**4 – Tree-based Regression Models**

**4.1 - Introduction and Motivation**Tree-based modelling began as a primarily exploratory technique for uncovering structure in data. Specifically, the technique is useful for classification and regression problems where one has a set of predictor variables and a single response . When is nominal or ordinal (i.e non-numeric) then classification rules are determined from the given set of predictors which can be any mixture of data types, e.g.,

When the response is numeric, then regression tree rules for prediction are of the form:

The first tree models were called CART models, (**C**lassification **A**nd **R**egression **T**rees) and were fit using an algorithm called ***recursive partitioning*** which is described in the case of regression trees in the next section. As computing power has increased there has been work on improving the performance of tree-based models. We will certainly examine these improvements as these tend to produce among the best, if not the best, off-the-shelf predictive models. Thus tree-based models have become quite popular.

Some additional reasons for their popularity:

**1.** In certain applications, especially where the set of predictors contains a mix of numeric variables and factors, tree-based models are sometimes easier to interpret and discuss than linear models.

**2.** Tree-based models are **invariant to monotone transformations of predictor variables**, thus the precise form in which these appear in the model is irrelevant. As we have seen in several earlier covered methods and associated examples this is a particularly appealing property!!

**3.** Tree-based models are more adept at capturing non-additive behavior; the standard MLR model does not allow interactions between variables unless they are pre-specified and of a particular multiplicative form. We saw that in the MLR models we developed for the diamond price data. Tree-models because of the way they are constructed naturally have interactions, typically lots of them!

4. Tree-based models can be “combined” to produce very powerful prediction machines.

**4.2 - Regression Trees and Recursive Partitioning**

The form of the fitted surface or smooth obtained from a regression tree is

where the are constants and the are regions defined a series of binary splits. If all the predictors are numeric these regions form a set of disjoint hyper-rectangles with sides parallel to the axes such that

Regardless of how the neighborhoods are defined if we use the least squares criterion for each region

the best estimator of the response, , is just the average of the in the region , i.e.

.

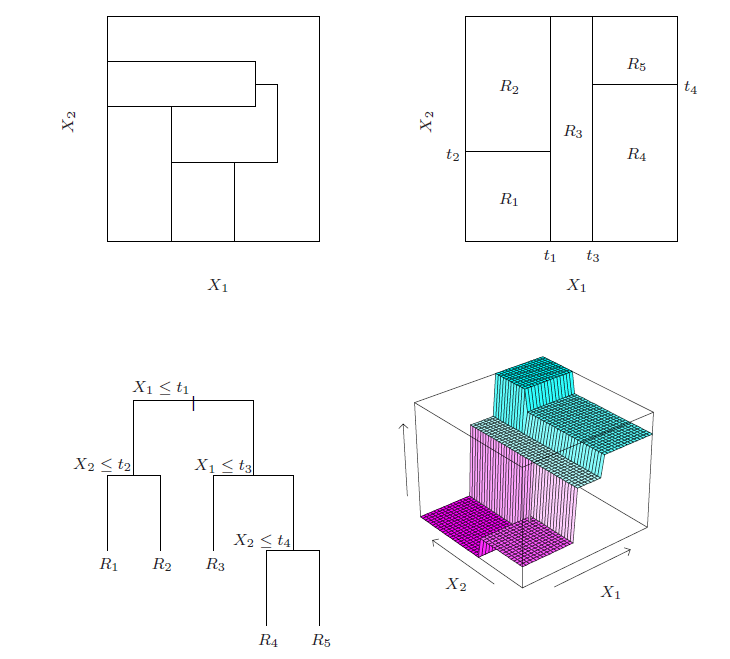
Thus to obtain a fitted regression tree model we need to somehow obtain the neighborhoods This is accomplished by an algorithm called *recursive partitioning*, see Breiman et al. (1984). We present the basic idea below though an example for the case where the number of neighborhoods and the number of predictor variables The task of determining neighborhoods is solved by determining a *split coordinate or variate* , i.e. which variable to split on, and *split point* . A split coordinate and split point define the rectangles as

The residual sum of squares (RSS) for a split determined by is

The goal at any given stage is to find the pair such that is minimal or the overall RSS is maximally reduced. This may seem overwhelming, however this only requires examining at most splits for each variable because the points in a neighborhood only change when the split point crosses an observed value. If we wish to split into three neighborhoods, i.e. split or after the first split, we have possibilities for the first split and possibilities for the second split, given the first split. In total we have operations to find the best splits for neighborhoods. In general for neighborhoods we have,

possibilities if all predictors are numeric! This gets too big for an exhaustive search, therefore we use the technique for recursively. This is the basic idea of recursive partitioning. One starts with the first split and obtains as explained above. This split stays fixed and the same splitting procedure is applied recursively to the two regions . Thus it is a greedy algorithm as previous choices are not reconsidered once they are made. This procedure is then repeated until we reach some stopping criterion such as the nodes become homogenous or contain very few observations. The rpart function uses two such stopping criteria. A node will not be split if it contains fewer minsplit observations (default =20). Additionally we can specify the minimum number of observations in terminal node by specifying a value for minbucket (default = ).

The figures below are from pg. 306 of *Elements of Statistical Learning* (Hastie, et al.) show a hypothetical tree fit based on two numeric predictors .



**Example 4.1: Ozone Levels in L.A. Basin**  
Let's examine these ideas using ozone pollution data for the Los Angeles Basin. For simplicity we consider the case where . Here we will develop a regression tree using rpart for predicting upper ozone concentration using the temperature at Sandburg Air Force Base (safb) and inversion base height (inbh).

> Ozdata = read.table(file.choose(),header=T,sep=”,”)

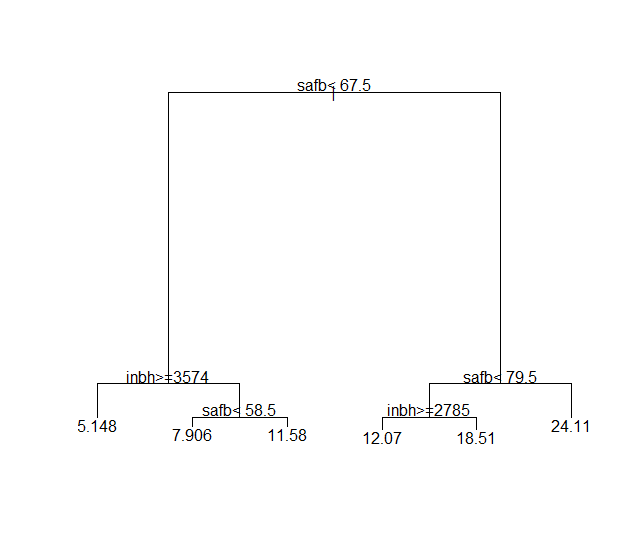
> library(rpart)

> oz.rpart <- rpart(upoz ~ inbh + safb,data=Ozdata)

> summary(oz.rpart)

> plot(oz.rpart)

> text(oz.rpart)



> post(oz.rpart,"Regression Tree for Upper Ozone Concentration")

**Plot the fitted surface**

> x1 = seq(min(Ozdata$inbh),max(Ozdata$inbh),length=100)

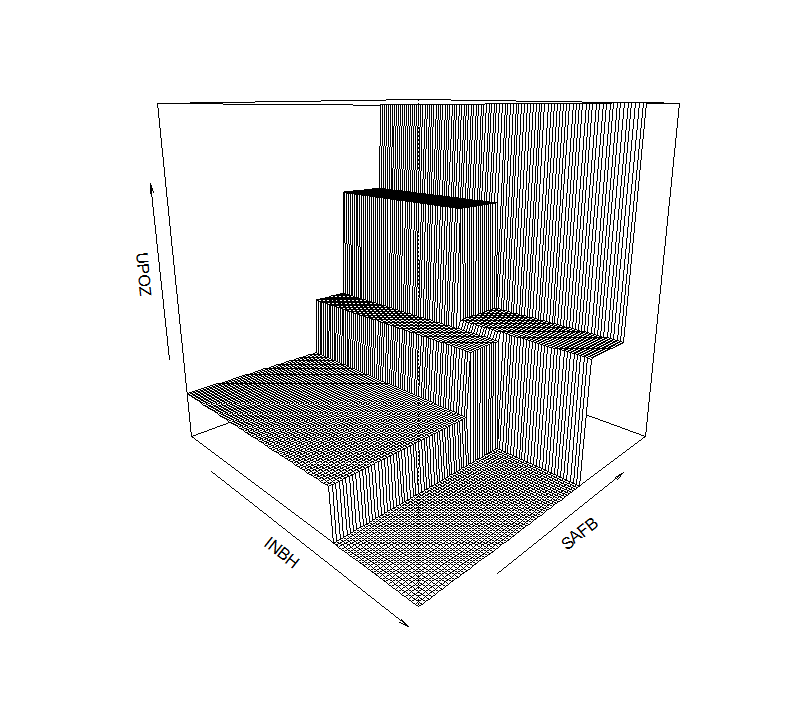
> x2 = seq(min(Ozdata$safb),max(Ozdata$safb),length=100)

> x = expand.grid(inbh=x1,safb=x2)

> ypred = predict(oz.rpart,newdata=x)

> persp(x1,x2,z=matrix(ypred,100,100),theta=45,xlab="INBH",

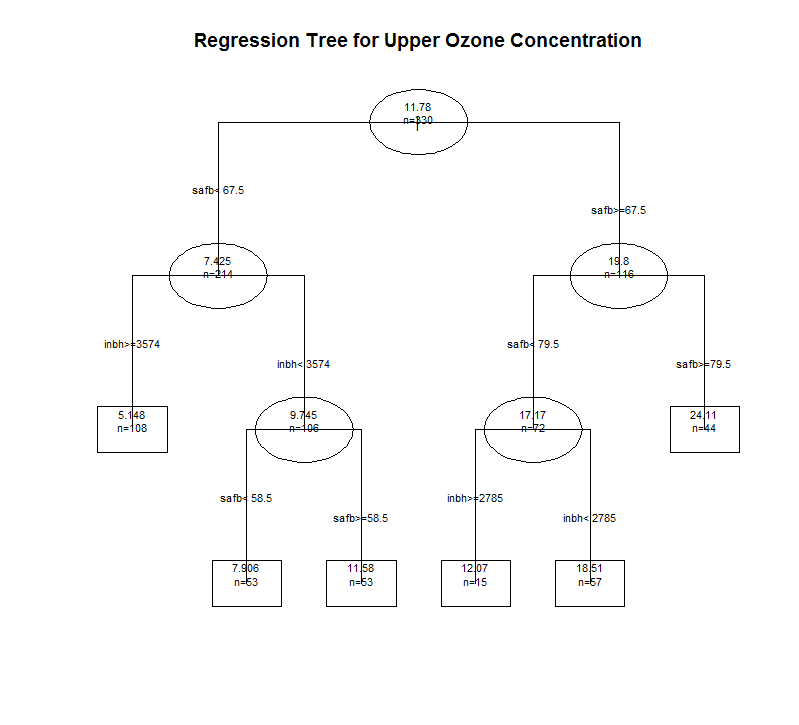
ylab="SAFB",zlab="UPOZ")

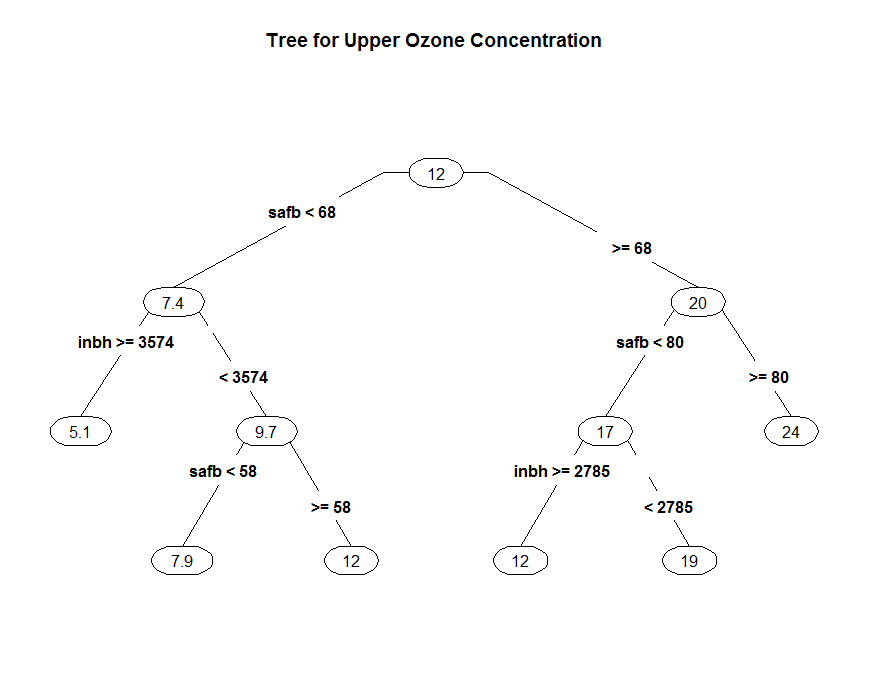


> plot(oz.rpart,uniform=T,branch=1,compress=T,margin=0.05,cex=.5)

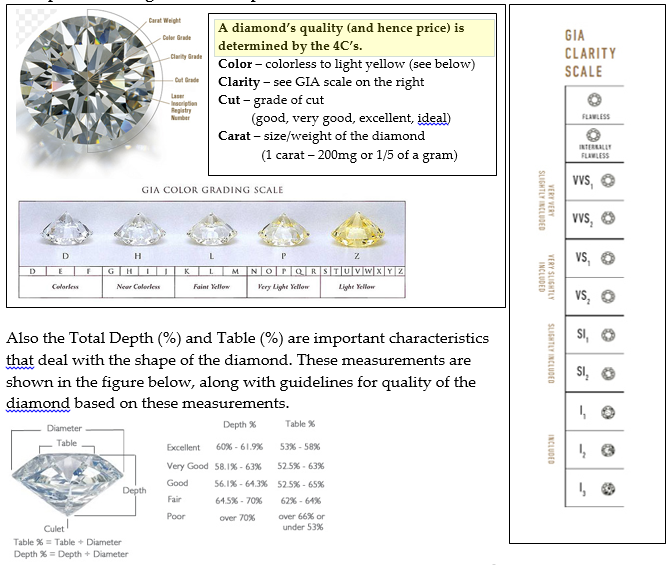
> text(oz.rpart,all=T,use.n=T,fancy=T,cex=.7)

> title(main="Regression Tree for Upper Ozone Concentration")



> prp(oz.rpart,main="Tree for Upper Ozone Concentration",type=4)  


**Example 4.2: Diamond Prices**



To begin we will load libraries for fitting and plotting trees and we again form training, validation, and test data sets based on Test variable in the Diamonds data set.

> library(rpart)

> library(rpart.plot)

> Diamonds = read.table(file.chooseIO,header=T,sep=”,”)

> names(Diamonds)

[1] "Price" "Carats" "Color" "Clarity" "Depth" "Table" "Cut" "TDdiff"

[9] "TDratio" "Test"

> table(Diamonds$Test)

0 1 2

1613 539 538

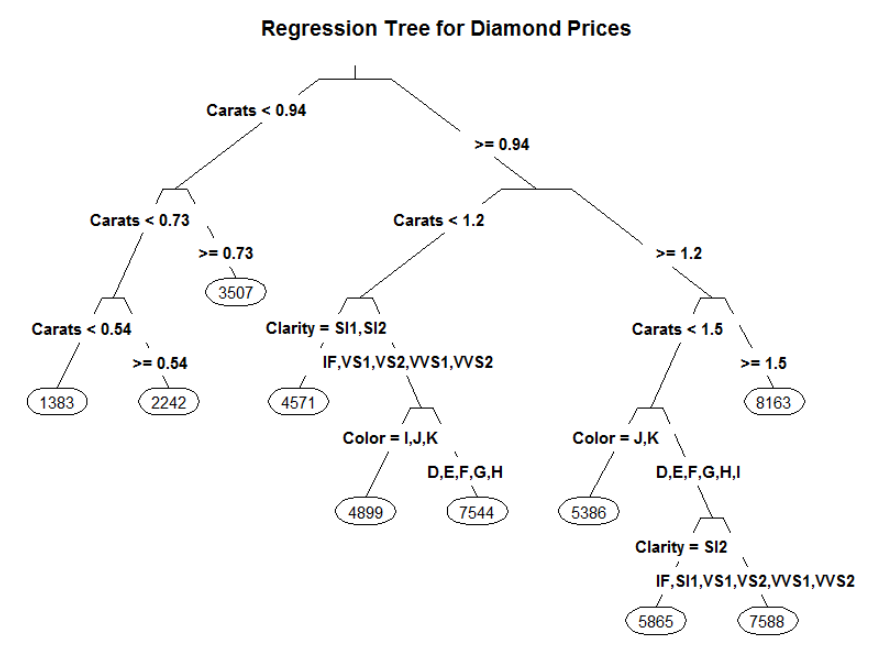
> diam.train = Diamonds[Diamonds$Test==0,-10]

> diam.valid = Diamonds[Diamonds$Test==1,-10]

> diam.test = Diamonds[Diamonds$Test==2,-10]

Fit a preliminary tree model with the default settings (i.e. ***tuning parameters***).  
  
