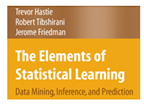
**10.6 - Boosting Regression Trees (R package –** gbm**)**

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.

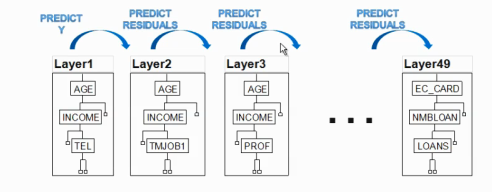
1. Fit a regression tree using as the response, giving terminal node regions .
2. For compute the mean of the residuals in each of the terminal nodes, call these
3. Update the model as follows:

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 .

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.

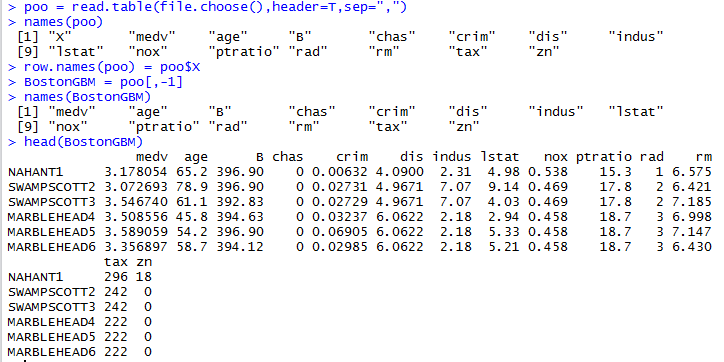
keep.data = TRUE,

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 ☺.

We will now consider some examples.

**Example 10.7: Boston Housing Data (BostonGBM.csv on website)**

Notes: The names of the census tracts are stored as row names for the data frame. They are containing in the variable X upon reading in the *.csv* file. The response (medv) has been log-transformed.

> bos.gbm = gbm(medv~.,data=BostonGBM,

+ distribution="gaussian",

+ n.trees=1000,

+ shrinkage=.001,

+ interaction.depth=2,   
+ bag.fraction=0.5,   
+ train.fraction=0.8,   
+ n.minobsinnode=10,

+ cv.folds=5,

+ keep.data=T,

+ verbose=T)

> bos.gbm = gbm(medv~.,data=BostonGBM,distribution="gaussian",

n.trees=2000,shrinkage=.1,interaction.depth=4,bag.fraction=.5,

train.fraction=.8,n.minobsinnode=10,cv.folds=5,keep.data=T,verbose=T)  
Iter TrainDeviance ValidDeviance StepSize Improve

1 0.1247 0.2539 0.1000 0.0217

2 0.1077 0.2112 0.1000 0.0153

3 0.0951 0.1894 0.1000 0.0120

4 0.0831 0.1707 0.1000 0.0105

5 0.0731 0.1504 0.1000 0.0080

6 0.0640 0.1315 0.1000 0.0079

7 0.0568 0.1161 0.1000 0.0070

8 0.0513 0.1067 0.1000 0.0048

9 0.0463 0.1049 0.1000 0.0041

10 0.0421 0.0983 0.1000 0.0032

20 0.0226 0.0698 0.1000 0.0007

40 0.0136 0.0640 0.1000 -0.0001

60 0.0104 0.0647 0.1000 -0.0000

80 0.0089 0.0619 0.1000 -0.0001

100 0.0076 0.0641 0.1000 -0.0000

120 0.0065 0.0649 0.1000 -0.0001

140 0.0058 0.0646 0.1000 -0.0001

160 0.0051 0.0632 0.1000 -0.0000

180 0.0046 0.0637 0.1000 -0.0000

200 0.0041 0.0644 0.1000 -0.0000

220 0.0037 0.0643 0.1000 -0.0001

240 0.0033 0.0658 0.1000 -0.0000

260 0.0030 0.0657 0.1000 -0.0000

280 0.0028 0.0665 0.1000 -0.0000

300 0.0026 0.0670 0.1000 -0.0000

320 0.0024 0.0666 0.1000 -0.0000

340 0.0022 0.0676 0.1000 -0.0000

360 0.0021 0.0672 0.1000 -0.0000

380 0.0019 0.0681 0.1000 -0.0000

400 0.0018 0.0674 0.1000 -0.0000

420 0.0017 0.0690 0.1000 -0.0000

440 0.0015 0.0682 0.1000 -0.0000

460 0.0014 0.0689 0.1000 -0.0000

480 0.0014 0.0688 0.1000 -0.0000

500 0.0013 0.0693 0.1000 -0.0000

520 0.0012 0.0686 0.1000 -0.0000

540 0.0011 0.0691 0.1000 -0.0000

560 0.0011 0.0697 0.1000 -0.0000

580 0.0010 0.0696 0.1000 -0.0000

600 0.0010 0.0681 0.1000 -0.0000

620 0.0009 0.0685 0.1000 -0.0000

640 0.0009 0.0690 0.1000 -0.0000

660 0.0008 0.0687 0.1000 -0.0000

680 0.0008 0.0695 0.1000 -0.0000

700 0.0007 0.0705 0.1000 -0.0000

720 0.0007 0.0707 0.1000 -0.0000

740 0.0006 0.0704 0.1000 -0.0000

760 0.0006 0.0702 0.1000 -0.0000

780 0.0006 0.0704 0.1000 -0.0000

800 0.0006 0.0705 0.1000 -0.0000

820 0.0005 0.0710 0.1000 -0.0000

840 0.0005 0.0706 0.1000 -0.0000

860 0.0005 0.0709 0.1000 -0.0000

880 0.0005 0.0706 0.1000 -0.0000

900 0.0004 0.0707 0.1000 -0.0000

920 0.0004 0.0702 0.1000 -0.0000

940 0.0004 0.0709 0.1000 -0.0000

960 0.0004 0.0713 0.1000 -0.0000

980 0.0004 0.0712 0.1000 -0.0000

1000 0.0003 0.0715 0.1000 -0.0000

1020 0.0003 0.0716 0.1000 -0.0000

1040 0.0003 0.0715 0.1000 -0.0000

1060 0.0003 0.0716 0.1000 -0.0000

1080 0.0003 0.0715 0.1000 -0.0000

1100 0.0003 0.0717 0.1000 -0.0000

1120 0.0003 0.0718 0.1000 -0.0000

1140 0.0002 0.0718 0.1000 -0.0000

1160 0.0002 0.0718 0.1000 -0.0000

1180 0.0002 0.0719 0.1000 -0.0000

1200 0.0002 0.0718 0.1000 -0.0000

1220 0.0002 0.0719 0.1000 -0.0000

1240 0.0002 0.0722 0.1000 -0.0000

1260 0.0002 0.0722 0.1000 -0.0000

1280 0.0002 0.0722 0.1000 -0.0000

1300 0.0002 0.0716 0.1000 -0.0000

1320 0.0002 0.0718 0.1000 -0.0000

1340 0.0002 0.0717 0.1000 -0.0000

1360 0.0001 0.0716 0.1000 -0.0000

1380 0.0001 0.0718 0.1000 -0.0000

1400 0.0001 0.0716 0.1000 -0.0000

1420 0.0001 0.0718 0.1000 -0.0000

1440 0.0001 0.0717 0.1000 -0.0000

1460 0.0001 0.0717 0.1000 -0.0000

1480 0.0001 0.0718 0.1000 -0.0000

1500 0.0001 0.0717 0.1000 -0.0000

1520 0.0001 0.0718 0.1000 -0.0000

1540 0.0001 0.0718 0.1000 -0.0000

1560 0.0001 0.0719 0.1000 -0.0000

1580 0.0001 0.0719 0.1000 -0.0000

1600 0.0001 0.0719 0.1000 -0.0000

1620 0.0001 0.0719 0.1000 -0.0000

1640 0.0001 0.0719 0.1000 -0.0000

1660 0.0001 0.0717 0.1000 -0.0000

1680 0.0001 0.0717 0.1000 -0.0000

1700 0.0001 0.0718 0.1000 -0.0000

1720 0.0001 0.0721 0.1000 -0.0000

1740 0.0001 0.0719 0.1000 -0.0000

1760 0.0001 0.0721 0.1000 -0.0000

1780 0.0001 0.0721 0.1000 -0.0000

1800 0.0001 0.0722 0.1000 -0.0000

1820 0.0001 0.0723 0.1000 -0.0000

1840 0.0001 0.0723 0.1000 -0.0000

1860 0.0001 0.0722 0.1000 -0.0000

1880 0.0000 0.0721 0.1000 -0.0000

1900 0.0000 0.0721 0.1000 -0.0000

1920 0.0000 0.0723 0.1000 -0.0000

1940 0.0000 0.0721 0.1000 -0.0000

1960 0.0000 0.0721 0.1000 -0.0000

1980 0.0000 0.0722 0.1000 -0.0000

2000 0.0000 0.0722 0.1000 -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 on the pages that follow.

