

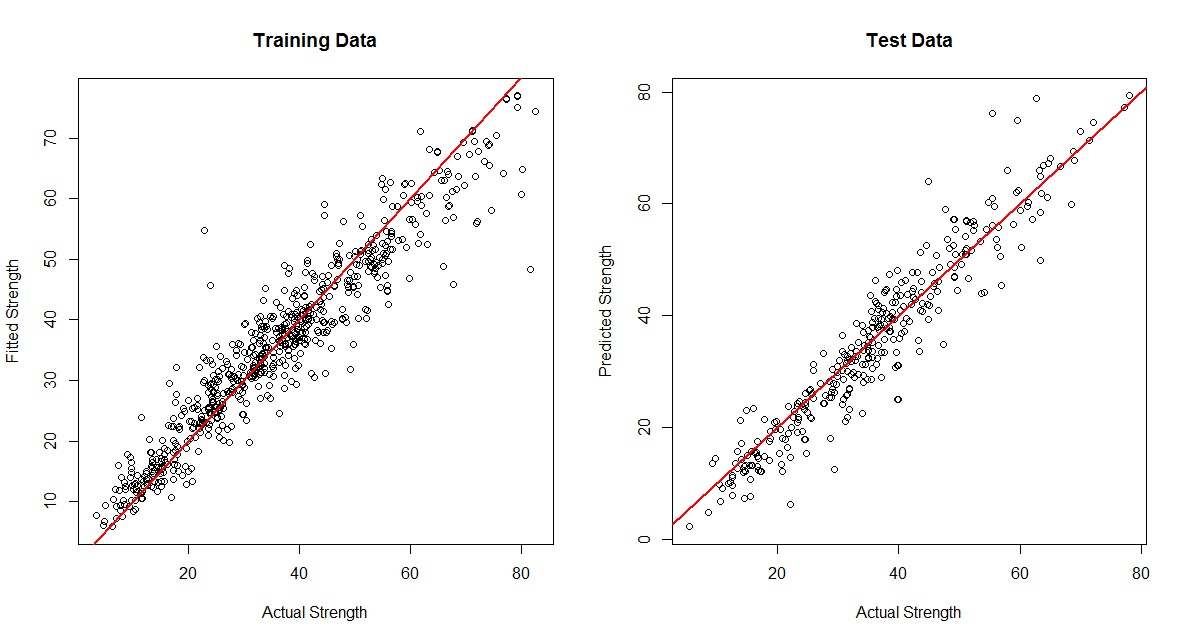
11.59418

> par(mfrow=c(1,2))  
> plot(Concrete.train$Strength,predict(conc.final),xlab="Actual Strength",ylab="Fitted Strength",main="Training Data")

> abline(0,1,col="red",lwd=2)

> plot(ypred,Concrete.valid$Strength,xlab="Actual Strength",ylab="Predicted Strength",main="Test Data")

> abline(0,1,col="red",lwd=2)  
> par(mfrow=c(1,1))



Partial Effect Plots for Concrete Strength Random Forest

> par(mfrow=c(3,3))

> partialPlot(conc.final,Concrete.train,Cement)

> partialPlot(conc.final,Concrete.train,BlastFurn)

> partialPlot(conc.final,Concrete.train,FlyAsh)