> tree1 = rpart(Price~.,data=diam.train)

> prp(tree1,type=3,main=”Regression Tree for Diamond Prices”)



> summary(tree1)

Call:

rpart(formula = Price ~ ., data = diam.train)

n= 1613

CP nsplit rel error xerror xstd

1 0.62641917 0 1.0000000 1.0013290 0.030195774

2 0.05266618 1 0.3735808 0.3803366 0.014950445

3 0.05202894 2 0.3209147 0.2763722 0.012340517

4 0.05119268 3 0.2688857 0.2705374 0.012251444

5 0.02757021 4 0.2176930 0.2290717 0.010087800

6 0.01653198 5 0.1901228 0.1987306 0.009088610

7 0.01341838 6 0.1735908 0.1854338 0.008643600

8 0.01037896 7 0.1601725 0.1748261 0.008552891

9 0.01034023 8 0.1497935 0.1748261 0.008552891

10 0.01000000 9 0.1394533 0.1719344 0.008476092

Variable importance

Carats Color Clarity Depth TDdiff Table

61 19 17 1 1 1

Node number 1: 1613 observations, complexity param=0.6264192

mean=3934.667, MSE=5707547

left son=2 (898 obs) right son=3 (715 obs)

Primary splits:

Carats < 0.945 to the left, improve=0.626419200, (0 missing)

Color splits as LLLRRRRR, improve=0.061002450, (0 missing)

Clarity splits as LRRRRLR, improve=0.044188650, (0 missing)

Depth < 62.75 to the right, improve=0.003738784, (0 missing)

TDratio < 1.066896 to the left, improve=0.002396600, (0 missing)

Surrogate splits:

Color splits as LLLRRRRR, agree=0.691, adj=0.303, (0 split)

Clarity splits as LRRLLLL, agree=0.663, adj=0.241, (0 split)

Table < 60.5 to the left, agree=0.561, adj=0.010, (0 split)

Depth < 58.5 to the right, agree=0.560, adj=0.007, (0 split)

TDdiff < -9.85 to the right, agree=0.559, adj=0.006, (0 split)

Node number 2: 898 observations, complexity param=0.05266618

mean=2247.445, MSE=1295045

left son=4 (670 obs) right son=5 (228 obs)

Primary splits:

Carats < 0.735 to the left, improve=0.41692160, (0 missing)

Cut splits as LLRL, improve=0.01407325, (0 missing)

Clarity splits as LLLLLLR, improve=0.01039031, (0 missing)

TDratio < 0.9390058 to the right, improve=0.01037088, (0 missing)

TDdiff < -3.85 to the right, improve=0.01006658, (0 missing)

Surrogate splits:

Color splits as LLLLLLLR, agree=0.754, adj=0.031, (0 split)

TDdiff < -9.55 to the right, agree=0.748, adj=0.009, (0 split)

TDratio < 0.8623332 to the right, agree=0.748, adj=0.009, (0 split)

Node number 3: 715 observations, complexity param=0.05202894

mean=6053.723, MSE=3183688

left son=6 (435 obs) right son=7 (280 obs)

Primary splits:

Carats < 1.175 to the left, improve=0.21042260, (0 missing)

Clarity splits as RLLRRRR, improve=0.18706960, (0 missing)

Color splits as RRRRLLLL, improve=0.08024459, (0 missing)

Cut splits as RLRL, improve=0.02706446, (0 missing)

Table < 58.5 to the right, improve=0.01169681, (0 missing)

Surrogate splits:

Color splits as LLLLLLRL, agree=0.635, adj=0.068, (0 split)

Depth < 58.05 to the right, agree=0.613, adj=0.011, (0 split)

TDdiff < 4.05 to the left, agree=0.613, adj=0.011, (0 split)

TDratio < 1.066558 to the left, agree=0.613, adj=0.011, (0 split)

Table < 53.5 to the right, agree=0.610, adj=0.004, (0 split)

Node number 4: 670 observations, complexity param=0.01341838

mean=1818.799, MSE=529807.9

left son=8 (330 obs) right son=9 (340 obs)

Primary splits:

Carats < 0.535 to the left, improve=0.34800920, (0 missing)

Clarity splits as RLLLLRR, improve=0.02850527, (0 missing)

Cut splits as LLRL, improve=0.01868145, (0 missing)

Color splits as RRLLLLLL, improve=0.01772654, (0 missing)

Depth < 62.75 to the right, improve=0.01086100, (0 missing)

Surrogate splits:

Clarity splits as LRRLRLL, agree=0.696, adj=0.382, (0 split)

Color splits as LLLRRRRR, agree=0.601, adj=0.191, (0 split)

TDdiff < -5.55 to the left, agree=0.536, adj=0.058, (0 split)

TDratio < 0.9002426 to the left, agree=0.536, adj=0.058, (0 split)

Depth < 62.55 to the right, agree=0.533, adj=0.052, (0 split)

Node number 5: 228 observations

mean=3507.066, MSE=1417194

Node number 6: 435 observations, complexity param=0.05119268

mean=5397.055, MSE=2586490

left son=12 (267 obs) right son=13 (168 obs)

Primary splits:

Clarity splits as RLLRRRR, improve=0.41888210, (0 missing)

Color splits as RRRRLLLL, improve=0.28377810, (0 missing)

Cut splits as RLRL, improve=0.02613429, (0 missing)

Depth < 62.75 to the right, improve=0.01424792, (0 missing)

Table < 58.5 to the right, improve=0.01094202, (0 missing)

Surrogate splits:

Color splits as LLRLLLLR, agree=0.634, adj=0.054, (0 split)

Carats < 0.955 to the right, agree=0.618, adj=0.012, (0 split)

TDdiff < -8.2 to the right, agree=0.616, adj=0.006, (0 split)

TDratio < 0.8702553 to the right, agree=0.616, adj=0.006, (0 split)

Node number 7: 280 observations, complexity param=0.01653198

mean=7073.904, MSE=2400789

left son=14 (192 obs) right son=15 (88 obs)

Primary splits:

Carats < 1.495 to the left, improve=0.22641050, (0 missing)

Color splits as RRRRRRLL, improve=0.13375920, (0 missing)

Clarity splits as RRLRRRR, improve=0.04623199, (0 missing)

TDdiff < 1.35 to the left, improve=0.02091462, (0 missing)

TDratio < 1.022955 to the left, improve=0.02091462, (0 missing)

Surrogate splits:

Depth < 62.85 to the left, agree=0.700, adj=0.045, (0 split)

TDdiff < -8.05 to the right, agree=0.696, adj=0.034, (0 split)

TDratio < 0.8737384 to the right, agree=0.696, adj=0.034, (0 split)

Table < 63.5 to the left, agree=0.693, adj=0.023, (0 split)

Node number 8: 330 observations

mean=1382.948, MSE=143196.6

Node number 9: 340 observations

mean=2241.829, MSE=541715.1

Node number 12: 267 observations

mean=4571.397, MSE=697765.4

Node number 13: 168 observations, complexity param=0.02757021

mean=6709.262, MSE=2782891

left son=26 (53 obs) right son=27 (115 obs)

Primary splits:

Color splits as RRRRRLLL, improve=0.54289830, (0 missing)

Depth < 61.35 to the right, improve=0.03768856, (0 missing)

Clarity splits as R--RLRR, improve=0.02350434, (0 missing)

Cut splits as RLRL, improve=0.01326847, (0 missing)

Carats < 1.095 to the right, improve=0.01222508, (0 missing)

Surrogate splits:

Carats < 1.075 to the right, agree=0.708, adj=0.075, (0 split)

TDdiff < 3.35 to the right, agree=0.696, adj=0.038, (0 split)

TDratio < 1.056732 to the right, agree=0.696, adj=0.038, (0 split)

Depth < 58.6 to the left, agree=0.690, adj=0.019, (0 split)

Node number 14: 192 observations, complexity param=0.01037896

mean=6574.771, MSE=2006238

left son=28 (50 obs) right son=29 (142 obs)

Primary splits:

Color splits as RRRRRRLL, improve=0.24805840, (0 missing)

Clarity splits as RRLRRRR, improve=0.19357610, (0 missing)

Cut splits as RLRL, improve=0.07074066, (0 missing)

Depth < 62.75 to the right, improve=0.03307478, (0 missing)

TDdiff < -2.55 to the right, improve=0.02199928, (0 missing)

Surrogate splits:

Carats < 1.365 to the right, agree=0.755, adj=0.06, (0 split)

Depth < 59.05 to the left, agree=0.750, adj=0.04, (0 split)

Node number 15: 88 observations

mean=8162.92, MSE=1532107

Node number 26: 53 observations

mean=4898.679, MSE=865713.8

Node number 27: 115 observations

mean=7543.704, MSE=1459339

Node number 28: 50 observations

mean=5385.92, MSE=929906.1

Node number 29: 142 observations, complexity param=0.01034023

mean=6993.38, MSE=1712330

left son=58 (49 obs) right son=59 (93 obs)

Primary splits:

Clarity splits as RRLRRRR, improve=0.39150570, (0 missing)

Color splits as RRLRLL--, improve=0.09441569, (0 missing)

Depth < 62.55 to the right, improve=0.07979308, (0 missing)

Cut splits as RLRL, improve=0.04604033, (0 missing)

Carats < 1.245 to the left, improve=0.03852810, (0 missing)

Surrogate splits:

Depth < 59.75 to the left, agree=0.683, adj=0.082, (0 split)

TDdiff < -7.45 to the left, agree=0.683, adj=0.082, (0 split)

TDratio < 0.8807051 to the left, agree=0.683, adj=0.082, (0 split)

Carats < 1.365 to the right, agree=0.676, adj=0.061, (0 split)

Color splits as LLLRRR--, agree=0.676, adj=0.061, (0 split)

Node number 58: 49 observations

mean=5865.388, MSE=539400.8

Node number 59: 93 observations

mean=7587.699, MSE=1306724

Viewing the tree is certainly easier in graphical form!! We can plot the fitted values vs. the actual prices and the residuals vs. fitted values as we did in MLR.

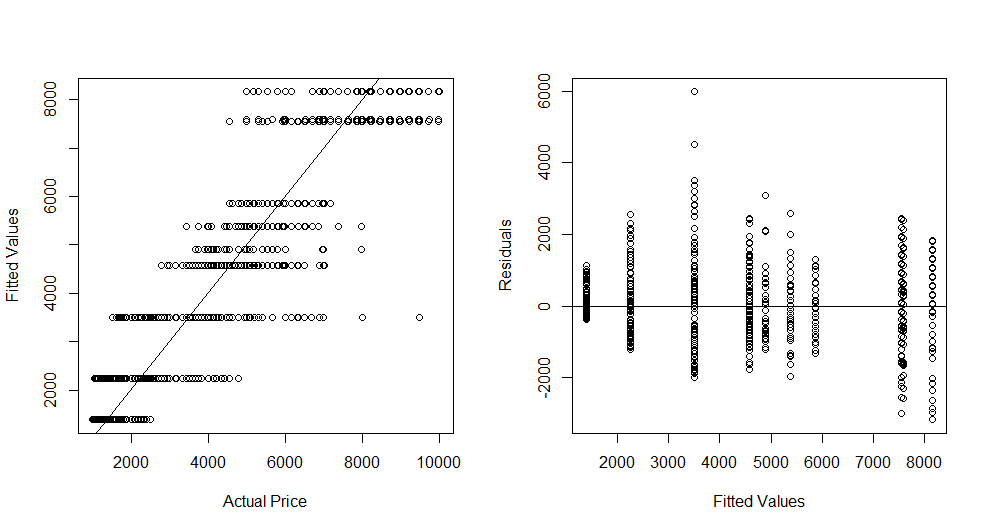
> par(mfrow=c(1,2))

> plot(diam.train$Price,predict(tree1),xlab="Actual Price",ylab="Fitted Values")

> abline(0,1)

> plot(predict(tree1),resid(tree1),xlab="Fitted Values",ylab="Residuals")

> abline(h=0)



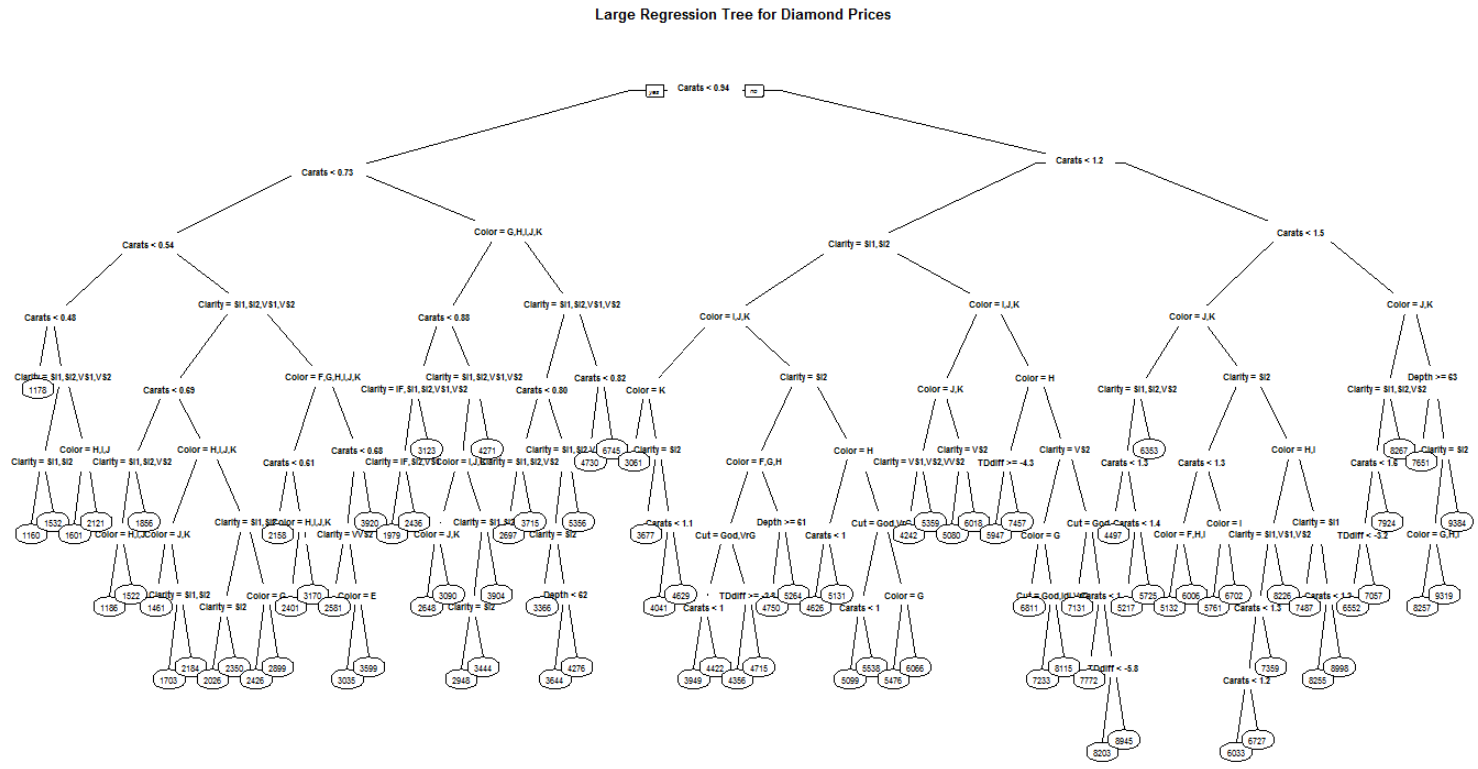
> par(mfrow=c(1,1))

This tree seems too simplistic and underfits, thus we should force the algorithm to consider larger trees. This most easily done by making the complexity parameter (cp) smaller. The complexity parameter penalizes the RSS for tree based upon the number of terminal nodes according to the formula below:

where,

> tree2 = rpart(Price~.,data=diam.train,control=rpart.control(cp=.0001))

> prp(tree2,main=”Large Regression Tree for Diamond Prices”,cex=0.5)



YIKES!

Below are a plot of the fitted vs. actual response values and a plot of the residuals vs. fitted values. This model does a much better job of matching the actual response values, possibly overfitting, and nonconstant variation is the response is evident in the residuals.

> par(mfrow=c(1,2))

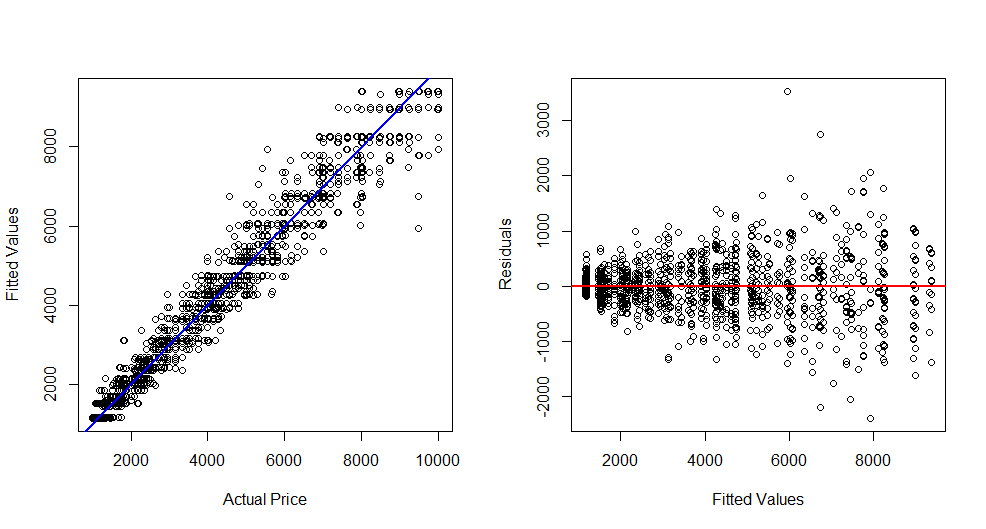
> plot(diam.train$Price,predict(tree2),xlab="Actual Price",

ylab="Fitted Price")

> abline(0,1,lwd=2,col=”blue”)

> plot(predict(tree2),resid(tree2),xlab=”Fitted Values”,ylab=”Residuals”)

> abline(h=0,lwd=2,col=”red”)



> par(mfrow=c(1,1))

We can compare these two trees by comparing their predictive performance on the validation cases. The prediction accuracy can be measured using several metrics and the function below will compute RMSEP, MAE, and MAPE given the actual and predicted response values.