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

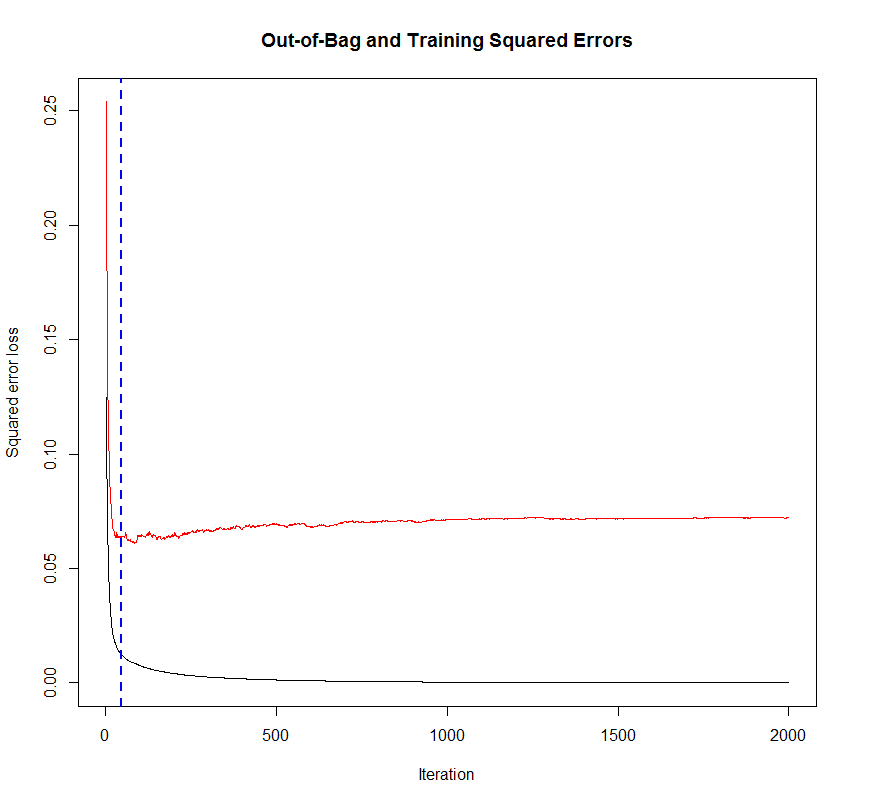
[1] 46

Warning message:

In gbm.perf(bos.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.

> title(main="Out-of-Bag and Training Squared Errors")

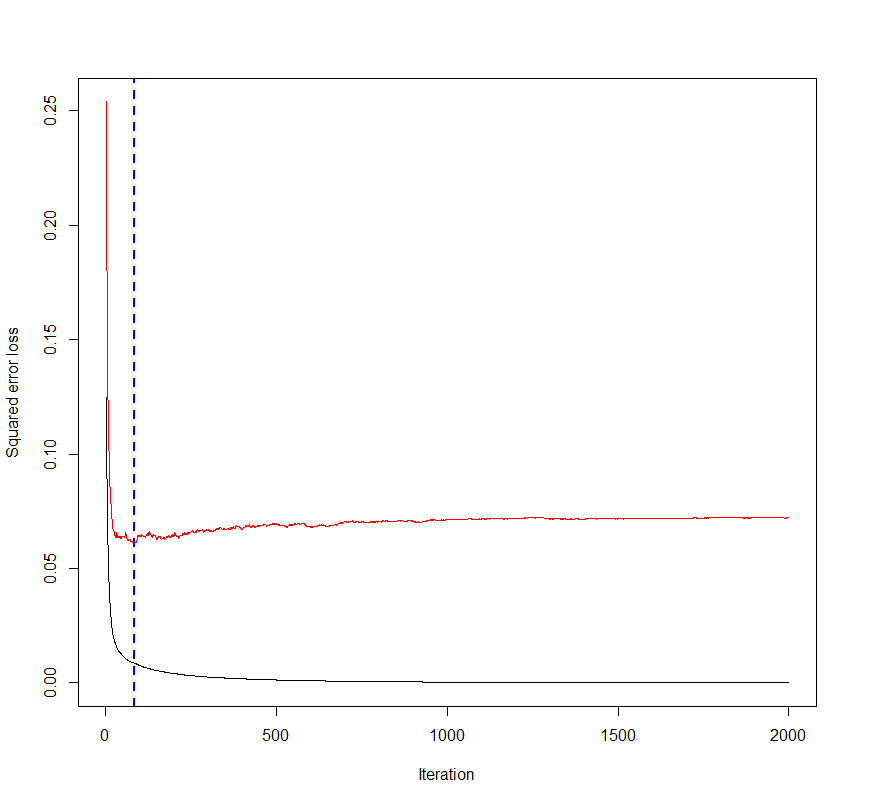


OOB error

Training error

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

[1] 83

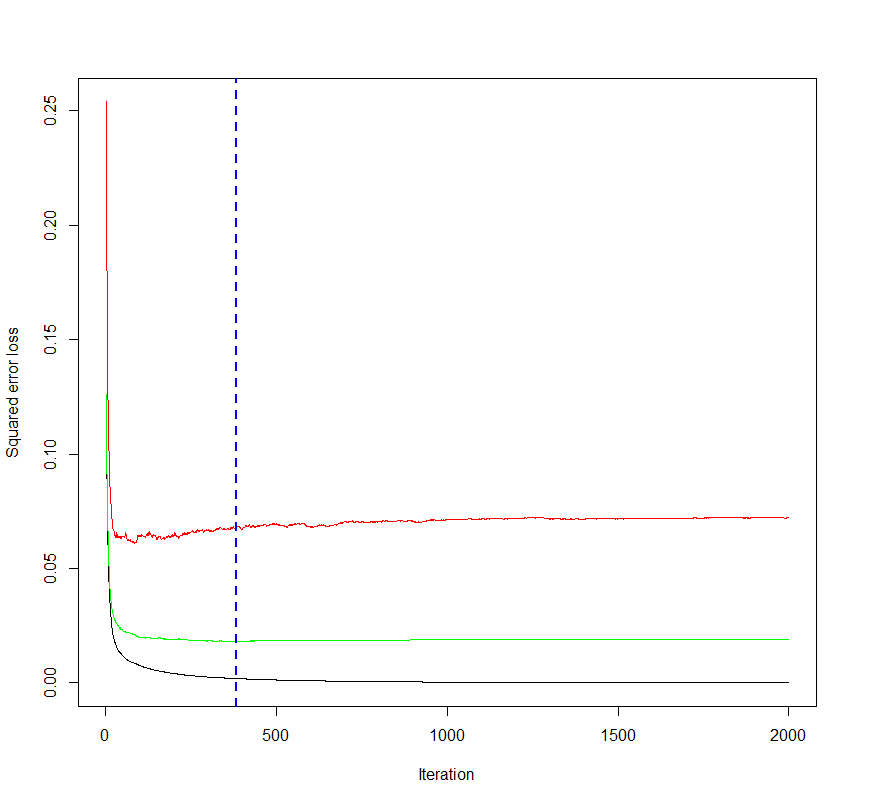


Test error

Training error

> gbm.perf(bos.gbm,method="cv")