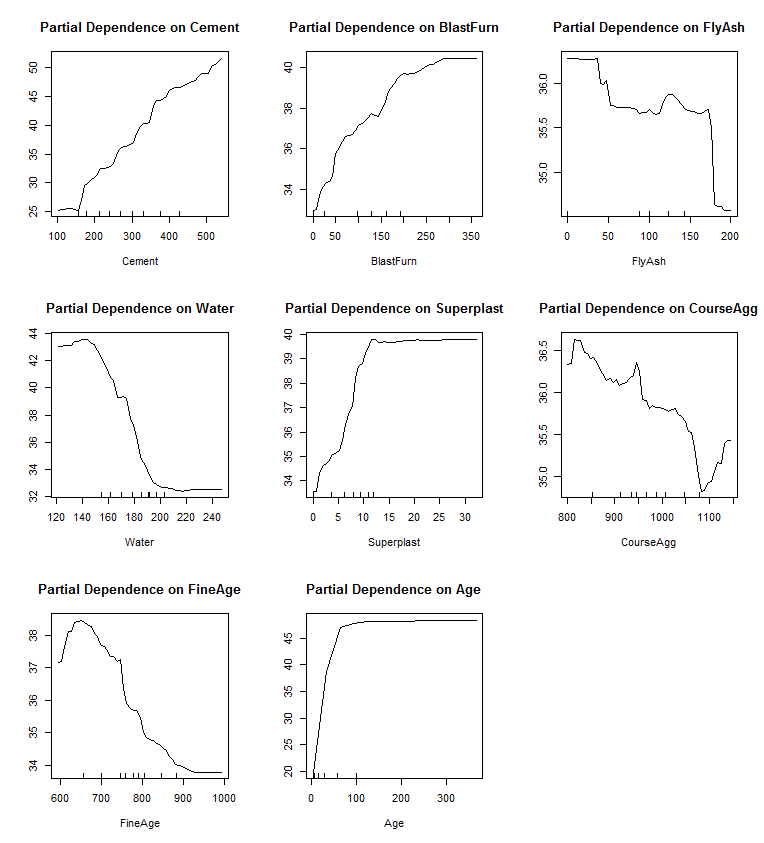
> partialPlot(conc.final,Concrete.train,Water)

> partialPlot(conc.final,Concrete.train,Superplast)

> partialPlot(conc.final,Concrete.train,CourseAgg)

> partialPlot(conc.final,Concrete.train,FineAge)

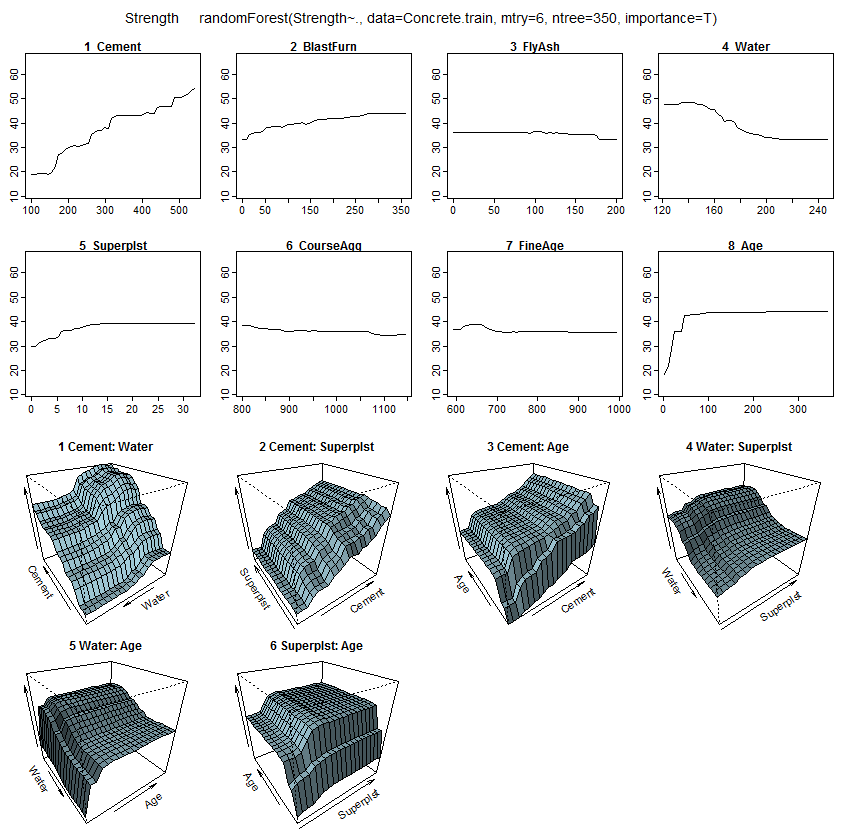
> partialPlot(conc.final,Concrete.train,Age)