PredAcc = function(y,ypred){

RMSEP = sqrt(mean((y-ypred)^2))

MAE = mean(abs(y-ypred))

MAPE = mean(abs(y-ypred)/y)\*100

cat("RMSEP\n")

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

cat(RMSEP,"\n\n")

cat("MAE\n")

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

cat(MAE,"\n\n")

cat("MAPE\n")

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

cat(MAPE,"\n\n")

return(data.frame(RMSEP=RMSEP,MAE=MAE,MAPE=MAPE))

}

Prediction accuracy of the base tree model (cp = .01, the default setting)

> ypred = predict(tree1,newdata=diam.valid)  
> PredAcc(diam.valid$Price,ypred)

RMSEP

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

1000.356

MAE

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

742.0048

MAPE

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

20.95349

RMSEP MAE MAPE

1 1000.356 742.0048 20.95349

Prediction accuracy of the larger tree model (cp = .0001)  
> ypred2 = predict(tree2,newdata=diam.valid)

> PredAcc(diam.valid$Price,ypred2)

RMSEP

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

673.6085

MAE

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

444.1392

MAPE

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

11.17317

RMSEP MAE MAPE

1 673.6085 444.1392 11.17317

> bestMLR = lm(log(Price)~poly(Carats,3)+Clarity\*Color + Cut + TDdiff + TDratio,data=diam.train)

> ypredlog = predict(best.MLR,newdata=diam.valid)

> ypredMLR = exp(ypredlog)

> PredAcc(diam.valid$Price,ypredMLR)

RMSEP

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

493.787

MAE

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

333.2352

MAPE

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

8.160273

Tree models are invariant to predictor transformations but **not** the response, thus we can consider using as the response as we did in MLR models fit to these data.

> tree3 = rpart(log(Price)~.,data=diam.train,cp=.0001)

> ypredlog = predict(tree3,newdata=diam.valid)

> ypred3 = exp(ypredlog)

> PredAcc(diam.valid$Price,ypred3)

RMSEP

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

662.5886

MAE

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

428.2071

MAPE

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

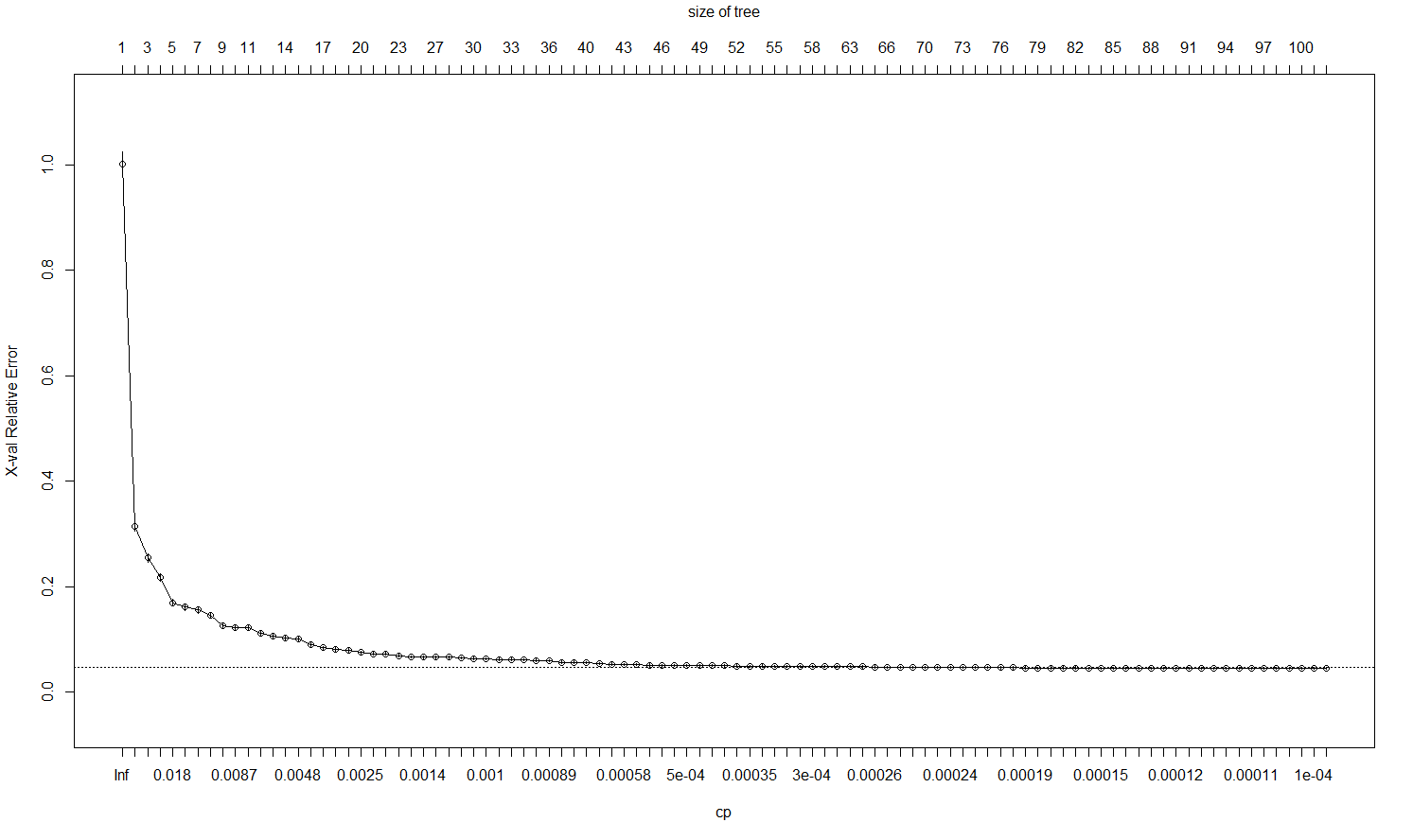
10.53469

The log transformation improved performance of the tree model, however it did not do as well as the MLR model using diamond price in the log scale.

The rpart function will perform internal 10-fold cross validation to help us choose an “optimal” cp value. The function plotcp will plot the cross-validation error vs. the cp and give table of the results. To use it effectively fit a base model with a very small cp value (.0001 or even smaller) and then use plotcp to find the optimal setting for cp.

> tree4 = rpart(log(Price)~.,data=diam.train,cp=.0001)

> plotcp(tree4)



Optimal cp around here?

> tree.opt = rpart(log(Price)~.,data=diam.train,cp=.00058)

> ypredlog = predict(tree.opt,newdata=diam.valid)

> ypred = exp(ypredlog)

> PredAcc(diam.valid$Price,ypred)

RMSEP

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

750.3385

MAE

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

495.6356

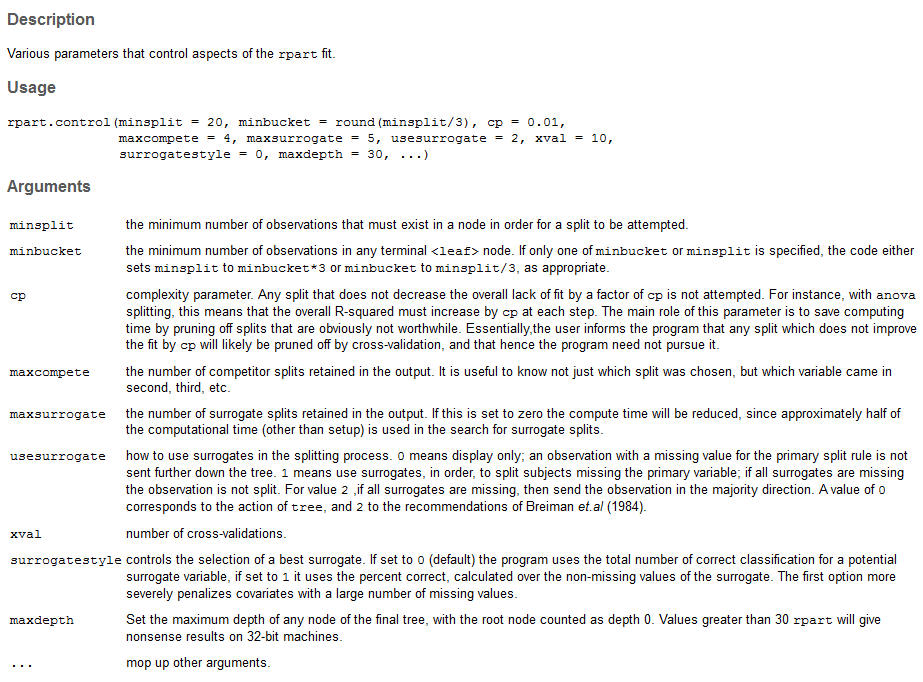
MAPE

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

11.91246

Hmmm… the 10-fold cross-validation suggests a cp that actually decreases predictive performance for the validation data. We have the validation dataset to assess our models, so let’s try tweaking some more settings (tuning parameters) in the recursive partitioning algorithm.

> ? rpart.control



The parameters minsplit and minbucket are good choices for additional model tuning.

I generally try minsplit before minbucket but both can manipulated. Also minbucket automatically equals minbucket/3 so there is generally no reason to change both. After experimenting with different settings for cp and minsplit I arrived at the following reasonably good RPART model, however it still falls short of our “best” MLR model.

> tree.opt = rpart(log(Price)~.,data=diam.train,control=rpart.control(cp=.00005,minsplit=5))

> ypredlog = predict(tree.opt,newdata=diam.valid)

> ypred = exp(ypredlog)

> PredAcc(diam.valid$Price,ypred)

RMSEP

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

640.7596

MAE

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

397.7874

MAPE

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

9.706756

We can use Monte Carlo Split-Sample CV instead of the train/validation/test set approach. The function below will perform MCCV for an RPART model.

rpart.sscv = function(fit,data,p=.667,B=100,

cp=fit$control$cp,minsplit=fit$control$minsplit) {

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 = rpart(formula(fit),data=data[sam,],cp=cp,minsplit=minsplit)

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("MAEP\n")

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

cat(MAEP,"\n\n")

cat("MAPEP\n")

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

cat(MAPEP,"\n\n")

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

return(temp)

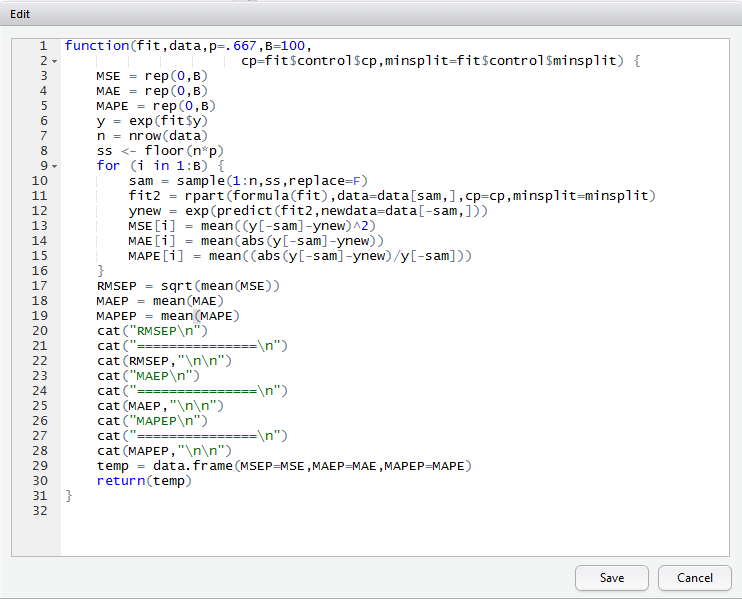
}

As the response will be transformed to the log scale for the diamond data, we will need to modify the highlighted lines, so we can measure predictive performance in the original scale.

To make these changes in R, do the following

> temp = edit(rpart.sscv)

An editor window will then open which you can use your mouse to navigate through.



We can then change the y = fit$y to be y = exp(fit$y) and change ynew = predict(…) to ynew = exp(predict(…)). (see the above highlighted portions) Once you are satisfied the temp function you created is working you can give it a more meaningful name.

> rpart.logsscv = temp

As we will be using MCCV to choose an “optimal” model, we start with the full dataset again.

> Diamonds = read.table(file.choose(),header=T,sep=”,”)

> Diamonds = Diamonds[,-10]

> tree1 = rpart(log(Price)~.,data=Diamonds)

> results = rpart.logsscv(tree1,data=Diamonds,cp=.00005,minsplit=5)

RMSEP

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

592.2878

MAEP

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

382.4264

MAPEP

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

0.09480442

> results = rpart.logsscv(tree1,data=Diamonds,cp=.0005,minsplit=5)

RMSEP

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

685.5508

MAEP

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

458.3314

MAPEP

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

0.112681

> results = rpart.logsscv(tree1,data=Diamonds,cp=.000025,minsplit=5)

RMSEP

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

601.4671

MAEP

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

387.8461

MAPEP

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

0.09645279

> results = rpart.logsscv(tree1,data=Diamonds,cp=.00005,minsplit=3)

RMSEP

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

601.7505

MAEP

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

386.1854

MAPEP

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

0.09610241

Best I could find using this approach was cp = .00005 and minsplit=5. We can then run the model on our training and validation cases combined and predict the test cases.

> temp = rbind(diam.train,diam.valid)

> rpart.final = rpart(log(Price)~.,data=temp,cp=.00005,minsplit=5)

> ypredlog = predict(rpart.final,newdata=diam.test)

> ypred.test = exp(ypredlog)

> PredAcc(diam.test$Price,ypred.test)

RMSEP

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

548.6262

MAE

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

348.7335

MAPE

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

8.798837

Still falls short of our best MLR model, but not by much.

**Tasks**

1. Build a regression tree for **Chicago Homes** **training** data set and use it to predict the list prices of homes in the **Chicago Homes test** set. Again do not use ZIP code.

> library(rpart)

> library(rpart.plot)

> ChiTrain = read.table(file.choose(),header=T,sep=”,”)

> ChiTest = read.table(file.choose(),header=T,sep=”,”)

> ChiTrain2 = ChiTrain[,-3]

> ChiTest2 = ChiTest[,-3]

> chi.rpart = rpart(log(ListPrice)~.,data=ChiTrain2)

> par(mfrow=c(1,1)

> plot(chi.rpart)

> text(chi.rpart)

> prp(chi.rpart,type=4,digits=4)

What is the prediction accuracy of this default model?

1. Now use the training data along with the rpart.logsscv function to tune your RPART model using cp and minsplit.   
     
   Who can find the best RPART model for these data using the metrics in the PredAcc function?
2. Using the **Diamonds** data set run the following code.

Diamonds = read.table(file.choose(),header=T,sep=”,”)  
library(rpart)  
library(rpart.plot)  
fit = rpart(log(Price)~.,data=Diamonds)

Copy the code for the function below into R

tree.vary = function(fit,data) {

n = nrow(data)

sam = sample(1:n,floor(n\*.5),replace=F)

temp = rpart(formula(fit),data=data[sam,])

prp(temp,type=4,digits=3)

}

> tree.vary(fit,data=Diamonds)

Then use the arrow keys run the tree.vary function above multiple times.   
 Look at each tree as it is plotted, what do you notice?

**4.3 – Ensemble Methods (Bagging, Random Forests, and Boosted Trees)**

**Bagging for Regression Trees**

Suppose we are interested in predicting a numeric response variable

and

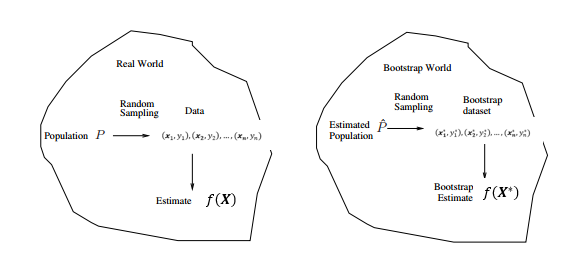
For example, , might come from a MLR model or from a RPART model using ***x*** with a complexity parameter cp = .005. Letting denote , where the expectation is with respect to the distribution underlying the training sample (since, viewed as a random variable, is a function of training sample, which can be viewed as a high-dimensional random variable) and not (which is considered fixed), we have that:

Thus in theory, if our prediction could be based on instead of then we would have a smaller mean squared error for prediction (and a smaller RMSEP as well). How can we approximate ? We could take a large number of samples of size *n* from the population and fit the specified model to each. Then average across these samples to get an average model , more specifically we could get the average prediction from the different models for a given set of predictors *.* Of course, this is silly as we only take one sample of size *n* in general when conducting any study. However, we can approximate this via the bootstrap. The bootstrap involves taking *B* random samples of size *n* drawn with replacement from our original sample . For each bootstrap sample, *b*, we will obtain an estimated model and average those to obtain a final estimate of , i.e.

This estimator for should in theory be better than the one obtained from the training data. This process of averaging the predicted values from a given is called ***bagging***. Bagging works best when the fitted models vary substantially from one bootstrap sample to the next. Modeling schemes that are complicated and involve the effective estimation of a large number parameters will benefit from bagging most. Bagging (or model averaging) is an example of what is referred to as an ***ensemble model*** in statistical learning, where results from different models are combined, in the case of bagging through averaging. We will examine other types of ensemble models later in this section.

**More on Bootstrap Sampling in Model Building**

The ***bootstrap*** in statistics is a method for approximating the sampling distribution of a statistic by resampling from our observed random sample. To put it simply, a bootstrap sample is a sample of size *n* drawn with replacement from our original sample.   
  
The bootstrap treats the original random sample as the estimated population () and draws repeated samples with replacement from it. For each bootstrap sample we can fit our predictive model.



A bootstrap sample for model building problems is illustrated below.

here the are the *p*-dimensional predictor vectors.

where is a random selected observation from the original data drawn with replacement.