[1] 383

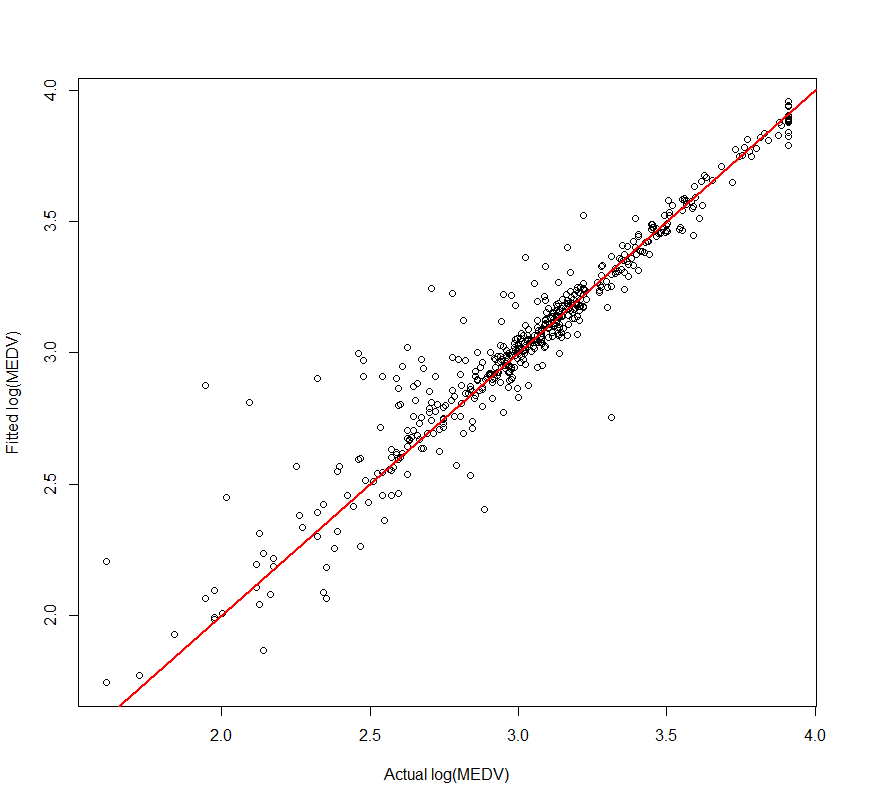


Test error  
5-fold CV error   
Training error

> yhat = predict(bos.gbm,n.trees=383) 🡨 using optimal boosting iterations from CV

> plot(BostonGBM$medv,yhat,xlab="Actual log(MEDV)",ylab="Fitted log(MEDV)”)

> abline(0,1,col=”red”,lwd=2)



> cor(BostonGBM$medv,yhat)

[1] 0.9542598

> cor(BostonGBM$medv,yhat)^2 🡨

[1] 0.9106117

Using the number of trees suggest by test/validation set cross-validation instead we use a smaller number of boosting iterations, .

> yhat = predict(bos.gbm,n.trees=83)

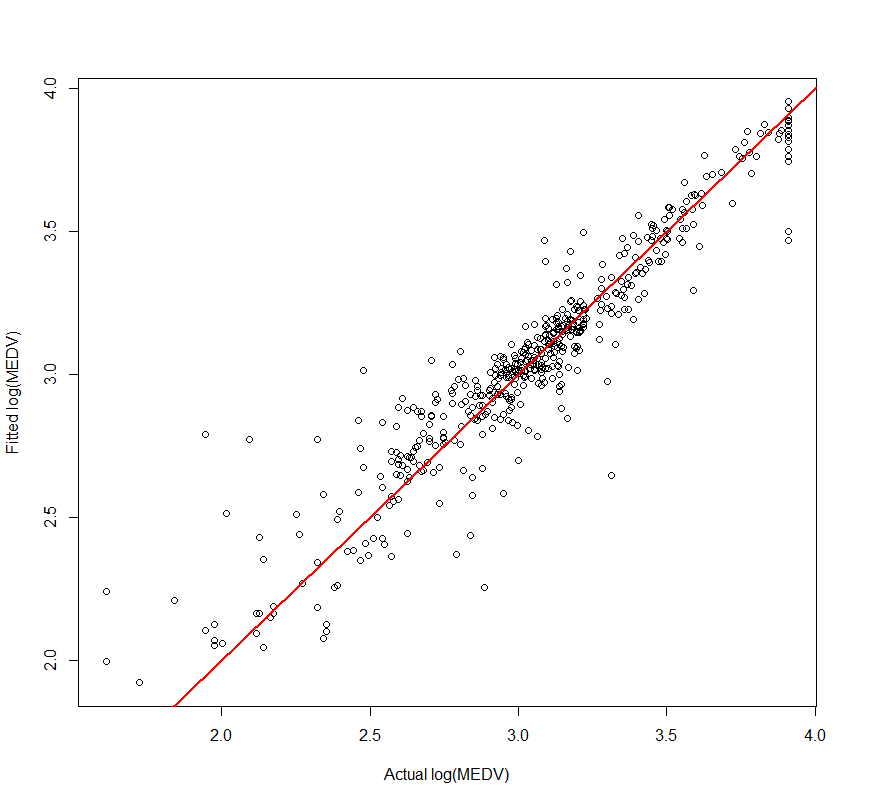
> plot(BostonGBM$medv,yhat,xlab="Actual log(MEDV)",ylab="Fitted log(MEDV)")

> cor(BostonGBM$medv,yhat)

[1] 0.9408466

> cor(BostonGBM$medv,yhat)^2

[1] 0.8851923



We can consider varying the shrinkage parameter as well. For example, we might consider changing the shrinkage to for the Boston Housing data. This requires are larger number of iterations (*M*) as evidenced below.

> bos.gbm = gbm(medv~.,data=BostonGBM,distribution="gaussian",n.trees==40000,shrinkage=.01,interaction.depth=4,bag.fraction=0.5,train.fraction=0.8,n.minobsinnode=10,cv.folds=5,keep.data=T,verbose=TRUE)  
  
> gbm.perf(bos.gbm,method="test")

[1] 613

> gbm.perf(bos.gbm,method="cv")