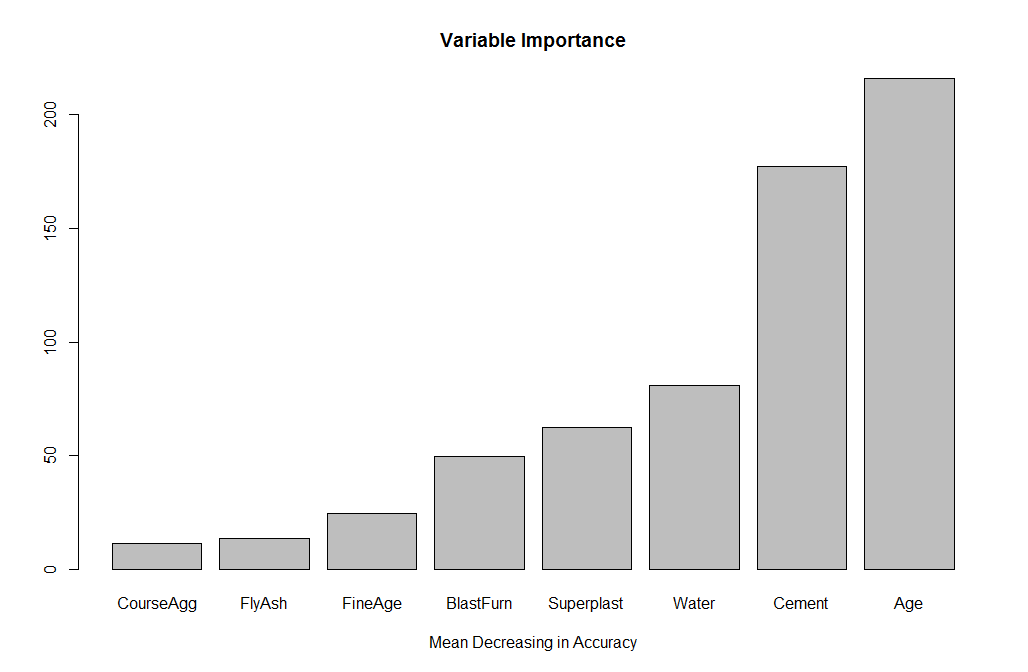
Which predictors matter most?   
We can actually make that determination from the plots above, how?   
  
Do any of these plots look troubling in terms of the fit? Does this suggest overfitting?

The plotmo package can be used to examine views of the fitted model using the function of the same name applied to conc.final.

> plotmo(conc.final)



> rfimp(conc.final)



**10.5 – Extremely Randomized Trees**

Similar to a Random Forest a new ensemble method has been proposed known as Extremely Randomized Trees (Geurts, Ernst, & Wehenkel (2006) . To introduce **more variation** into the ensemble, we will change how we build trees as follows.

Each simple tree will be built with the following criteria:

1. All the data available in the training set is used when building each stumpy tree.
2. To form the root node or any subsequent node, the best split is determined by searching in a subset of randomly selected features of size .   
   The split of each selected predictor/feature is chosen at random. This is sharp contract to the random forest which chooses the best split on the randomly selected variable.
3. The maximum size of the trees generated this way is typically 5 terminal nodes/leaves for regression problems and 2 for classification problems.
4. The final prediction is the average prediction from a sequence of extremely randomized trees.

Notice that both the predictors/features and splits are selected at random; hence, “Extremely Randomized Tree”. Since splits are chosen at random for each feature in the fitted tree, it’s less computationally expensive than a Random Forest.

The package extraTrees contains functions for training and predicting from extremely randomized trees. This package also requires the package rJava which in turn requires an installation of Java on your computer as well. The is a link to guide you through the rJava installation process, including downloading and installing the latest release of Java.  
  
<https://cimentadaj.github.io/blog/2018-05-25-installing-rjava-on-windows-10/installing-rjava-on-windows-10/>

Once Java has been installed you can set up a path to where Java is stored on your  
computer then load both libraries.  
> Sys.setenv(JAVA\_HOME="C:/Program Files/Java/jdk-13.0.2/")

> library(rJava)

> library(extraTrees)

**Example 10.5 (cont’d) – Selling Price of San Francisco Homes**

> names(SFhomes)

[1] "ListPrice" "BEDS" "BATHS" "SQFT" "YrBuilt" "ParkSpots" "Garage" "LATITUDE"

[9] "LONGITUDE"

Note: ListPrice has already been transformed the log-scale.

For purposes of model development we will form a training and validation set. > dim(SFhomes)

> sam = sample(1:nrows(SFhomes),213,replace=F)

> SF.train = SFhomes[sam,]

> SF.test = SFhomes[-sam,]

First we need to create a response vector y and a predictor matrix X based on the

training data.