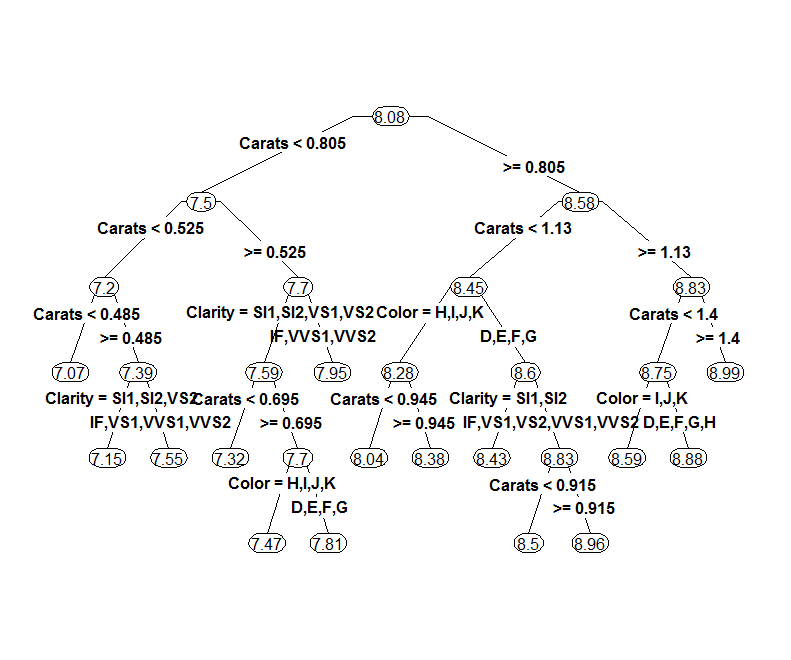
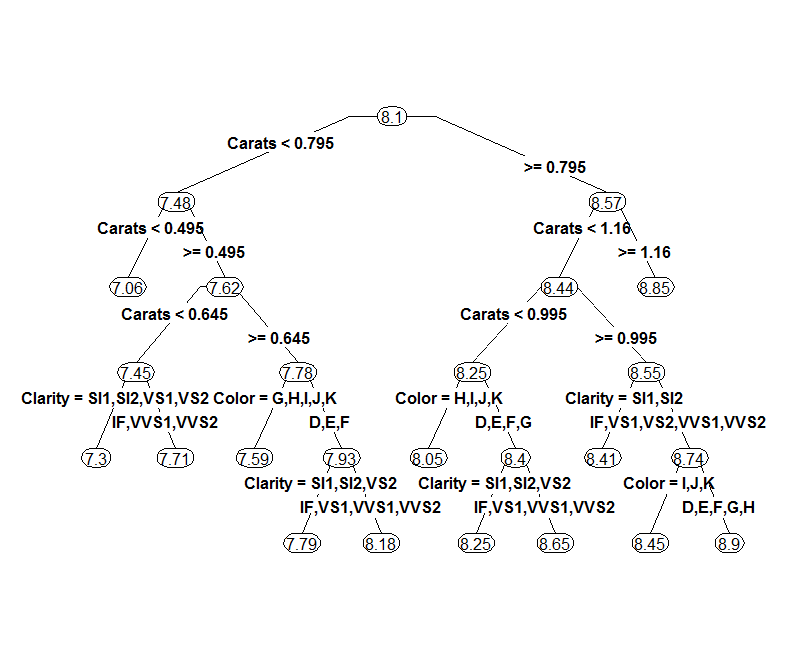
We can use the bootstrap sample to calculate any statistic of interest. This process is then repeated a large number of times (B = 500, 1000, 5000, etc.).

For estimating prediction error we fit whatever model we are considering to our bootstrap sample and use it to predict the response value for observations not selected in our bootstrap sample. One can show that about 63.2% of the original observations will represented in the bootstrap sample and about 36.8% of the original observations will not be selected. *Can you show this?* Thus we will almost certainly have some observations that are not represented in our bootstrap sample to serve as a validation set, with the selected observations in our bootstrap sample serving as our training set. For each bootstrap sample we can predict the response for the cases in the validation set (i.e. indices for observations not represented in our bootstrap sample). Bagging and other ensemble methods will use this idea to estimate predictive performance internally.

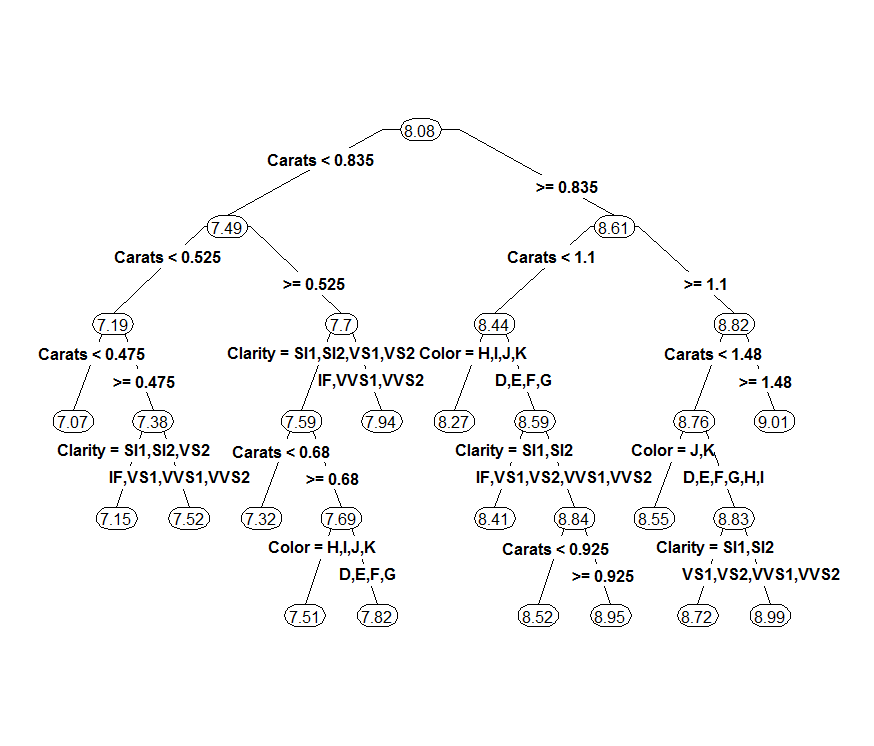
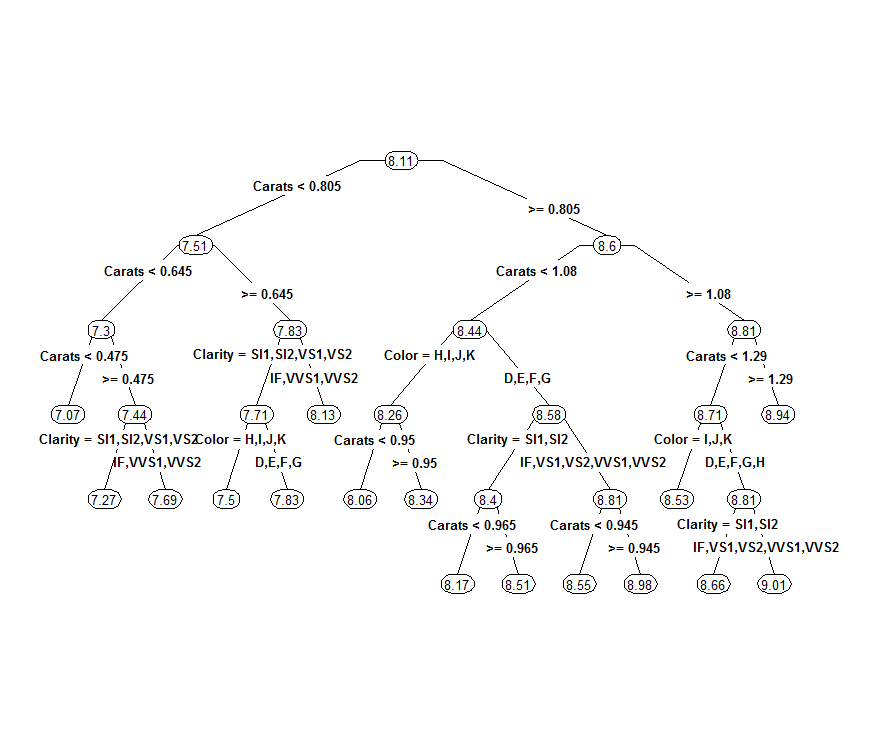
**Bagging Example: Diamond Data**

We have already determined in the previous section that an RPART model with cp = .00005 and minsplit = 5 worked well for these data. We will now use bagging to hopefully arrive at an even better model the price of a diamond. For simplicity we will first use smaller and simpler trees to to illustrate the idea of bagging. Below are four different trees fit to bootstrap samples drawn from the full Diamonds data frame. For each tree fit I used cp =.005 and minsplit = 5.

Tree 1 Tree 2



Tree 3 Tree 4

The bagged tree estimate for diamond price would simply take the average of the fitted values from all four trees obtained from bootstrap samples drawn from the training data. For predicting the response of a test or validation set, we would obtain the predictions from each of these trees and simply average them.

We now consider using bagging to improve the predictions from RPART applied to one specific (i.e. set of tuning parameters) model fit to our training sample. The package ipred from CRAN contains a function bagging which will perform bagging on regression trees obtain using RPART.

> library(ipred)

Below we use bagging to fit 10-bootstrap regression trees to a sample drawn from our training data frame diam.train. The predictions for the cases not appearing in each of the 10 bootstrap samples are used to estimate the RMSEP in the usual way.

> diam.train = Diamonds[Diamonds$Test==0,-10]

> diam.valid = Diamonds[Diamonds$Test==1,-10]

> diam.test = Diamonds[Diamonds$Test==2,-10]

**Bagging regression trees with 10 bootstrap replications**

> diam.bag = bagging(log(Price)~.,data=diam.train,coob=T,nbagg=10,  
 control=rpart.control(cp=.005,minsplit=5,xval=0))

> diam.bag

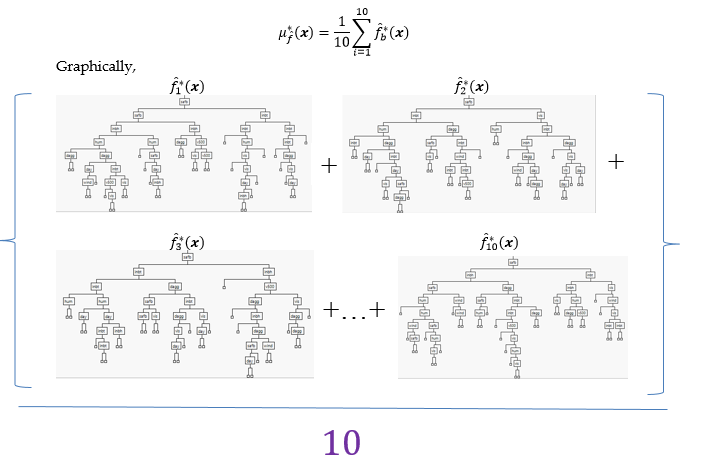
Bagging regression trees with 10 bootstrap replications

Call: bagging.data.frame(formula = log(Price) ~ ., data = diam.train,

control = rpart.control(cp = 0.005, minsplit = 5, xval = 0),

coob = T, nbagg = 10)

Out-of-bag estimate of root mean squared error: 0.1855 🡨 RMSEP estimate (LOG SCALE!!)  
  
Recall that the final estimated bagged estimate () is the average of the predictions from the 10-bootstrap regression trees, i.e.



As we can see the regression trees fit to the 10 bootstrap samples differ to some degree.

Increasing the number of bootstrap samples used () should improve the performance of the bagged estimate of the regression tree.

How well does this simple bagged estimate work? Let’s use it to predict the prices in the validation cases in the original scale using a bagged model using trees.

> ypredlog = predict(diam.bag,newdata=diam.valid)

> ypred = exp(ypredlog)

> PredAcc(diam.valid$Price,ypred)

RMSEP

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

894.8456

MAE

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

619.428

MAPE

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

15.00576

Well if this is your first impression of bagging (which I assume it is), your conclusion is probably that bagging stinks! This is far worse than any of the other models we have developed for these data. Before we throw bagging out the window, consider that (1) we haven’t fit very good trees (too small) and (2) we have only used B = 10 models in our averaging. Let’s use the tuning parameters we chose earlier (cp = .00005, minsplit = 5) and increase the number of trees being averaged (i.e. bagged).

> diam.bag2 = bagging(log(Price)~.,data=diam.train,coob=T,nbagg=25,  
control=rpart.control(cp=.00005,minsplit=5,xval=0))

> diam.bag2

Bagging regression trees with 25 bootstrap replications

Call: bagging.data.frame(formula = log(Price) ~ ., data = diam.train,

coob = T, nbagg = 25, control = rpart.control(cp = 5e-05,

minsplit = 5, xval = 0))

Out-of-bag estimate of root mean squared error: 0.1117

The out-of-bag estimate of the RMSEP comes from predicting the cases not selected in of the 25 bootstrap samples used in the bagging process.

> diam.bag2 = bagging(log(Price)~.,data=diam.train,coob=T,nbagg=100,  
control=rpart.control(cp=.00005,minsplit=5,xval=0))

> diam.bag2

Bagging regression trees with 100 bootstrap replications

Call: bagging.data.frame(formula = log(Price) ~ ., data = diam.train,

coob = T, nbagg = 100, control = rpart.control(cp = 5e-05,

minsplit = 5, xval = 0))

Out-of-bag estimate of root mean squared error: 0.1036

> diam.bag2 = bagging(log(Price)~.,data=diam.train,coob=T,nbagg=1000,  
control=rpart.control(cp=.00005,minsplit=5,xval=0))

> diam.bag2

Bagging regression trees with 1000 bootstrap replications

Call: bagging.data.frame(formula = log(Price) ~ ., data = diam.train,

coob = T, nbagg = 1000, control = rpart.control(cp = 5e-05,

minsplit = 5, xval = 0))

Out-of-bag estimate of root mean squared error: 0.1018

🡨 warning this is slow! (3 minutes or so – don’t; run this!)

> diam.bag2 = bagging(log(Price)~.,data=diam.train,coob=T,nbagg=10000,  
control=rpart.control(cp=.00005,minsplit=5,xval=0))

> diam.bag2

Bagging regression trees with 10000 bootstrap replications

Call: bagging.data.frame(formula = log(Price) ~ ., data = diam.train,

coob = T, nbagg = 10000, control = rpart.control(cp = 5e-05,

minsplit = 5, xval = 0))

Out-of-bag estimate of root mean squared error: 0.1021

Fortunately we see diminishing returns on the number of bootstrap samples used in the bagging process. For larger more complex data than the diamonds data, using bootstrap samples could be very expensive computationally!

How well do these different bagged trees perform when predicting the validation cases?

> ypredlog = predict(diam.bag2,newdata=diam.valid)

> ypred = exp(ypredlog)

> PredAcc(diam.valid$Price,ypred)

RMSEP

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

524.4023

MAE

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

336.7267

MAPE

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

8.289095

> ypredlog = predict(diam.bag2,newdata=diam.valid)

> ypred = exp(ypredlog)

> PredAcc(diam.valid$Price,ypred)

RMSEP

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

525.2738

MAE

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

337.7534

MAPE

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

8.30029

> ypredlog = predict(diam.bag2,newdata=diam.valid)

> ypred = exp(ypredlog)

> PredAcc(diam.valid$Price,ypred)

RMSEP

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

527.0671

MAE

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

338.9526

MAPE

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

8.316951

Below is code for a split-sample Monte Carlo cross-validation function that takes a model from specified using rpart, (not bagging) as an argument. The function bag.sscv will automatically grab any tuning parameters that have been set in the rpart fit, namely cp, minbucket, and minsplit. If you did not specify these yourselves it will still grab their default settings. It will compute the usual prediction measures (RMSEP, MAEP, and MAPEP) but also will save the out-of-bag estimates of the RMSEP from the bootstrap samples used in the bagging process.

We need can also specify the number the fraction of data to using the Monte Carlo split-sample training sets ( by default), the number of Monte Carlo simulations to run ( by default), and the number of bootstrap samples based upon the training data to use in the bagging process ( by default).

**Monte Carlo Split-Sample CV code for Bagging Regression Trees**

bag.sscv = function(fit,data,p=.667,M=100,B=25,  
cp=fit$control$cp,minbucket=fit$control$minbucket,minsplit=fit$control$minsplit) {

OOBMSE = rep(0,M)

MSE = rep(0,M)

MAE = rep(0,M)

MAPE = rep(0,M)

y = fit$y

n = nrow(data)

ss <- floor(n\*p)

for (i in 1:M) {

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

fit2 = bagging(formula(fit),data=data[sam,],nbagg=B,coob=T,  
 control=rpart.control(cp=cp,  
 minbucket=minbucket,  
 minsplit=minsplit,  
 xval=0))

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

OOBMSE[i] = fit2$err  
 MSE[i] = mean((y[-sam]-ynew)^2)

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

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

}

OOB.RMSEP = mean(OOBMSE)

RMSEP = sqrt(mean(MSE))

MAEP = mean(MAE)

MAPEP = mean(MAPE)

cat("OOB RMSEP\n")

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

cat(OOB.RMSEP,"\n\n")

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(OOB.RMSEP=OOBMSE,MSEP=MSE,MAEP=MAE,MAPEP=MAPE)

return(temp)

}

Again add exp() to the lines highlighted if a log transform was used for the response.

> fit = rpart(log(Price)~.,data=Diamonds,cp=.00005,minsplit=5

> bag.logsscv = edit(bag.logsscv)

> results = bag.logsscv(fit,Diamonds,M=25,B=50)

OOB RMSEP

=============== 🡨 THIS IS IN LOG SCALE!

0.103759

RMSEP

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

484.9549

MAE

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

311.9162

MAPE

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

7.66815 🡨 best yet?!?

**Tasks**

1. Build a bagged regression tree for **Chicago Homes** **training** data set and use it to predict the list prices of homes in the **Chicago Homes test** set. Again do not use ZIP code.

> library(rpart)

> library(rpart.plot)

> library(ipred)

> ChiTrain = read.table(file.choose(),header=T,sep=”,”)

> ChiTest = read.table(file.choose(),header=T,sep=”,”)

> ChiTrain2 = ChiTrain[,-3]

> ChiTest2 = ChiTest[,-3]

> fit = rpart(log(ListPrice)~.,data=ChiTrain,cp=??,minsplit=??)

> home.bag = bagging(fit,data=ChiTrain,coob=T,nbagg=??,  
 control=rpart.control(cp=??,minsplit=??,xval=0))

> ypredlog = predict(home.bag,newdata=ChiTest)  
> ypred = exp(ypredlog)

> PredAcc(ChiTest$ListPrice,ypred)

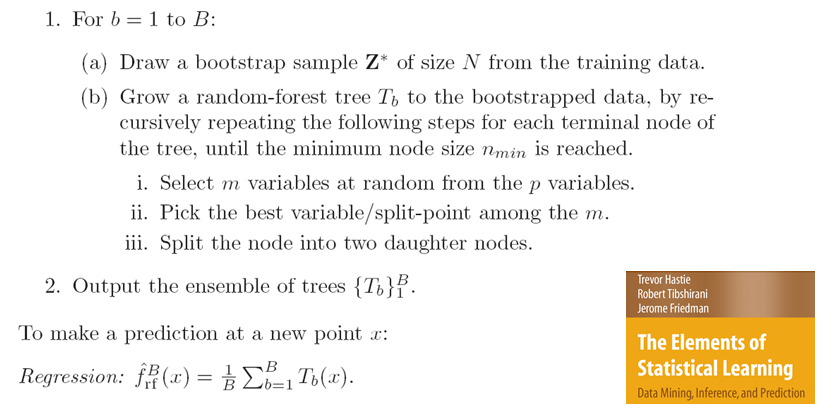
Remember you can use the bag.logsscv() function to help find you tuning parameters.