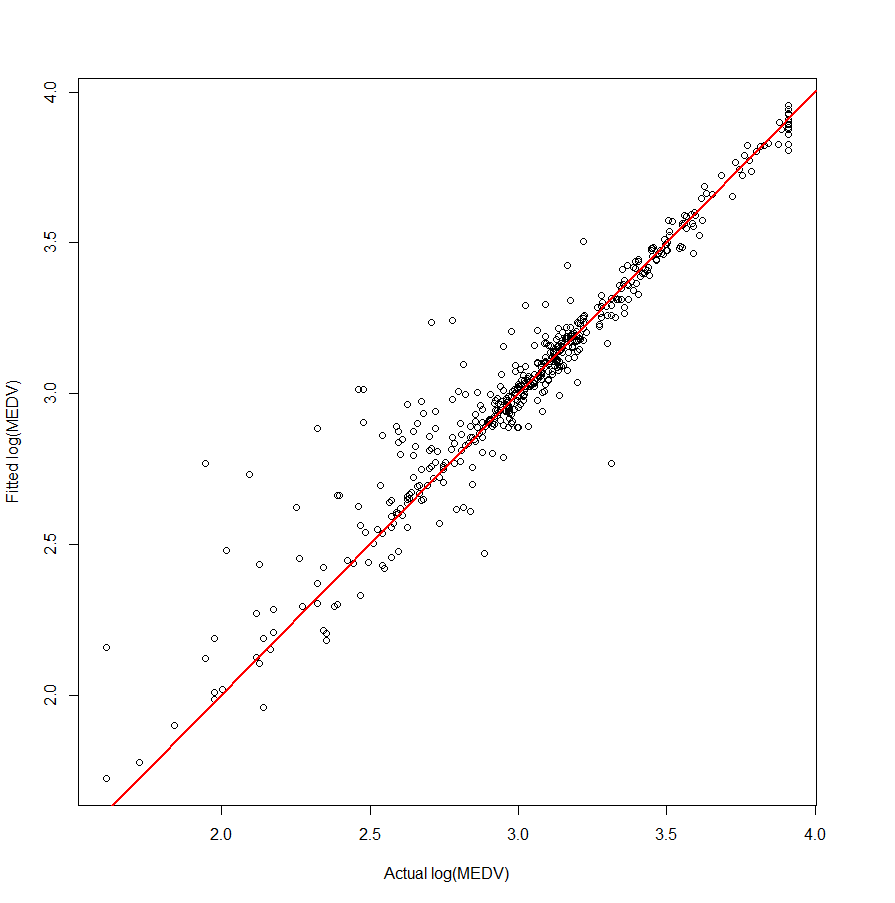
[1] 4331

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

[1] 433

> yhat = predict(bos.gbm,n.trees=4331)

> plot(BostonGBM$medv,yhat,xlab="Actual log(MEDV)",ylab="Fitted log(MEDV)")

  
> cor(BostonGBM$medv,yhat)

[1] 0.9593978

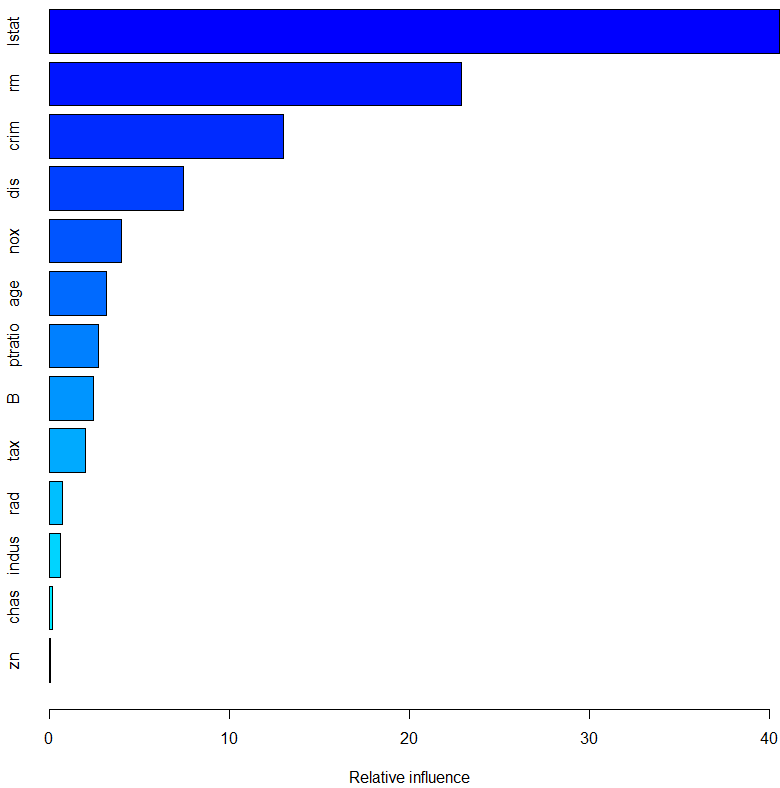
> cor(BostonGBM$medv,yhat)^2

[1] 0.9204441

We might consider trying even smaller and possibly larger values than our initial shrinkage (. If you decease then you must increase the boosting iterations (n.trees) to a very large value. For example with current settings and using the shrinkage the optimal number of iterations using 5-fold CV is 61,470!! Thus setting n.trees to a very large value (e.g. 100,000) is recommended when using small shrinkage values. In the examples above we also used trees with terminal nodes (interaction.depth=4). We might now consider increasing or decreasing the model sizes in terms of the number of terminal nodes as well. Before exploring alternative tree size and shrinkage settings, we will look at the variable importance for the predictors in our current model.

> bos.gbm = gbm(medv~.,data=BostonGBM,distribution="gaussian",n.trees=40000,  
shrinkage=.01,interaction.depth=4,bag.fraction=0.5,train.fraction=0.8,n.minobsinnode=10,cv.folds=5,keep.data=T,verbose=F)  
  
> best.iter = gbm.perf(bos.gbm,method="cv") 🡨 n.trees = 4331

> summary(bos.gbm,n.trees=best.iter)



var rel.inf

lstat lstat 40.5376394

rm rm 22.9124747

crim crim 12.9967678

dis dis 7.4882866

nox nox 4.0037487

age age 3.1909161

ptratio ptratio 2.7265925

B B 2.4553148

tax tax 2.0347182

rad rad 0.7605899

indus indus 0.6430307

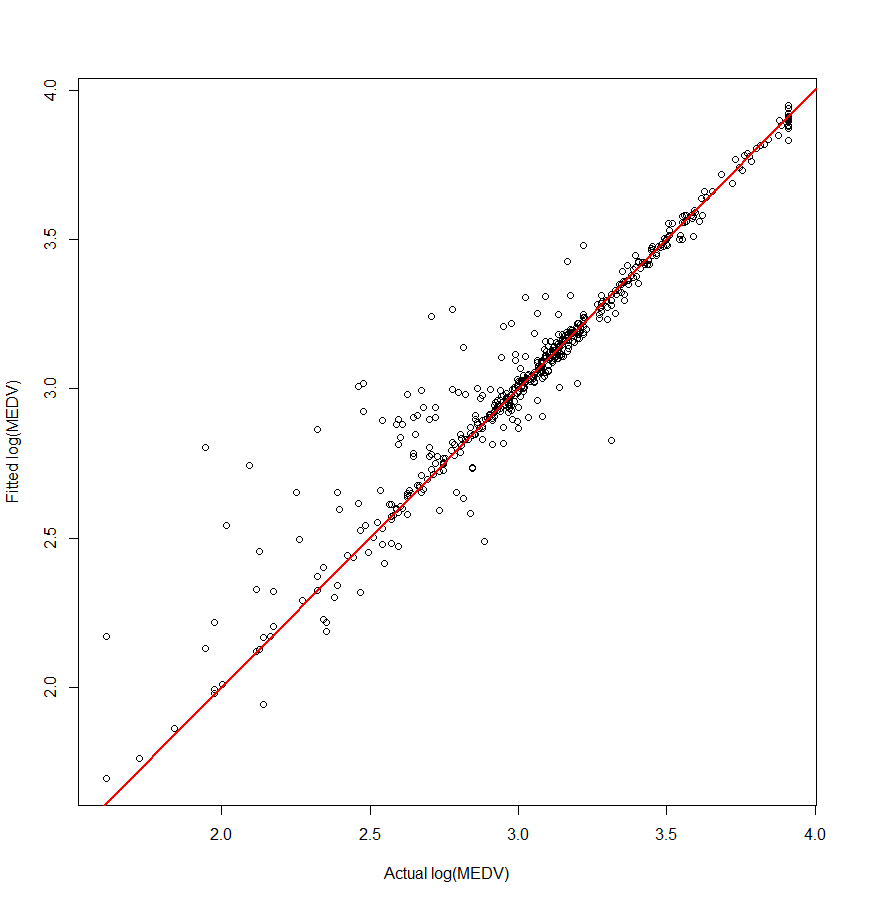
chas chas 0.1805651

zn zn 0.0693554

We can consider increasing the size of the trees being averaged through the use of the interaction depth, which determines the number of terminal nodes in each tree (*Jm* above).