Who can build the best bagged model?

**Random Forests – these are typically the benchmark for Kaggle ® problems**

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.

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. genetics, QSAR, etc.)
* It estimates the importance of input variables in the model.
* 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,

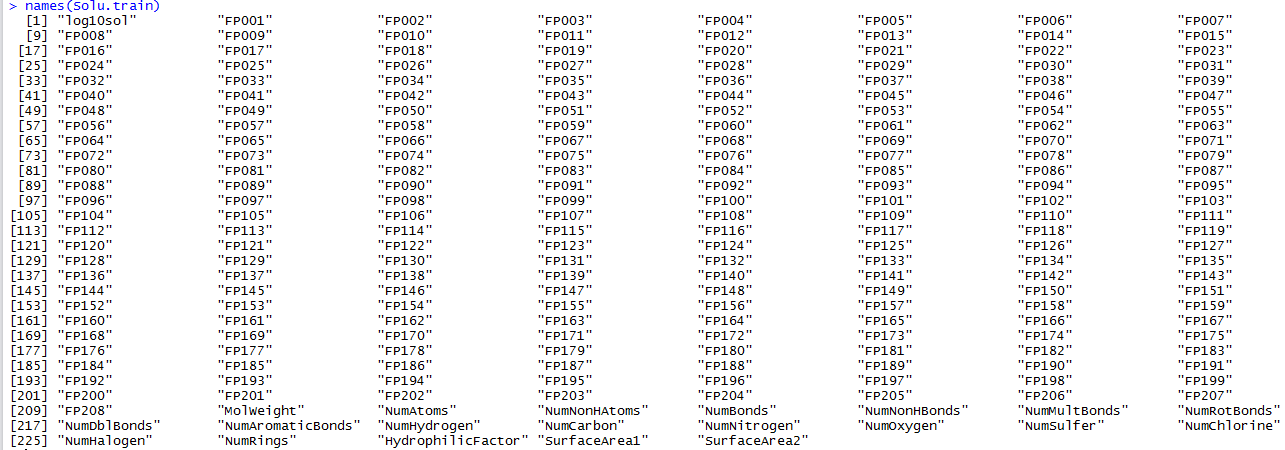
Finally, average the raw scores over all the *B* trees in the forest, i.e. expression below is computed

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.

**Random Forest Example: Chemical Solubility (not diamonds, hurray!)  
Datafiles: Solubility(train).csv and Solubility(test).csv**

> Solu.train = read.table(file.choose(),header=T,sep=”,”)

> Solu.test = read.table(file.choose(),header=T,sep=”,”)

> names(Solu.train)  


The response is the and there are potential predictors!

> dim(Solu.train)

[1] 951 228

> dim(Solu.test)

[1] 316 228

We will begin by fitting a baseline random forest with all of the tuning parameters (which are described below) set to their default settings.  
  
> solu.rf = randomForest(log10sol~.,data=Solu.train)

> solu.rf

Call:

randomForest(formula = log10sol ~ ., data = Solu.train)

Type of random forest: regression

Number of trees: 500

No. of variables tried at each split: 76

Mean of squared residuals: 0.4257285

% Var explained: 90.28

Additional settings, i.e. tuning parameters, in fitting a random forest using randomForest:

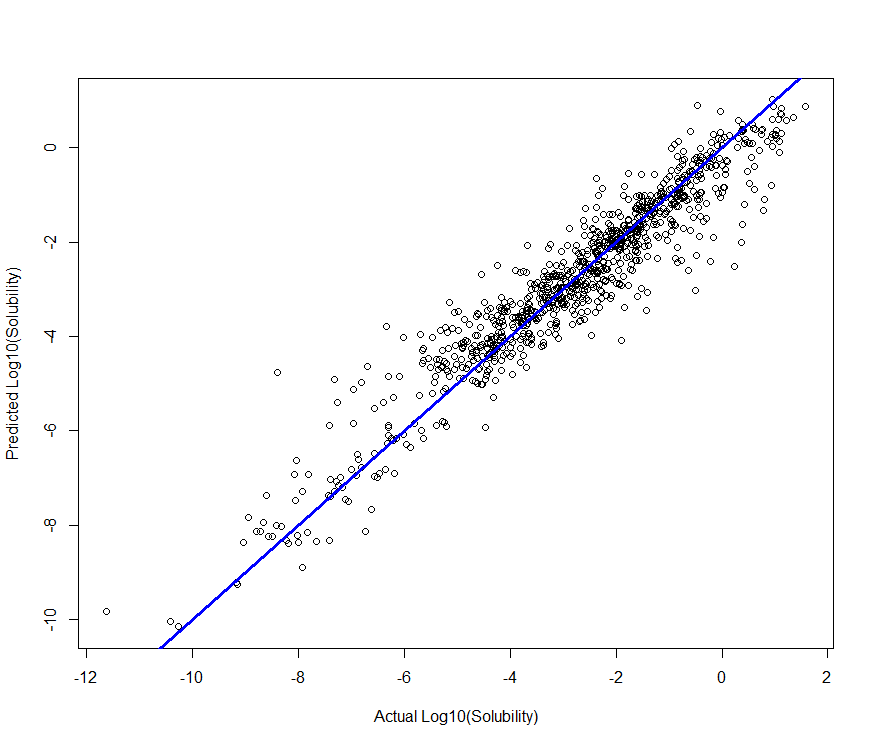
ntree – number of trees to grow, like B or nbagg in bagging

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(Solu.train$log10sol,predict(solu.rf),  
xlab="Actual Log10(Solubility)",ylab="Predicted Log10(Solubility)")

> abline(0,1,lwd=3,col="blue")



Our first model seems pretty good, but we have no basis for comparison. We can again use cross-validation to choose values for the tuning parameters to improve our model.

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)

}

> solu.rf = randomForest(log10sol~.,data=Solu.train,mtry=20,ntree=500)

> results = rf.sscv(solu.rf,data=Solu.train,B=20)

RMSEP

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

0.7256308

MAE

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

0.5284393

Log10(Solubility) is 0 for several compounds, thus MAPE is infinite.

MAPE

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

Inf

> solu.rf = randomForest(log10sol~.,data=Solu.train,mtry=30,ntree=500)

> results = rf.sscv(solu.rf,data=Solu.train,B=20)

RMSEP

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

0.7126363

MAE

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

0.520465

> solu.rf = randomForest(log10sol~.,data=Solu.train,mtry=40,ntree=500)

> results = rf.sscv(solu.rf,data=Solu.train,B=20)

RMSEP

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

0.7113822

MAE

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

0.5100485

> solu.rf = randomForest(log10sol~.,data=Solu.train,mtry=50,ntree=500)

> results = rf.sscv(solu.rf,data=Solu.train,B=20)

RMSEP

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

0.7050137

MAE

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

0.5101033

> solu.rf = randomForest(log10sol~.,data=Solu.train,mtry=60,ntree=500)

> results = rf.sscv(solu.rf,data=Solu.train,B=20)

RMSEP

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

0.7067367

MAE

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

0.5094528

> solu.rf = randomForest(log10sol~.,data=Solu.train,mtry=70,ntree=500)

> results = rf.sscv(solu.rf,data=Solu.train,B=20)

RMSEP

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

0.6929079

MAE

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

0.4995794

> solu.rf = randomForest(log10sol~.,data=Solu.train,mtry=80,ntree=500)

> results = rf.sscv(solu.rf,data=Solu.train,B=20)

RMSEP

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

0.6924968

MAE

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

0.501014

> solu.rf = randomForest(log10sol~.,data=Solu.train,mtry=90,ntree=500)

> results = rf.sscv(solu.rf,data=Solu.train,B=20)

RMSEP

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

0.7239825

MAE

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

0.5218389

For our final random forest for these data we will use mtry = 80 and ntree =500.

> solu.final = randomForest(log10sol~.,data=Solu.train,mtry=80,ntree=500)

> solu.final

Call:

randomForest(formula = log10sol ~ ., data = Solu.train, mtry = 80,ntree = 500)

Type of random forest: regression

Number of trees: 500

No. of variables tried at each split: 80

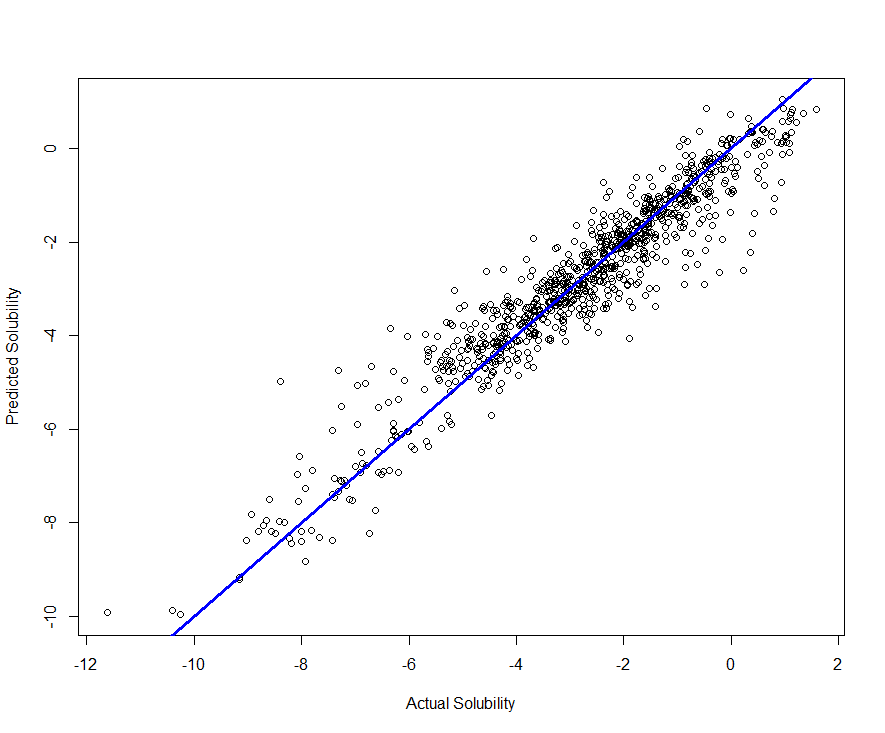
Mean of squared residuals: 0.4278436

% Var explained: 90.23

> plot(Solu.train$log10sol,predict(solu.final),xlab="Actual Solubility",

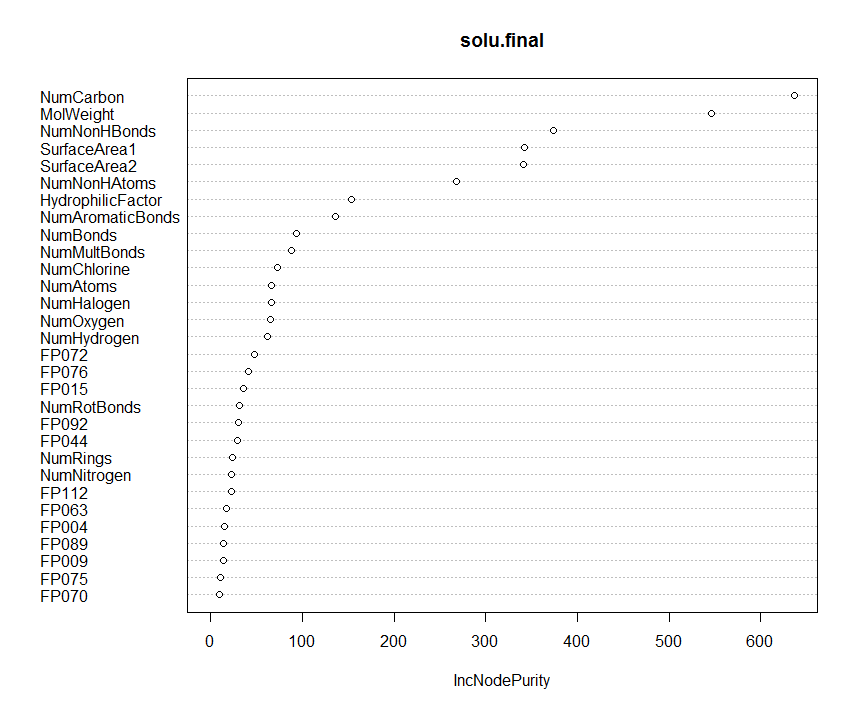
ylab = "Predicted Solubility")

> abline(0,1,lwd=3,col="blue")



As there are 227 predictors in our model, it would be helpful to know which are important. The function varImpPlot in the randomForest library will plot the most important predictors in the forest.

> varImpPlot(solu.final)



How does solubility change as a function of the top four numeric predictors? The command partialPlot in the randomForest library will display relationship between the response and the predictor of interest.

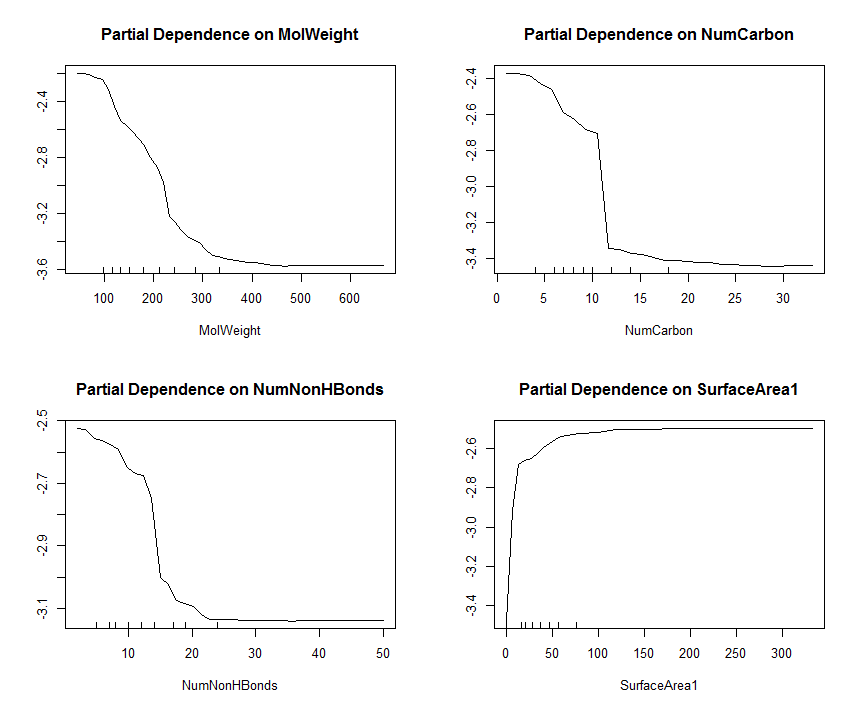
> par(mfrow=c(2,2))

> partialPlot(solu.final,Solu.train,MolWeight)

> partialPlot(solu.final,Solu.train,NumCarbon)

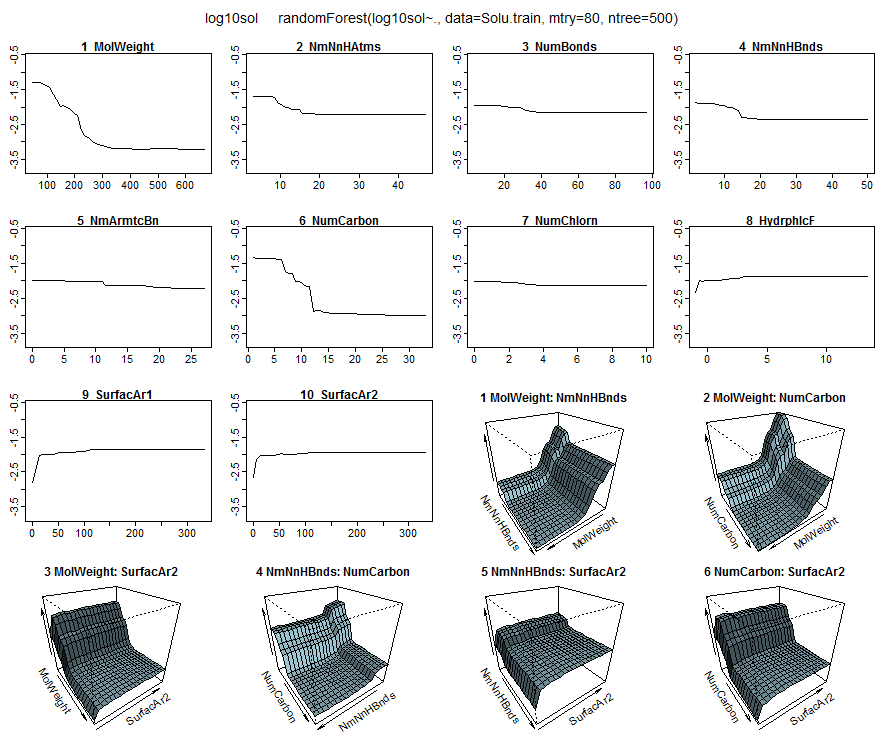
> partialPlot(solu.final,Solu.train,NumNonHBonds)

> partialPlot(solu.final,Solu.train,SurfaceArea1)



The plotmo package is a general plotting package for plotting models of various types including random forest.

> plotmo(solu.final)



Finally we predict the of the test cases.

> ypred = predict(solu.final,newdata=Solu.test)

> PredAcc(Solu.test$log10sol,ypred)

RMSEP

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

0.6470864

MAE

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

0.4559606

**Nice!**

**Tasks**

1. Build a random forest using **Chicago Homes** **training** data set and then predict the list prices of homes in the **Chicago Homes test** set. Again we will not use ZIP code in the modeling process.

library(randomForest)

library(plotmo)

ChiTrain = read.table(file.choose(),header=T,sep=”,”)

ChiTest = read.table(file.choose(),header=T,sep=”,”)

ChiTrain2 = ChiTrain[,-3]

ChiTest2 = ChiTest[,-3]

home.rf = randomForest(log(ListPrice)~.,data=ChiTrain)

home.rf

varImpPlot(home.rf)

par(mfrow=c(2,2)

partialPlot(home.rf,ChiTrain2,ImputedSQFT)

partialPlot(home.rf,ChiTrain2,BEDS)

partialPlot(home.rf,ChiTrain2,LATITUDE)

partialPlot(home.rf,ChiTrain2,LONGITUDE)

par(mfrow=c(1,1))

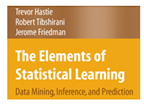
ypredlog = predict(home.rf,newdata=ChiTest)  
 ypred = exp(ypredlog)

PredAcc(ChiTest$ListPrice,ypred)

Remember you can use the rf.sscv() function to help find you tuning parameters. You can use this function to find best model using the log scale response and then convert your final predictions for the list prices of the test set homes.