Varying Interaction Depths with Shrinkage

terminal nodes (interaction.depth = 5)> bos.gbm6 = gbm(medv~.,data=BostonGBM, distribution="gaussian",  
interaction.depth=5,n.minobsinnode=10,n.trees=40000,bag.fraction=.5,  
train.fraction=.8,cv.folds=5,shrinkage=.01,verbose=F)



> gbm.perf(bos.gbm,method="cv")

[1] 5473

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

[1] 765

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

[1] 855

> yhat = predict(bos.gbm,n.trees=5473)

> plot(BostonGBM$medv,yhat,xlab="Actual log(MEDV)",ylab="Fitted log(MEDV)")

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

> cor(BostonGBM$medv,yhat)

[1] 0.9593558

> cor(BostonGBM$medv,yhat)^2

[1] 0.9203636

terminal nodes (interaction.depth = 7)> bos.gbm6 = gbm(medv~.,data=BostonGBM, distribution="gaussian",

interaction.depth=7,n.minobsinnode=10,n.trees=40000,bag.fraction=.5,

train.fraction=.8,cv.folds=5,shrinkage=.01,verbose=F)

> gbm.perf(bos.gbm,method="cv")

[1] 3599

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

[1] 442

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

[1] 855

> yhat = predict(bos.gbm,n.trees=3599)

> plot(BostonGBM$medv,yhat,xlab="Actual log(MEDV)",ylab="Fitted log(MEDV)")

> cor(BostonGBM$medv,yhat)

[1] 0.9586353

> cor(BostonGBM$medv,yhat)^2

[1] 0.9189817

Using larger trees does not appear to heading in the right direction, so we might try fitting smaller trees at each boosting iteration.

terminal nodes (interaction.depth = 3)

> bos.gbm = gbm(medv~.,data=BostonGBM,distribution="gaussian",

n.trees=40000,shrinkage=.01,interaction.depth=3,bag.fraction=0.5,

train.fraction=0.8,n.minobsinnode=10,cv.folds=5,keep.data=T,verbose=F)

> gbm.perf(bos.gbm,method="cv")

[1] 2996

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

[1] 1172

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

[1] 860  
  
> yhat = predict(bos.gbm,n.trees=2996)

> plot(BostonGBM$medv,yhat,xlab="Actual log(MEDV)",ylab="Fitted log(MEDV)")

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

> cor(BostonGBM$medv,yhat)

[1] 0.9547743

> cor(BostonGBM$medv,yhat)^2

[1] 0.911594

terminal nodes (interaction.depth = 2)

> bos.gbm = gbm(medv~.,data=BostonGBM,distribution="gaussian",n.trees=40000,  
shrinkage=.01,interaction.depth=2,bag.fraction=0.5,train.fraction=0.8,  
n.minobsinnode=10,cv.folds=5,keep.data=T,verbose=F)

> gbm.perf(bos.gbm,method="cv")

[1] 8061

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

[1] 748

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

[1] 1439

> yhat = predict(bos.gbm,n.trees=8061)

> plot(BostonGBM$medv,yhat,xlab="Actual log(MEDV)",ylab="Fitted log(MEDV)")

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

> cor(BostonGBM$medv,yhat)

[1] 0.9543036

> cor(BostonGBM$medv,yhat)^2

[1] 0.9106954

**Untransformed Response**

Surprisingly the non-transformed response model fits even better, with *Jm = 5* only.  
The R-square is 95.37% and the response is in the original scale, so no back-transforming needed when make predictions.

> BostonGBM.orig = BostonGBM

> BostonGBM.orig$medv = exp(BostonGBM$medv)

> bos.gbm2 = gbm(medv~.,data=BostonGBM.orig,distribution="gaussian",  
n.trees=40000, shrinkage=.01,interaction.depth=4,bag.fraction=0.5,  
train.fraction=0.8,n.minobsinnode=10,cv.folds=5,keep.data=T,verbose=F)

> gbm.perf(bos.gbm2,method="cv")

[1] 9647

> gbm.perf(bos.gbm2,method="test")

[1] 397

> gbm.perf(bos.gbm2,method="OOB")

[1] 1056  
  
> yhat = predict(bos.gbm2,n.trees=9647)

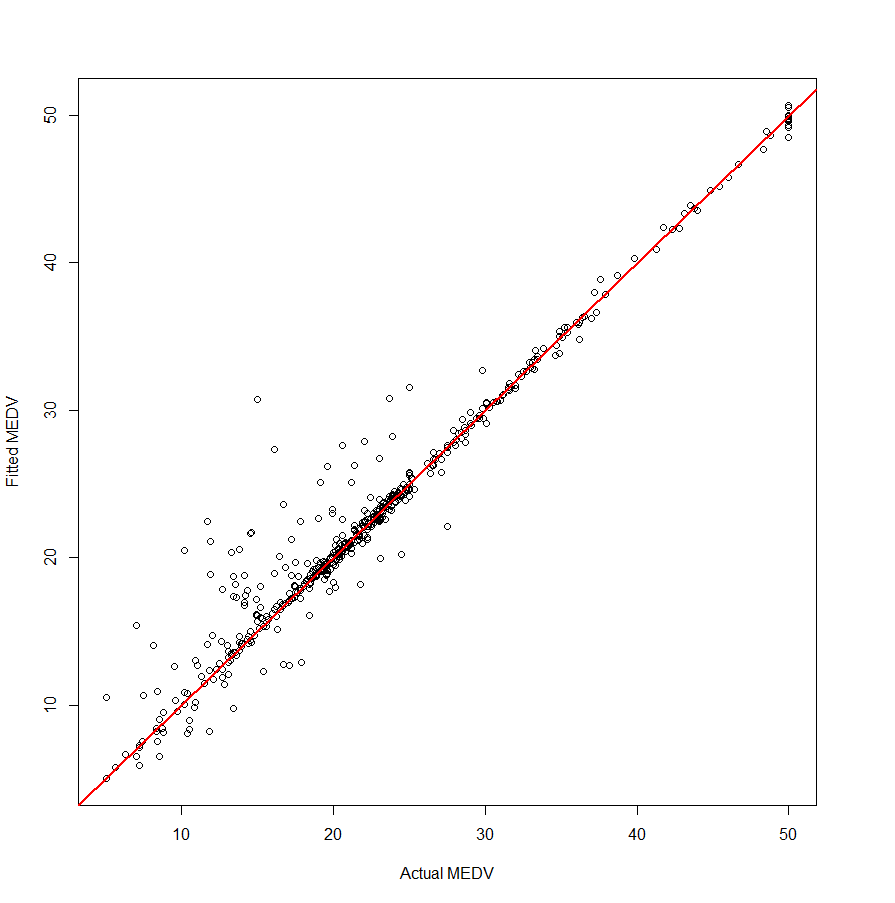
> plot(BostonGBM.orig$medv,yhat,xlab="Actual MEDV",ylab="Fitted MEDV")

> cor(BostonGBM.orig$medv,yhat)