Who can build the random forest for these data?

**Boosting**  
Boosting, like bagging, is way to combine or “average” the results of multiple trees in order to improve their predictive ability. Boosting however does not simply average trees constructed from bootstrap samples of the original data, rather it creates a sequence of trees where the next tree in sequence essentially uses the residuals from the previous trees as the response. Thus each successive tree in the sequence of trees is trying to explain the unexplained variation from the previous tree. This type of approach is referred to as *gradient boosting*. Using the squared error as the measure of fit, the Gradient Tree Boosting Algorithm is given below.



**Gradient Tree Boosting Algorithm (Squared Error)**

1. Initialize .
2. For
3. For compute

which are simply the residuals from the previous tree.

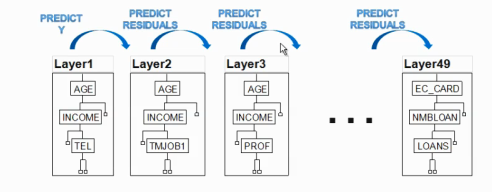
1. Fit a regression tree using as the response, giving terminal node regions .
2. For compute the mean of the residuals in each of the terminal nodes, call these
3. Update the model as follows:

The parameter is a shrinkage parameter which can be tweaked  
 along with and to improve cross-validated predictive performance.

1. Output .
2. *Stochastic Gradient Boosting* uses the same algorithm as above, but takes a random subsample of the training data (without replacement), and grows the next tree using only those observations. A typical fraction for the subsamples would be ½ but smaller values could be used when *n* is large.

The authors of Elements of Statistical Learning recommend using for the number of terminal nodes in the regression trees grown at each of the *M* steps. For classification trees smaller values of are used, with 2 being optimal in many cases. A two terminal node tree is called a *stump*. Small values of have been found to produce superior results for regression problems, however this generally will require a large value for *M*. For example, for shrinkage values between .001 and .01 it is recommended that the number of iterations be between 3,000 and 10,000. Thus in terms of model development one needs to consider various combinations of and *M*.

The algorithm as presented above looks a bit daunting at first, however the graphic below simplifies the boosting concept considerably.



Notice that in this diagram. The ***interaction depth*** is the number of splits mimus one, i.e.

# splits .

Models

Here M = 49, so the final model is simply,

The gbm function in the library of the same name will fit a boosted regression tree. It has numerous settings, some of which that are required and some that are optional. The basic function call for a boosted regression tree with a numeric response is shown below. This was taken from the help file for gbm with default settings shown.

gbm(formula = formula(data), 🡨 use of y~. notation is fine.

distribution = "bernoulli", 🡨 you must set this to “gaussian” for a numeric   
 response.  
 data = list(), 🡨 definitely needs this to go along with wild card model   
 specification. All variables besides response must be   
 valid predictors ()in the data frame.

weights,

var.monotone = NULL,

n.trees = 100, 🡨 number of layers in the notation above.

interaction.depth = 1, 🡨 , by default trees will have two terminal nodes,   
 i.e. a single split on single variable.

n.minobsinnode = 10, 🡨 will not allow splits that result in less than 10   
 observations in a terminal node.

shrinkage = 0.001, 🡨 , the shrinkage parameter, small values generally   
 have better predictive performance, but require   
 more iterations.

bag.fraction = 0.5, 🡨 will random select 50% of the cases at each stage to fit   
 the next tree (or layer).

train.fraction = 1.0, 🡨 will use all data to train the model. Values less than 1.0   
 will split data into a training and validation sets.

cv.folds=0, 🡨 setting to 5 or 10 will perform k-fold cross-validation   
 internally to estimate prediction error.

keep.data = TRUE,

verbose = TRUE, 🡨 will print fitting progress on screen.

class.stratify.cv=NULL,

n.cores = NULL) 🡨 change to # of cores on your computer if you have   
 multiple processors on your machine, like me ☺.

Let’s consider some examples.

**Boosted Tree Example: Diamond Data**

Using the training, validation, and test sets we have used previously we will develop boosted tree models using the training data and validate their performance using the validation set. Once we have chosen the “best” boosted tree using this process, we will give our final predictions for the test cases.

Here is the code we used to create these three data sets.

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

> diam.train = Diamonds[Diamonds$Test==0,-10]

> diam.valid = Diamonds[Diamonds$Test==1,-10]

> diam.test = Diamonds[Diamonds$Test==2,-10]

> diam.gbm = gbm(log(Price)~.,data=diam.train,distribution="gaussian",n.trees=5000,shrinkage=.01,interaction.depth=4,bag.fraction=0.5,train.fraction=.8,n.minobsinnode=5,cv.folds=5,keep.data=T,verbose=T)

Iter TrainDeviance ValidDeviance StepSize Improve

1 0.2971 1.1930 0.0100 0.0049

2 0.2923 1.1807 0.0100 0.0046

3 0.2875 1.1687 0.0100 0.0048

4 0.2830 1.1569 0.0100 0.0047

5 0.2784 1.1451 0.0100 0.0048

6 0.2738 1.1332 0.0100 0.0044

7 0.2694 1.1232 0.0100 0.0046

8 0.2651 1.1123 0.0100 0.0043

9 0.2609 1.1013 0.0100 0.0040

10 0.2568 1.0903 0.0100 0.0042

20 0.2198 0.9958 0.0100 0.0033

40 0.1631 0.8374 0.0100 0.0024

60 0.1238 0.7178 0.0100 0.0016

80 0.0962 0.6255 0.0100 0.0011  
 . . . . . . . . . . . . . . .

4840 0.0029 0.0686 0.0100 -0.0000

4860 0.0029 0.0686 0.0100 -0.0000

4880 0.0029 0.0685 0.0100 -0.0000

4900 0.0029 0.0683 0.0100 -0.0000

4920 0.0029 0.0680 0.0100 -0.0000

4940 0.0029 0.0679 0.0100 -0.0000

4960 0.0029 0.0679 0.0100 -0.0000

4980 0.0029 0.0677 0.0100 -0.0000

5000 0.0029 0.0677 0.0100 -0.0000

Verbose obviously provides a lot of detail about the fitted model that does not look particularly useful. We can plot the CV, Test/Validation set, and OOB performance results using the gbm.perf(). This function will create a plot of mean squared error vs. the number of boosting iterations, which is the number layers in the diagram above. The method used to determine the optimal number is determined by the argument to the method= option in the gbm.perf function. Options are “cv”, “test” and “OOB”. Examples are shown below.

> diam.gbm

gbm(formula = log(Price) ~ ., distribution = "gaussian", data = diam.train,

n.trees = 5000, interaction.depth = 4, n.minobsinnode = 5,

shrinkage = 0.01, bag.fraction = 0.5, train.fraction = 0.8,

cv.folds = 5, keep.data = T, verbose = T)

A gradient boosted model with gaussian loss function.

5000 iterations were performed.

The best cross-validation iteration was 3357.

The best test-set iteration was 4985.

There were 8 predictors of which 8 had non-zero influence.

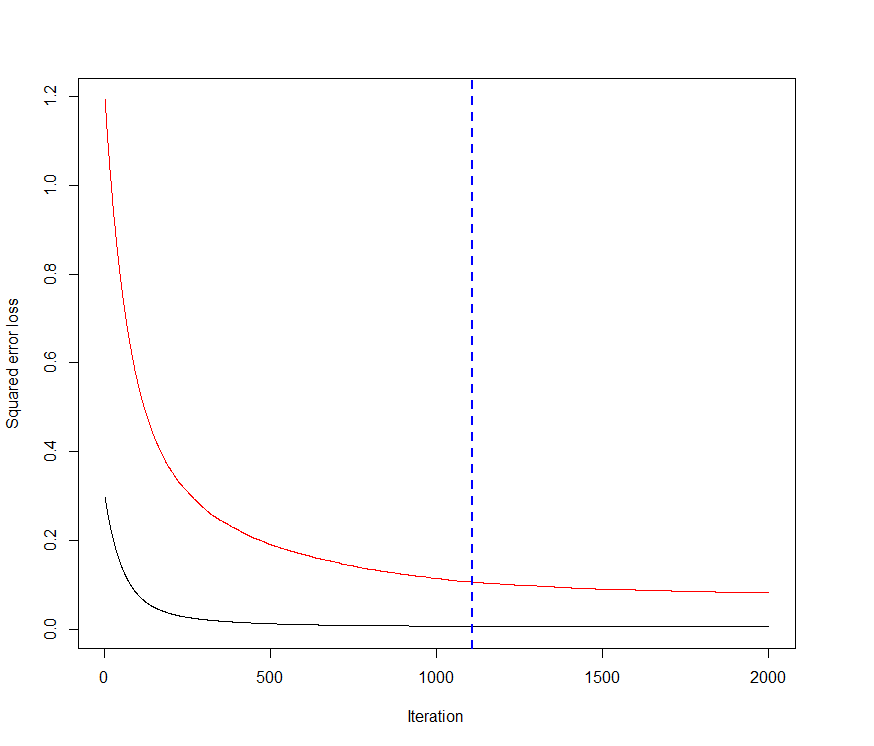
> gbm.perf(diam.gbm,method="OOB")

[1] 1107

Warning message:

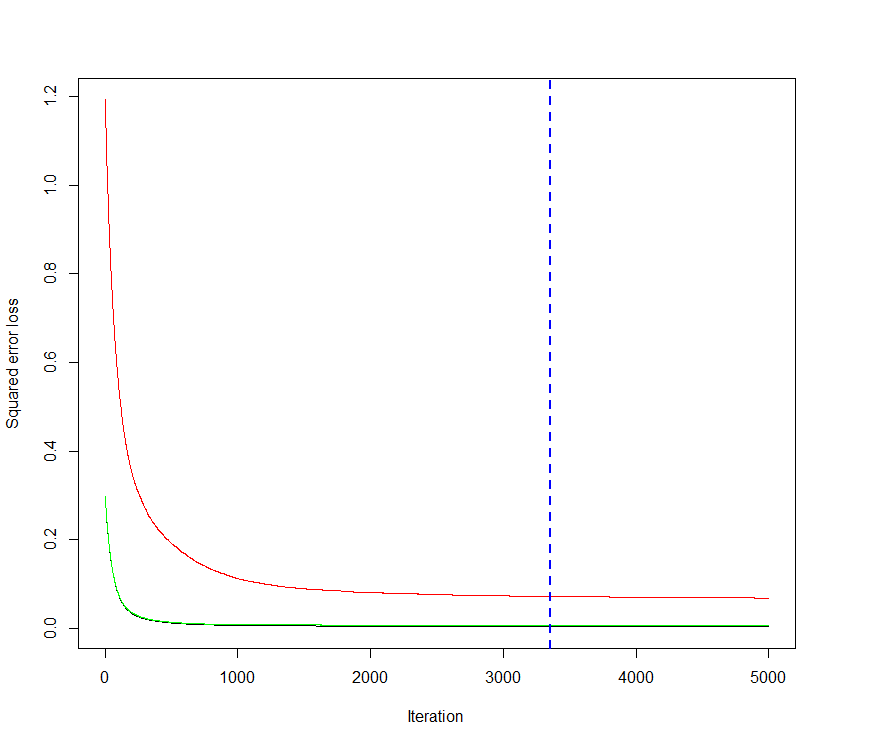
In gbm.perf(diam.gbm, method = "OOB") :

OOB generally underestimates the optimal number of iterations although predictive performance is reasonably competitive. Using cv.folds>0 when calling gbm usually results in improved predictive performance.



> gbm.perf(diam.gbm,method="test")

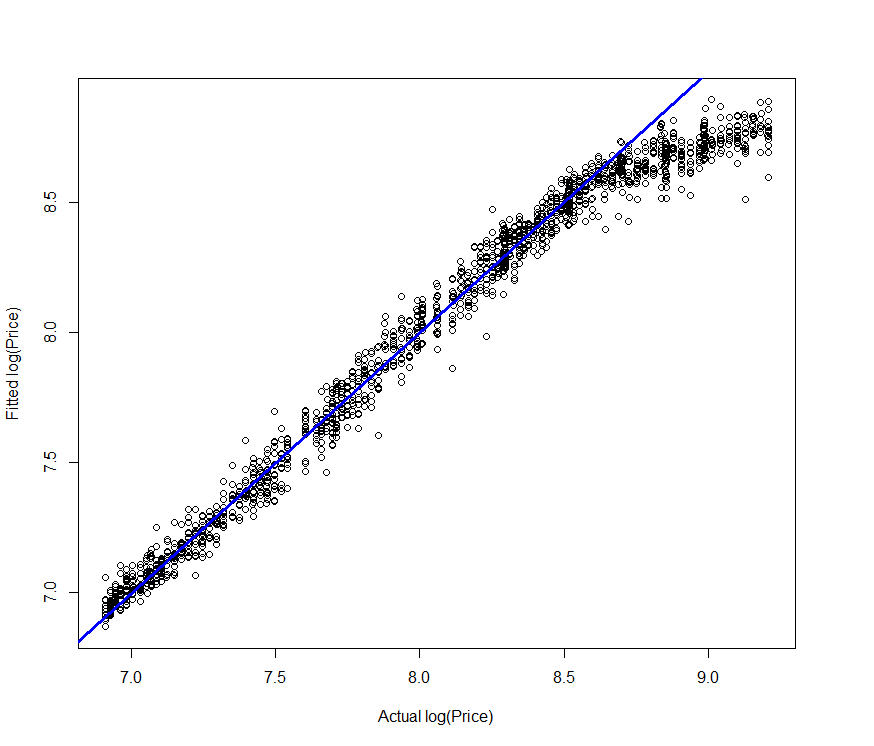
[1] 4985  
> gbm.perf(diam.gbm,method="cv")  
[1] 3357



Let’s examine the quality of the fit to the training data using the optimal number of iterations determined by 5-fold cross-validation.

> ypred = predict(diam.gbm,newdata=diam.train,n.trees=3357)

> plot(log(diam.train$Price),ypred,xlab="Actual log(Price)",ylab="Fitted log(Price)")

> abline(0,1,lwd=3,col="blue")  


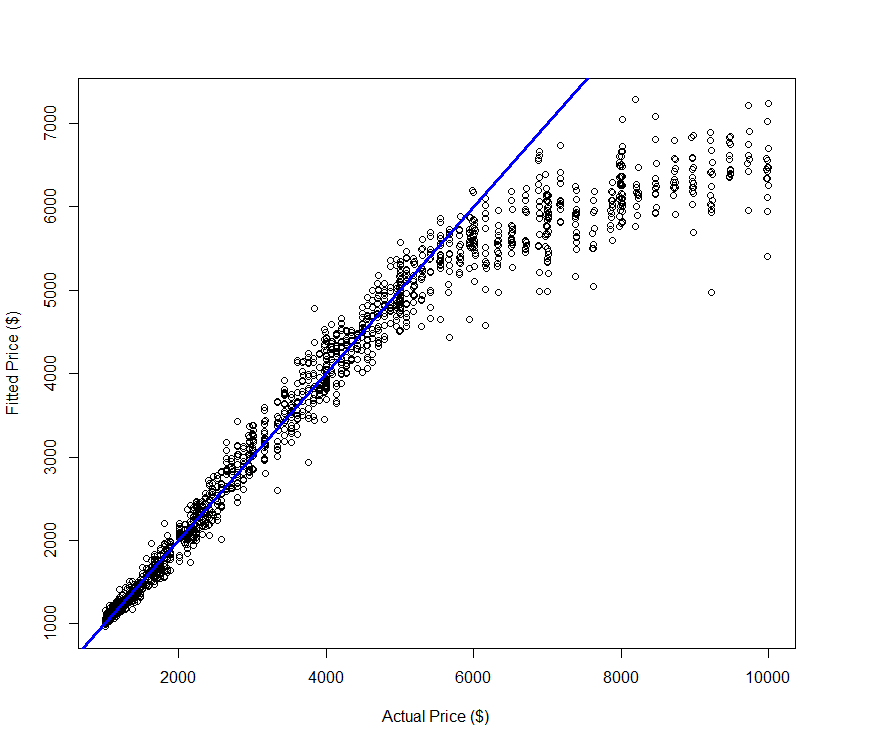
The model appears to underestimate the price of the most expensive diamonds.

Examining the quality of the fit in the original scale we have the following.

> ypred = exp(ypred)

> plot(diam.train$Price,ypred,xlab="Actual Price ($)",ylab="Fitted Price ($)")

> abline(0,1,lwd=3,col="blue")



The underfitting is more pronounced in the original scale.

After extensive experimentation with n.trees, shrinkage, and interaction.depth I arrived at the following reasonable model, though it is unlikely to be the “optimal” one.