[1] 0.9765777

> cor(BostonGBM.orig$medv,yhat)^2

[1] 0.9537041



**Predictor Importance (untransformed response)**> summary(bos.gbm2,n.trees=9647)

var rel.inf

rm rm 40.9927275

lstat lstat 27.5083713

dis dis 7.7256505

crim crim 6.7925743

age age 3.7092042

nox nox 2.8982558

B B 2.7605311

ptratio ptratio 2.6777408

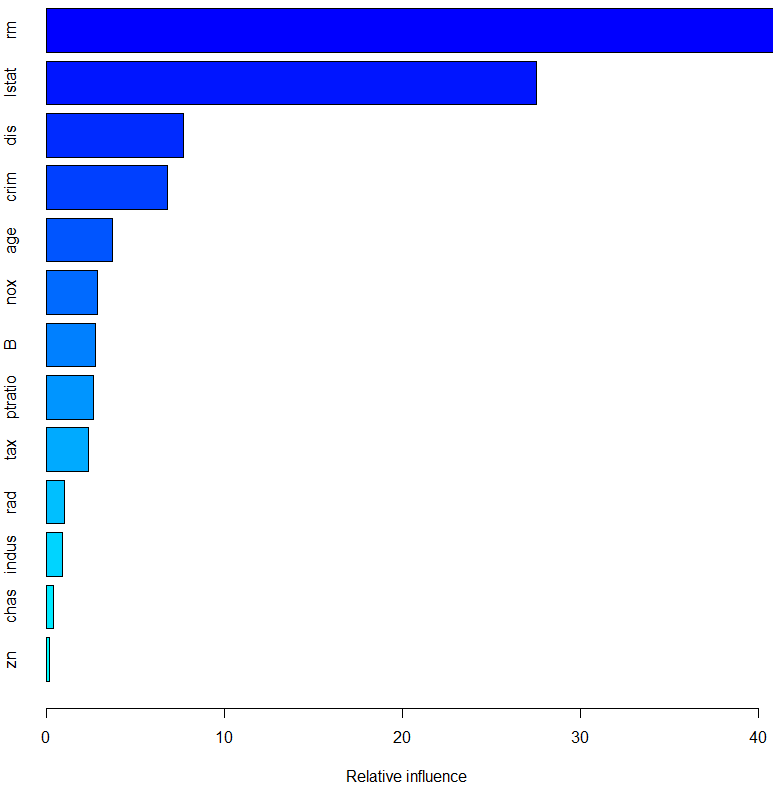
tax tax 2.3650581

rad rad 1.0169531

indus indus 0.9240594

chas chas 0.4079966

zn zn 0.2208773

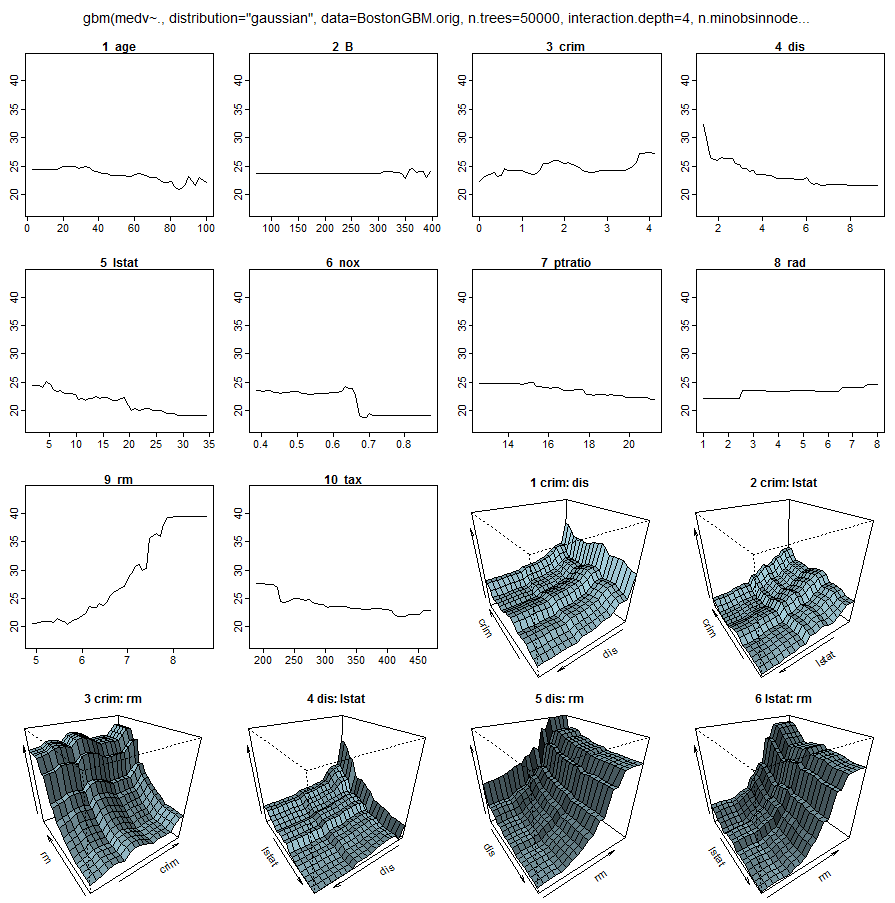


> library(plotmo)

> plotmo(bos.gbm2)

grid: age B chas crim dis indus lstat nox ptratio rad rm tax zn

66.6 392.69 0 0.1415 3.9454 6.91 9.5 0.493 17.8 4 6.266 304 0



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

> plot.gbm(bos.gbm2,1,n.trees=9647)

> plot.gbm(bos.gbm2,2,n.trees=9647)

> plot.gbm(bos.gbm2,3,n.trees=9647)

> plot.gbm(bos.gbm2,4,n.trees=9647)

> plot.gbm(bos.gbm2,5,n.trees=9647)

> plot.gbm(bos.gbm2,6,n.trees=9647)

> plot.gbm(bos.gbm2,7,n.trees=9647)