> diam.gbm = gbm(formula = log(Price) ~ ., distribution = "gaussian",

data = diam.train, n.trees = 100000, interaction.depth = 1, n.minobsinnode = 5,

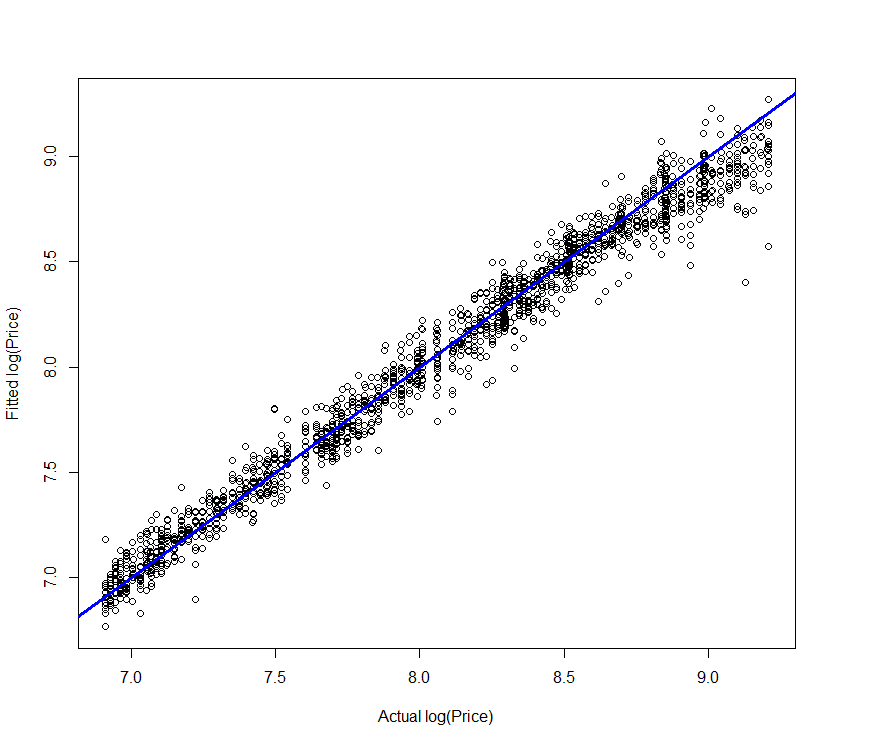
shrinkage = 0.0025, bag.fraction = 0.5, train.fraction = 0.8,

cv.folds = 5, keep.data = T, verbose = F)

> ypred = predict(diam.gbm,newdata=diam.train,n.trees=40339)

> plot(log(diam.train$Price),ypred,xlab="Actual log(Price)",ylab="Fitted log(Price)")

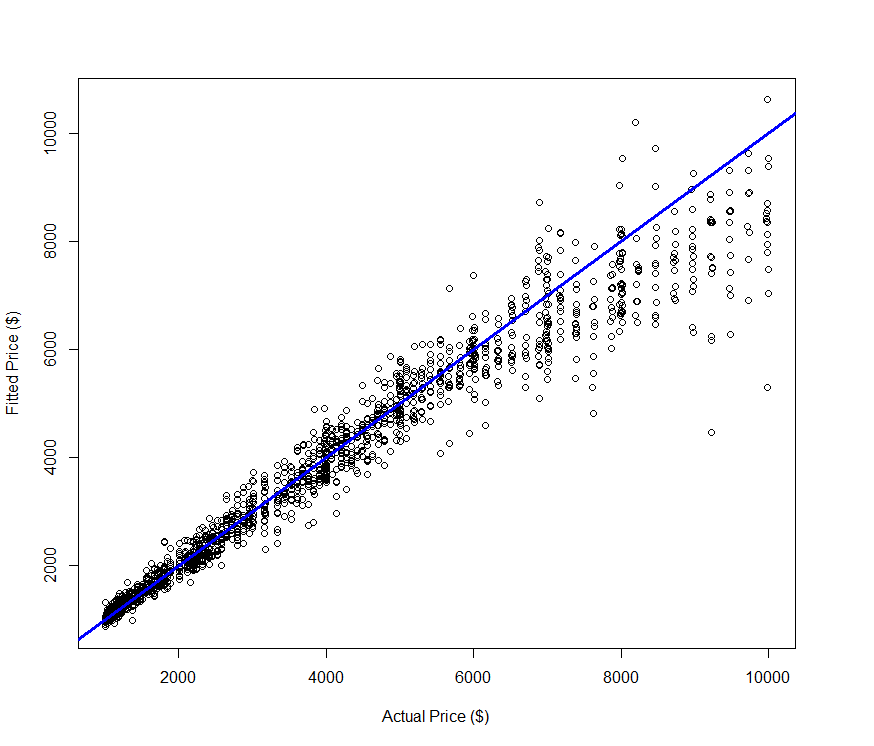
> abline(0,1,lwd=3,col="blue")



> ypred = exp(ypred)

> plot(diam.train$Price,ypred,xlab="Actual Price ($)",ylab="Fitted Price ($)")

> abline(0,1,lwd=3,col="blue")

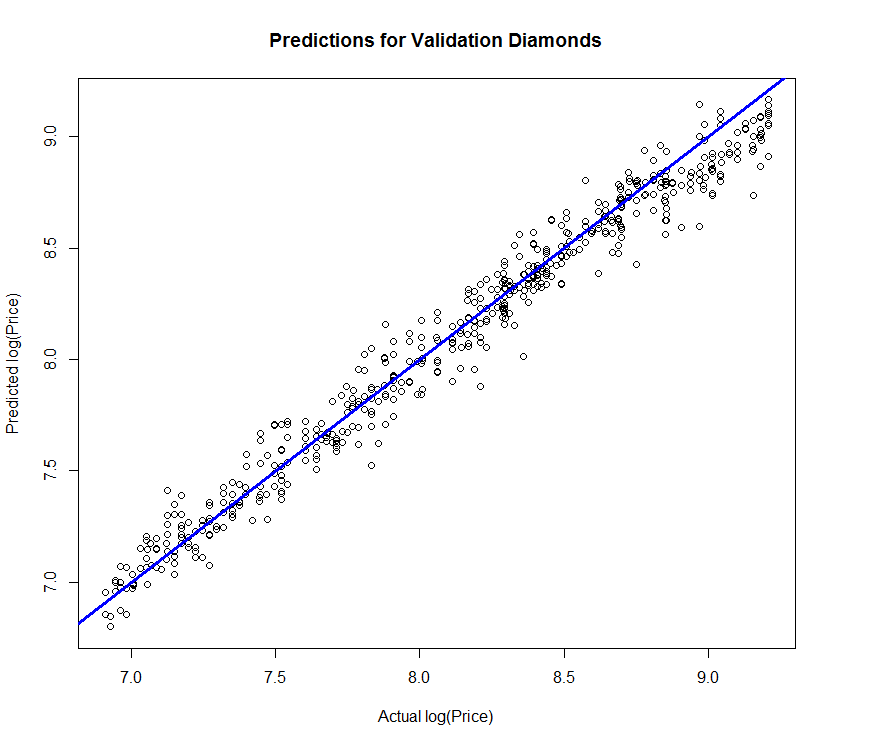


Next we will use this boosted tree model to predict the price of the validation diamonds in both the log and original scale.

> ypred = predict(diam.gbm,newdata=diam.valid,n.trees=40339)

> plot(log(diam.valid$Price),ypred,xlab="Actual log(Price)",ylab="Predicted log(Price)",main="Predictions for Validation Diamonds")

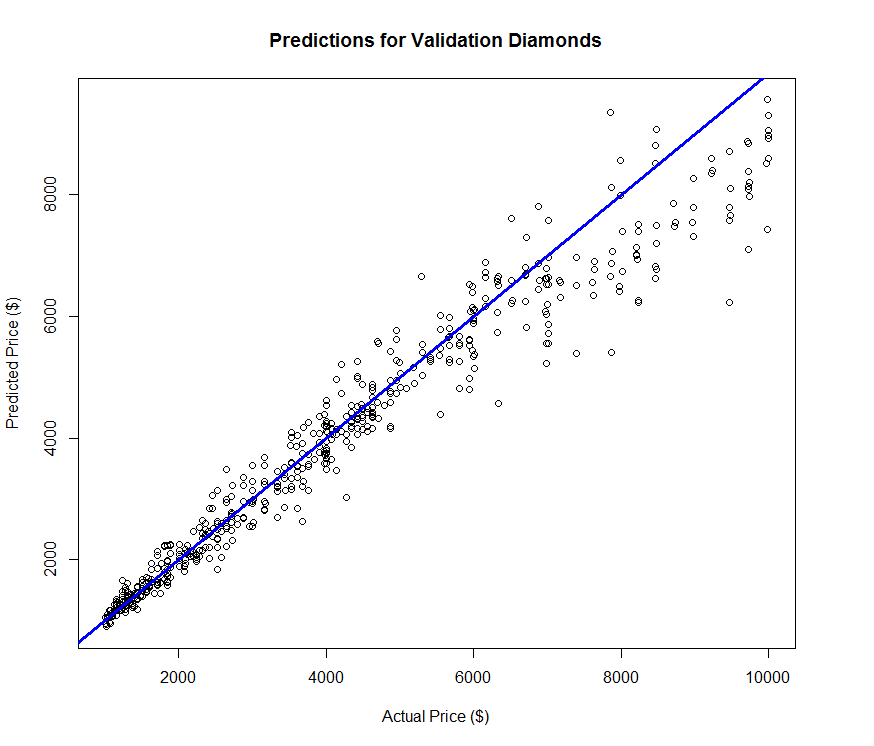
> abline(0,1,lwd=3,col="blue")



> ypred = exp(ypred)

> plot(diam.valid$Price,ypred,xlab="Actual Price ($)",ylab="Predicted Price ($)",main="Predictions for Validation Diamonds")

> abline(0,1,lwd=3,col="blue")



Again we see underestimation of the prices for the most expensive diamonds in the validation set.

> PredAcc(ypred,diam.valid$Price)

RMSEP MAE MAPE

1 594.0191 374.3965 8.685138

> diam.mlr = lm(log(Price)~poly(Carats,3)+Clarity\*Color + Cut + TDdiff + TDratio,data=diam.train)

> ypred = predict(diam.mlr,newdata=diam.valid)

> ypred = exp(ypred)

> PredAcc(ypred,diam.valid$Price)

RMSEP

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

493.787

MAE

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

333.2352

MAPE

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

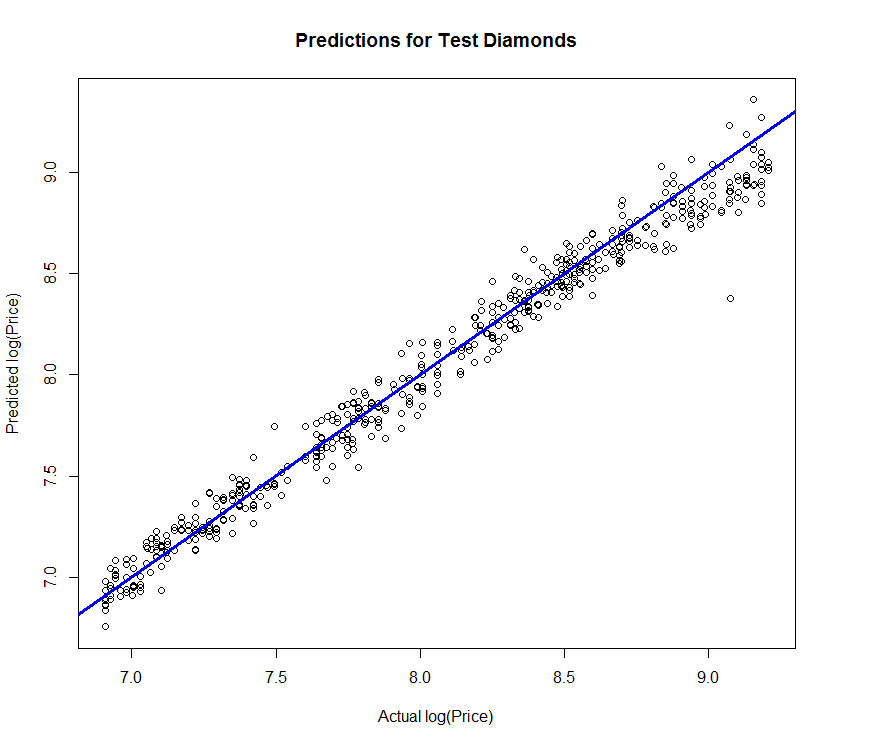
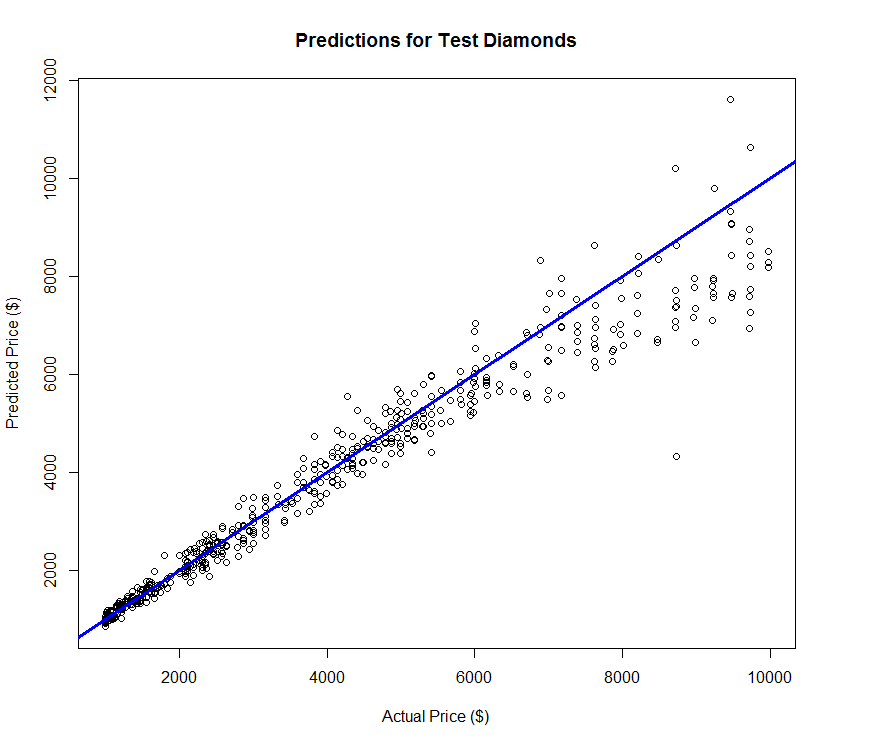
8.220324

Predicting for the test cases and comparing.

> ypred = predict(diam.gbm,newdata=diam.test,n.trees=40923)

> plot(log(diam.test$Price),ypred,xlab="Actual log(Price)",ylab="Predicted log(Price)",main="Predictions for Test Diamonds")

> abline(0,1,lwd=3,col="blue")

> ypred = exp(ypred)

> plot(diam.test$Price,ypred,xlab="Actual Price ($)",ylab="Predicted Price ($)",main="Predictions for Test Diamonds")

> abline(0,1,lwd=3,col="blue")

From Boosted Tree

For these data, it seems that the MLR model is superior to models using common predictive analytic tools.

> PredAcc(diam.test$Price,ypred)

RMSEP MAE MAPE

1 599.8503 354.8801 7.621739

From MLR model

> PredAcc(diam.test$Price,ypred)

RMSEP MAE MAPE

1 504.8733 326.18 7.953477

**Boosted Tree Example: Solubility Data**

For comparison purpose I first fit a stepwise reduced MLR model using all 228 of the predictors.

The stepwise reduction process takes a LONG time in R so don’t run this code. We also fit our “best” random forest model, and then develop a boosted tree model and compare performance in predicting the of the test chemicals.

> Solu.train = read.table(file.choose(),header=T,sep=”,”)

> Solu.test = read.table(file.choose(),header=T,sep=”,”)

**Multiple Linear Regression**

> solu.mlr = lm(log10sol~.,data=Solu.train)

> solu.step = step(solu.mlr) 🡨 takes a long time!

> ypred = predict(solu.step,newdata=Solu.test)

> PredAcc2(Solu.test$log10sol,ypred)

RMSEP

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

0.7949144

MAE

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

0.5849546

**Random Forest**

> library(randomForest)

> solu.rf = randomForest(log10sol~.,data=Solu.train,mtry=80,ntree=500)

> solu.rf

Call:

randomForest(formula = log10sol ~ ., data = Solu.train, mtry = 80, ntree = 500)

Type of random forest: regression

Number of trees: 500

No. of variables tried at each split: 80

Mean of squared residuals: 0.4222973

% Var explained: 89.91

> ypred = predict(solu.rf,newdata=Solu.test)

> PredAcc2(Solu.test$log10sol,ypred)

RMSEP

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

0.6475402

MAE

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

0.4561452

**Boosted Trees**   
(Note: Did not fine tune this model much, just tried a few settings for n.trees, shrinkage, & interaction.depth)

> library(gbm)

> solu.gbm = gbm(log10sol~.,data=Solu.train,distribution="gaussian",

n.trees=10000,shrinkage=.05,interaction.depth=5,bag.fraction=0.5,train.fraction=.8,n.minobsinnode=5,cv.folds=5,keep.data=T,verbose=F)

> solu.gbm

gbm(formula = log10sol ~ ., distribution = "gaussian", data = Solu.train,

n.trees = 4000, interaction.depth = 7, n.minobsinnode = 5,

shrinkage = 0.025, bag.fraction = 0.5, train.fraction = 0.8,

cv.folds = 5, keep.data = T, verbose = F)

A gradient boosted model with gaussian loss function.

10000 iterations were performed.

The best cross-validation iteration was 769.

The best test-set iteration was 1896.

> ypred = predict(solu.gbm,newdata=Solu.test,n.trees=769)

> PredAcc2(Solu.test$log10sol,ypred)

> PredAcc2(Solu.test$log10sol,ypred)

RMSEP

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

0.6209719

MAE

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

0.4435755

For these data the boosted tree outperforms the other modeling methods we have considered.

**Treed Regression**

In Treed Regression a tree is grown *where* the terminal nodes contain a traditional model. For example, in each terminal node we might fit an MLR regression model to the observations in that node.

There are two packages in R that perform treed regression, Cubist and party. We will use the implementation in Cubist for regression problems where the response (*Y*) is numeric. The basic function for fitting a treed MLR regression model is cubist().

cubist(x, y,

committees = 1, neighbors=0,

control = cubistControl(), ...)

**Arguments**

|  |  |
| --- | --- |
| x | a matrix or data frame of predictor variables. Missing data are allowed but (at this time) only numeric, character and factor values are allowed. |
| y | a numeric vector of outcome |
| committees | an integer: how many committee models (e.g.. boosting iterations) should be used? |
| **control** | options that control details of the cubist algorithm. See [cubistControl](http://127.0.0.1:14426/library/Cubist/help/cubistControl) |

neighbors number of nearest neighbors to consider in correcting the prediction (0 to 9).

The neighbors option specifies whether or not to use nearest neighbors in making predictions. The idea behind nearest-neighbors is outlined on the following page and is taken from the website [*www.rulequest.com*](http://www.rulequest.com).

For some applications, the predictive accuracy of a rule-based model can be improved by combining it with an instance-based or nearest-neighbor model. The latter predicts the target value of a new case by finding the ***k*** most similar cases in the training data, and averaging their target values.

Cubist employs an unusual method for combining rule-based and instance-based models. Cubist finds the ***k*** training cases that are "nearest" (most similar) to the case in question. Then, rather than averaging their target values directly, Cubist first adjusts these values using the rule-based model. Here's how it works:

Suppose that **x** is the case whose unknown target value is to be predicted, and is one of **x**'s nearest neighbors in the training data. The target value of **y** is known: let us call it **T()**. The rule-based model can be used to predict target values for any case, so let its predictions for **x** and  be **M(x)** and **M()** respectively. The model then predicts that the difference between the target values of **x** and  is **M(x)-M()**. The value of **x** predicted by neighbor  is adjusted to reflect this difference, so that Cubist uses **T()+M(x)-M()** instead of 's raw target value.   
  
The neighbors option instructs Cubist to use composite models of this type. Now for the value of ***k***, the number of nearest neighbors to be used the allowable range is from 0 to 9. We can use cross-validation to choose “optimal” values for the number of committees and the number of nearest-neighbors to use.

We now consider the usual examples.

**Cubist Example: Diamond Prices**

Again using the training, validation, and test sets for these data we will use treed regression and hopefully arrive at a model that outperforms MLR and the other tree-based methods.

> library(Cubist)

> names(diam.train)

[1] "Price" "Carats" "Color" "Clarity" "Depth" "Table" "Cut" "TDdiff"

[9] "TDratio"

> X = diam.train[,-1]

> y = log(diam.train$Price)

> diam.cubist = cubist(X,y)

> summary(diam.cubist)

Call:

cubist.default(x = X, y = y)

Cubist [Release 2.07 GPL Edition] Tue May 17 12:35:11 2016

---------------------------------

Target attribute `outcome'

Read 1613 cases (9 attributes) from undefined.data

Model:

Rule 1: [140 cases, mean 7.089416, range 6.907755 to 7.471363, est err 0.070827]

if

Carats <= 0.48

Clarity in {IF, VVS1, VVS2}

then

outcome = 5.891322 + 1.89 Carats + 0.008 Table

Rule 2: [109 cases, mean 7.500155, range 6.908755 to 8.781095, est err 0.081048]

if

Carats <= 0.94

Color in {D, E, F, G}

Clarity = VS1

then

outcome = 7.403948 + 3.25 Carats - 0.019 Depth - 0.009 Table

Rule 3: [55 cases, mean 7.505598, range 6.944087 to 8.189799, est err 0.145368]

if

Carats <= 0.94

Color in {J, K}

then

outcome = 17.16552 + 2.69 Carats - 11.4 TDratio + 0.182 Table

- 0.188 Depth

Rule 4: [29 cases, mean 7.540391, range 6.925595 to 8.289289, est err 0.074009]

if

Carats <= 0.94

Color = G

Clarity in {VS1, VS2}

then

outcome = 5.73178 + 2.84 Carats

Rule 5: [66 cases, mean 7.578313, range 6.907755 to 8.251143, est err 0.085354]

if

Carats <= 0.94

Color in {H, I}

Clarity in {SI1, SI2}

then

outcome = 43.453942 - 37.7 TDratio + 0.603 Table + 2.92 Carats

- 0.609 Depth

Rule 6: [74 cases, mean 7.611935, range 6.908755 to 8.375169, est err 0.111157]

if

Carats <= 0.94

Color in {D, E, F, G}

Clarity = SI2

then

outcome = 6.284961 + 2.82 Carats - 0.035 Table + 1.4 TDratio

Rule 7: [77 cases, mean 7.628675, range 6.907755 to 8.508152, est err 0.085406]

if

Carats <= 0.94

Color in {D, E, F}

Clarity = VS2

then

outcome = 6.35672 + 3.03 Carats - 0.01 Depth

Rule 8: [24 cases, mean 7.664202, range 6.908755 to 8.189799, est err 0.078264]

if

Carats <= 0.94

Color = I

Clarity in {VS1, VS2}

then

outcome = 5.724354 + 2.6 Carats

Rule 9: [106 cases, mean 7.668017, range 6.908755 to 8.534837, est err 0.087156]

if

Carats <= 0.94

Color in {D, E, F, G}

Clarity = SI1

then

outcome = 6.579948 + 3.04 Carats - 0.012 Depth - 0.3 TDratio

Rule 10: [58 cases, mean 7.750666, range 6.926577 to 8.574707, est err 0.134884]

if

Carats <= 0.94

Color in {H, I, J, K}

Clarity in {IF, VVS1, VVS2}

then

outcome = 7.071347 + 2.26 Carats - 0.041 Table + 1.6 TDratio

Rule 11: [31 cases, mean 7.855024, range 7.065613 to 8.408717, est err 0.076388]

if

Carats <= 0.94

Color = H

Clarity in {VS1, VS2}

then

outcome = 5.754416 + 2.76 Carats

Rule 12: [64 cases, mean 7.951024, range 7.395721 to 8.85438, est err 0.097094]

if

Carats > 0.48

Carats <= 0.94

Color in {F, G}

Clarity in {IF, VVS1, VVS2}

then

outcome = 6.175356 + 2.74 Carats

Rule 13: [48 cases, mean 7.966398, range 7.520235 to 9.207937, est err 0.063216]

if

Carats > 0.48

Color in {D, E}

Clarity = VVS2

then

outcome = 6.491923 + 2.97 Carats - 0.016 Table + 0.6 TDratio

Rule 14: [51 cases, mean 8.077394, range 7.539559 to 9.157361, est err 0.117837]

if

Carats > 0.48

Color in {D, E}

Clarity in {IF, VVS1}

then

outcome = -72.608222 + 89 TDratio - 1.471 Table + 1.303 Depth

+ 3.03 Carats

Rule 15: [20 cases, mean 8.315283, range 7.936303 to 9.128805, est err 0.088189]

if

Carats > 0.94

Color = K

Clarity in {SI1, SI2}

then

outcome = 6.778476 + 1.16 Carats - 0.3 TDratio + 0.005 Table

Rule 16: [59 cases, mean 8.447293, range 7.990238 to 9.208739, est err 0.076435]

if

Carats > 0.94

Color = J

Clarity in {SI1, SI2}

then

outcome = 8.11659 + 1.18 Carats - 0.4 TDratio - 0.012 Depth

Rule 17: [45 cases, mean 8.486957, range 7.991592 to 9.130215, est err 0.069873]

if

Carats > 0.94

Color = I

Clarity = SI2

then

outcome = 22.479786 - 15.1 TDratio + 0.23 TDdiff + 1.32 Carats

+ 0.014 Table - 0.022 Depth

Rule 18: [113 cases, mean 8.552159, range 7.990238 to 9.208639, est err 0.068193]

if

Carats > 0.94

Color in {F, G, H}

Clarity = SI2

then

outcome = 10.539928 + 1.41 Carats - 2.6 TDratio + 0.037 Table

- 0.054 Depth

Rule 19: [31 cases, mean 8.580795, range 8.229777 to 9.043577, est err 0.095577]

if

Carats > 0.94

Color in {J, K}

Clarity = VS2

then

outcome = 7.169456 + 1.08 Carats

Rule 20: [40 cases, mean 8.580815, range 8.113426 to 9.21014, est err 0.051344]

if

Carats > 0.94

Color = I

Clarity = SI1

then

outcome = 8.038007 + 1.33 Carats - 0.7 TDratio - 0.015 Depth

+ 0.009 Table

Rule 21: [73 cases, mean 8.637917, range 8.251143 to 9.182661, est err 0.063547]

if

Carats > 0.94

Color in {G, H}

Clarity = SI1

then

outcome = 8.245455 + 1.39 Carats - 0.7 TDratio - 0.015 Depth

+ 0.007 Table

Rule 22: [35 cases, mean 8.644493, range 8.250098 to 9.20994, est err 0.076217]

if

Carats > 0.94

Color in {D, E}

Clarity = SI2

then

outcome = 11.23684 + 1.34 Carats - 2.8 TDratio + 0.036 Table

- 0.058 Depth

Rule 23: [48 cases, mean 8.645463, range 8.210668 to 9.128913, est err 0.089350]

if

Carats > 0.94

Color in {J, K}

Clarity in {IF, VS1, VVS1, VVS2}

then

outcome = 13.094335 + 1.26 Carats - 5.1 TDratio + 0.051 TDdiff

+ 0.025 Table - 0.04 Depth

Rule 24: [49 cases, mean 8.689831, range 8.288786 to 9.156306, est err 0.090027]

if

Carats > 0.94

Color in {D, E, F}

Clarity = SI1

then

outcome = 11.230623 + 1.28 Carats - 0.063 Depth - 0.021 TDdiff

- 1.1 TDratio + 0.016 Table

Rule 25: [47 cases, mean 8.788722, range 8.361007 to 9.21034, est err 0.115292]

if

Carats > 0.94

Color = I

Clarity in {IF, VS1, VS2, VVS1, VVS2}

then

outcome = 28.629283 - 21.4 TDratio + 0.323 TDdiff + 1.19 Carats

Rule 26: [40 cases, mean 8.842575, range 8.423542 to 9.20914, est err 0.125265]

if

Carats > 0.94

Color = H

Clarity in {IF, VS1, VS2, VVS1, VVS2}

then

outcome = 18.950073 - 10.7 TDratio + 0.99 Carats + 0.147 Table

- 0.157 Depth

Rule 27: [59 cases, mean 8.946100, range 8.701679 to 9.208639, est err 0.083272]

if

Carats > 0.94

Color in {D, E, F, G}

Clarity = VS2

then

outcome = 12.115267 + 0.75 Carats - 3.7 TDratio + 0.059 Table

- 0.064 Depth

Rule 28: [56 cases, mean 9.009788, range 8.698681 to 9.208739, est err 0.113267]

if

Carats > 0.94

Color in {D, E, F, G}

Clarity in {IF, VS1, VVS1, VVS2}

then

outcome = 24.890775 - 15.4 TDratio + 0.115 TDdiff + 0.75 Carats

+ 0.124 Table - 0.145 Depth

Evaluation on training data (1613 cases):

Average |error| 0.088325

Relative |error| 0.15

Correlation coefficient 0.98

Attribute usage:

Conds Model

100% 100% Carats

97% Clarity

91% Color

72% Table

69% TDratio

66% Depth

15% TDdiff

Time: 0.0 secs

> Xvalid = diam.valid[,-1]

> ypredlog = predict(diam.cubist,newdata=Xvalid)

> ypred = exp(ypredlog)

> PredAcc(diam.valid$Price,ypred)

RMSEP

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

517.5964

MAE

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

332.3824

MAPE

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

8.301016

Using the default settings we already competitive with other methods we have examined for these data. We know flex the muscles a bit by using boosting and nearest neighbors.

> diam.cubist2 = cubist(X,y,committees=10)

> ypred = predict(diam.cubist2,newdata=X)  
> plot(log(diam.train$Price),ypred,xlab="Actual log(Price)",ylab="Predict log(Price)")

> abline(0,1,lwd=3,col="blue")

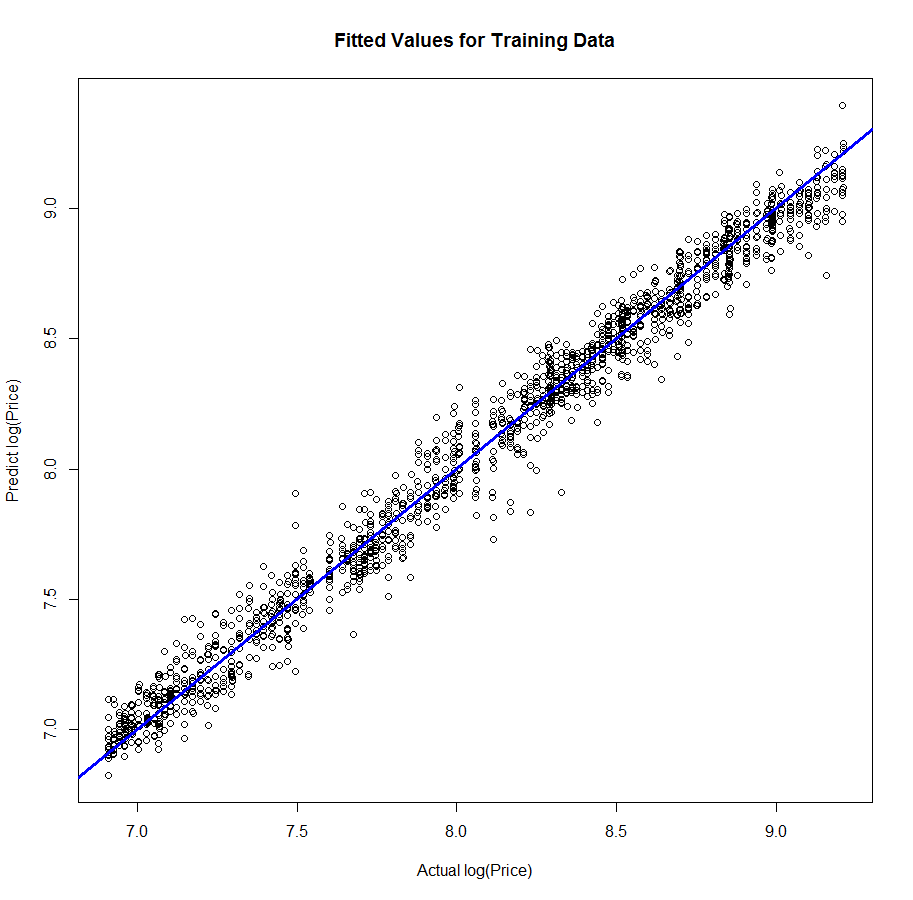
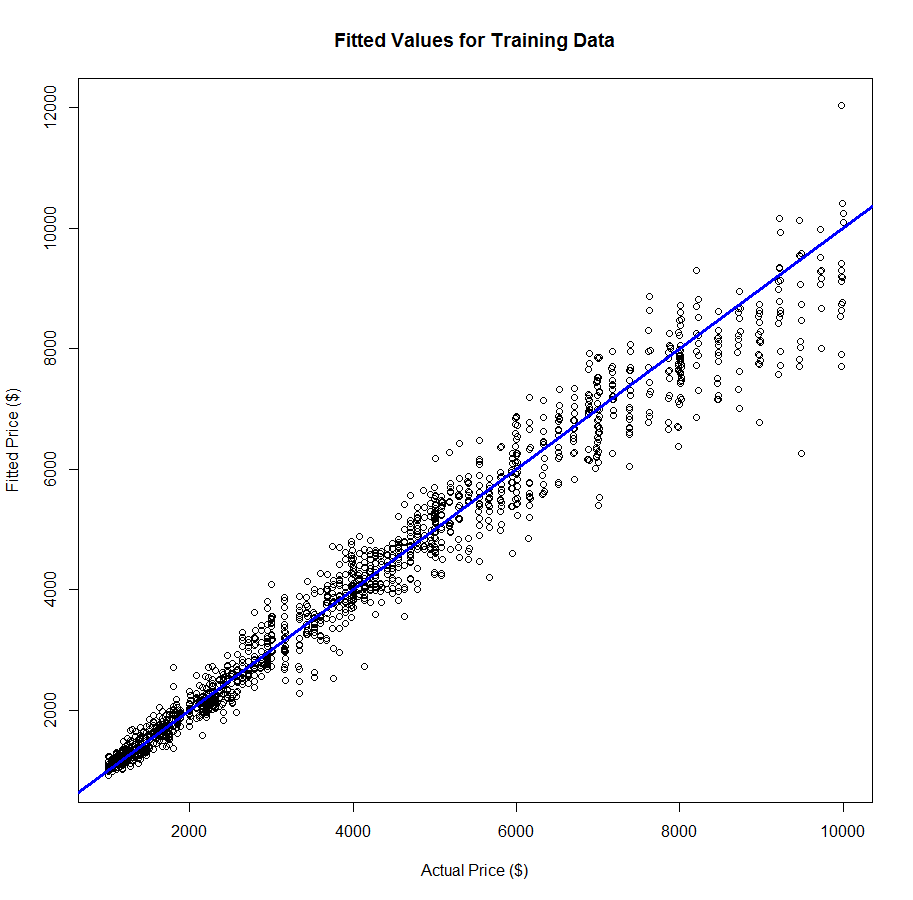
> title(main="Fitted Values for Training Data")

> ypred = predict(diam.cubist2,newdata=X)

> ypred = exp(ypred)

> plot(diam.train$Price,ypred,xlab="Actual Price ($)",ylab="Fitted Price ($)”)

> abline(0,1,lwd=3,col="blue")  
> title(main="Fitted Values for Training Data")

> Xvalid = diam.valid[,-1]

> ypredlog = predict(diam.cubist2,newdata=Xvalid)

> ypred = exp(ypredlog)

> PredAcc(diam.valid$Price,ypred)

RMSEP

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

458.7293

MAE

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

305.9408

MAPE

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

7.598628

This I believe is the best predictions we have had for the validation diamonds. We can still potentially improve our predictions by adjusting our predictions using the nearest neighbors approach.

Using nearest neighbors to enhance our predictions.

> ypredlog = predict(diam.cubist2,newdata=Xvalid,neighbors=9)

> ypred = exp(ypredlog)

> PredAcc(diam.valid$Price,ypred)

RMSEP

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

430.3995

MAE

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

285.4572

MAPE

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

7.179426

Definitely the best yet.

**Task**

1. Build a Cubist model for the chemical solubility data.

library(Cubist)

Solu.train = read.table(file.choose(),header=T,sep=”,”)

Solu.test = read.table(file.choose(),header=T,sep=”,”)

Xtrain = Solu.train[,-1]

Ytrain = Solu.train[,1]

Xtest = Solu.test[,-1]  
 ytest = Solu.test[,1]

solu.cub = cubist(Xtrain,ytrain,*extra stuff*)

summary(solu.cub)

ypred = predict(solu.cub,newdata=Xtest,*extra stuff*)

PredAcc(ytest,ypred)

How does the Cubist model compare to our results with these data above?