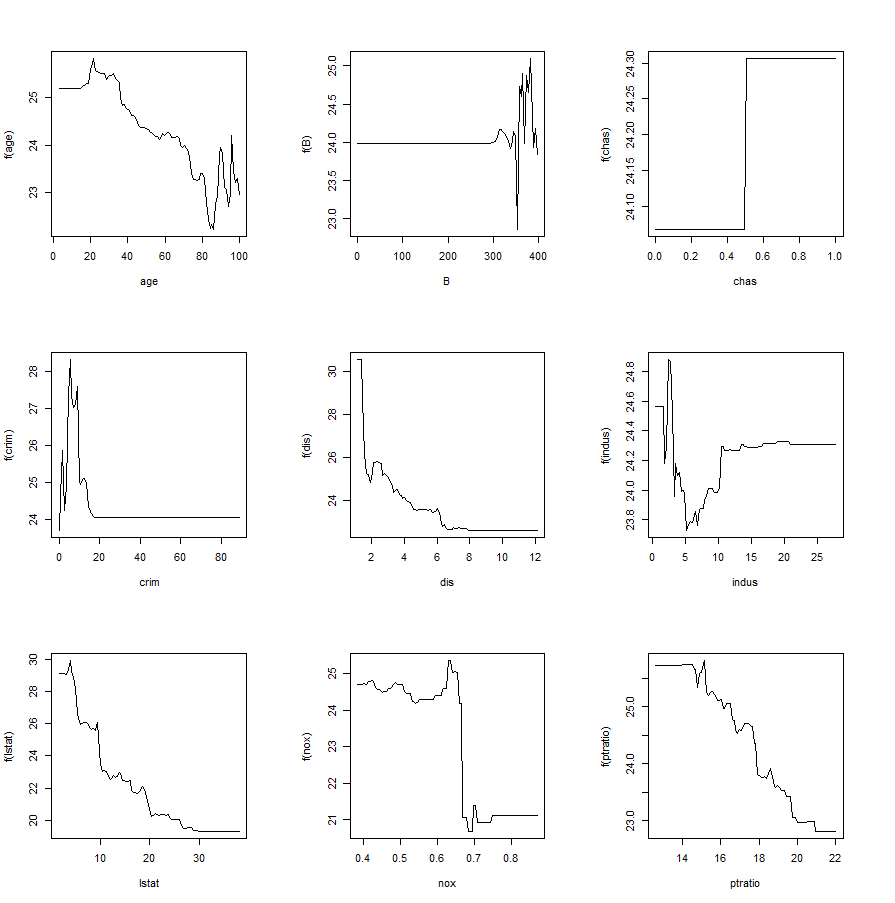
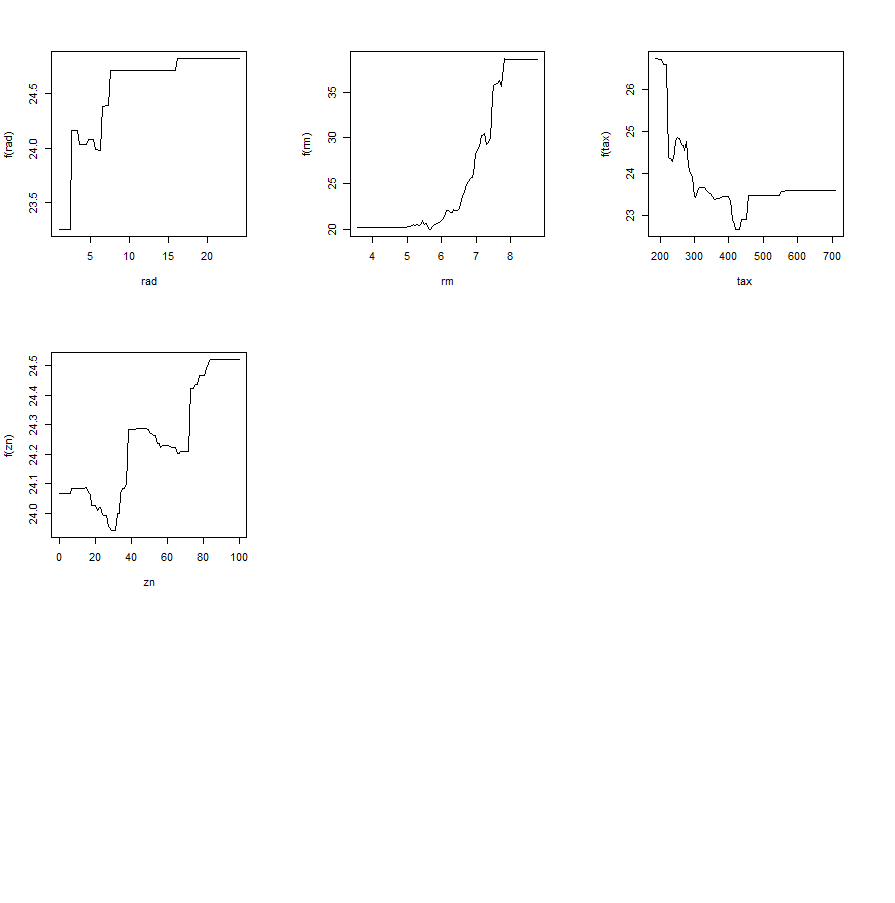
> plot.gbm(bos.gbm2,8,n.trees=9647)

> plot.gbm(bos.gbm2,9,n.trees=9647)  
> plot.gbm(bos.gbm3,10,n.trees=9647)

> plot.gbm(bos.gbm3,11,n.trees=9647)

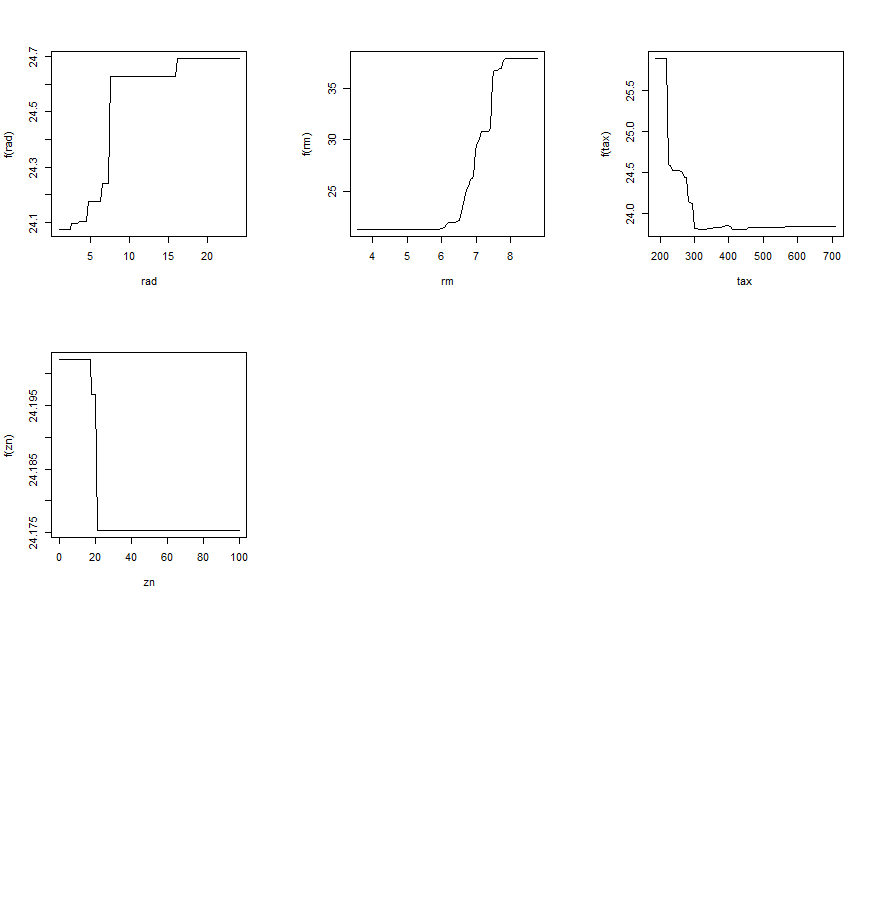
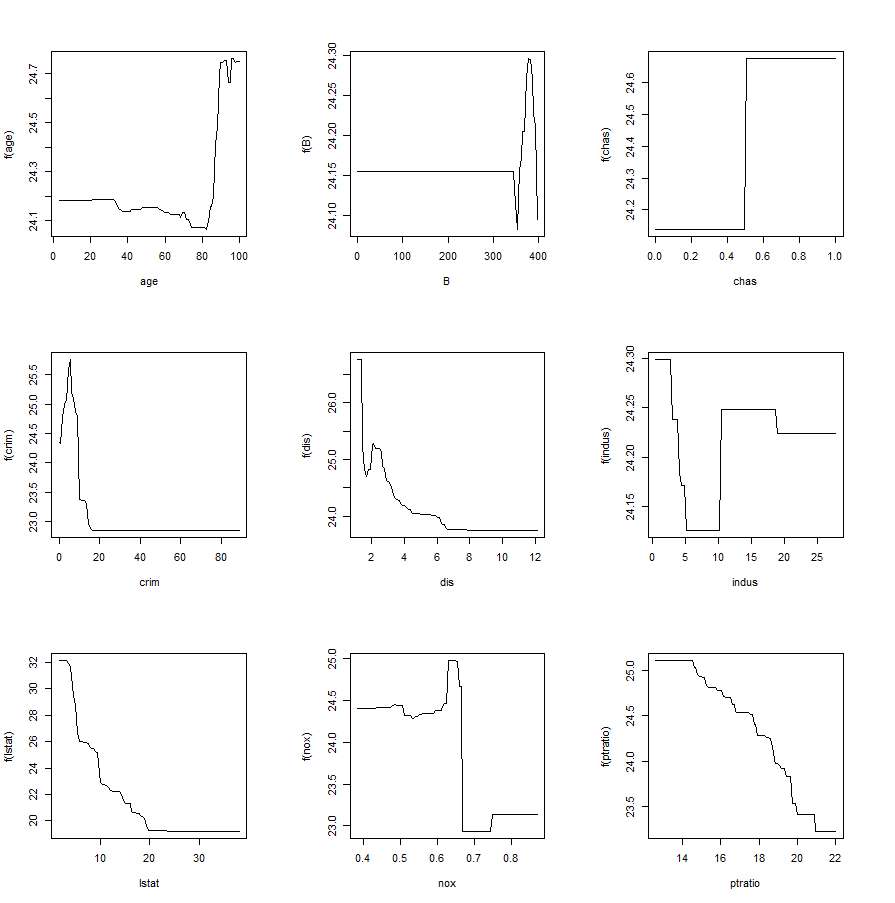
> plot.gbm(bos.gbm3,12,n.trees=9647)

> plot.gbm(bos.gbm3,13,n.trees=9647)

The marginal plots look quite noisy, however we need to look closely at the scale on the vertical axis. We might consider using a smaller number of boosting iterations (n.trees smaller) to see if the marginal effects are smoother.

Using n.trees = 400 suggested by test/validation set cross-validation

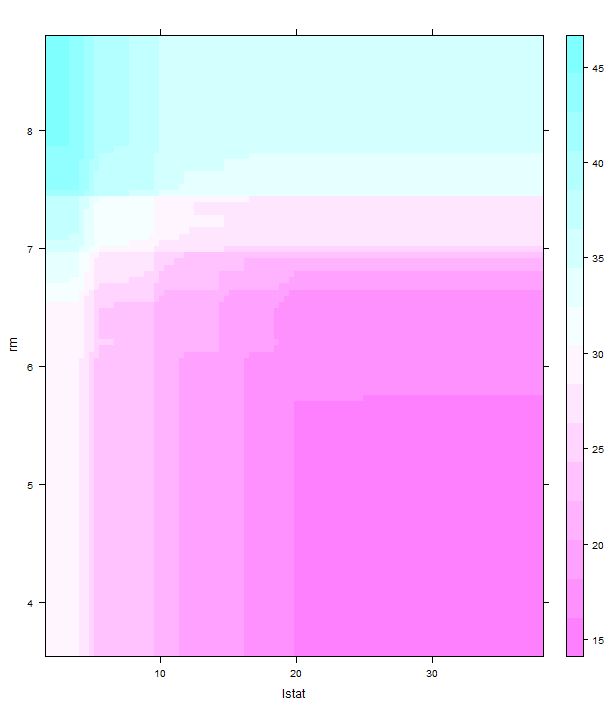
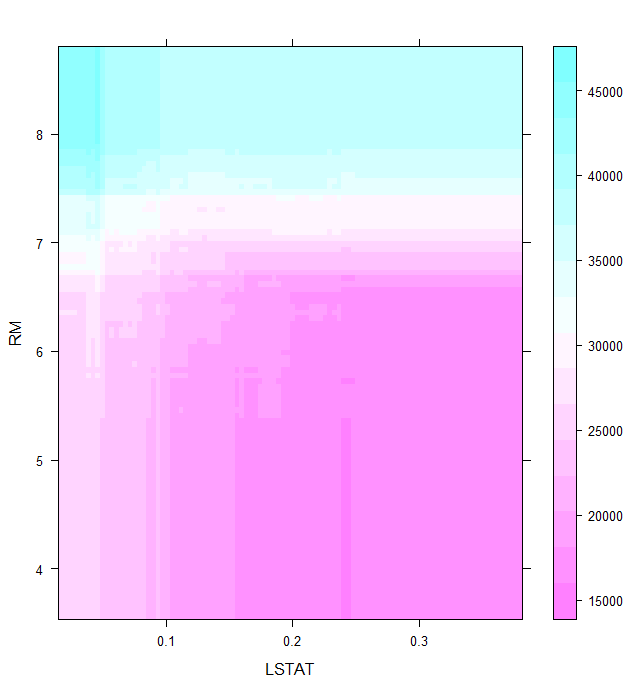


The marginal plots still look noisy, but they are a bit smoother. Further cross-validation will be necessary to gauge the predictive performance of these models.

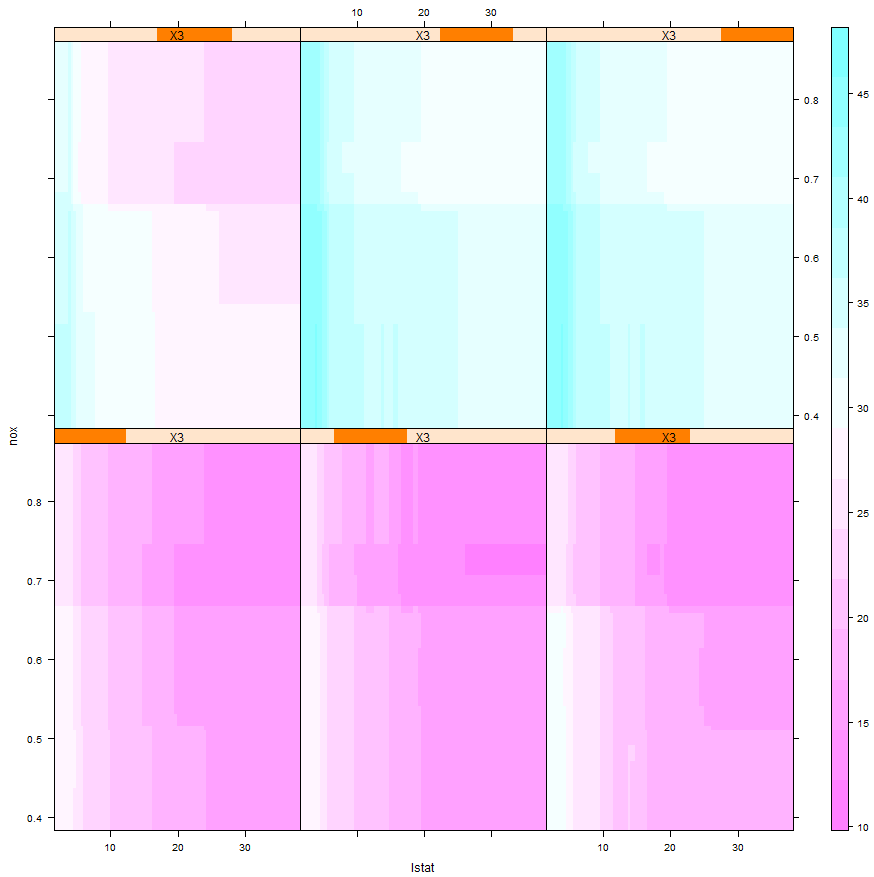
Contour plots (pick two variables by number to plot)

> plot.gbm(bos.gbm3,c(12,5),n.trees=9647) 🡨 left plot

> plot.gbm(bos.gbm3,c(12,5),n.trees=400) 🡨 right plot



> plot.gbm(bos.gbm2,c(7,8,11),n.trees=100)



WARNING: These take a VERY long time to run, even with an *M* this small!

These panels show the predicted median home value for a census tract as a function of the proportion of lower income families and the NOx concentration conditioning on the average number of rooms in the homes.

They are pretty poor approximations to what we can do in JMP to view the fitted response surface, however JMP does not have boosting for a numeric response.

**Example 10.8 - L.A. Basin Ozone Concentration**

> Ozdata = read.table(file.choose(),header=T,sep=”,”) 🡨 read **Ozone.csv**> Ozdata2 = data.frame(tupoz=Ozdata$upoz^.333,Ozdata[,-1])

> oz.gbm = gbm(tupoz~.,data=Ozdata2,distribution="gaussian",

n.trees=2000,shrinkage=.1,interaction.depth=4,bag.fraction=.5,  
train.fraction=.8,n.minobsinnode=10,cv.folds=5,keep.data=T,verbose=T)

> gbm.perf(oz.gbm,method="cv")

[1] 75

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

[1] 935

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

[1] 44

> yhat = predict(oz.gbm,n.trees=935)

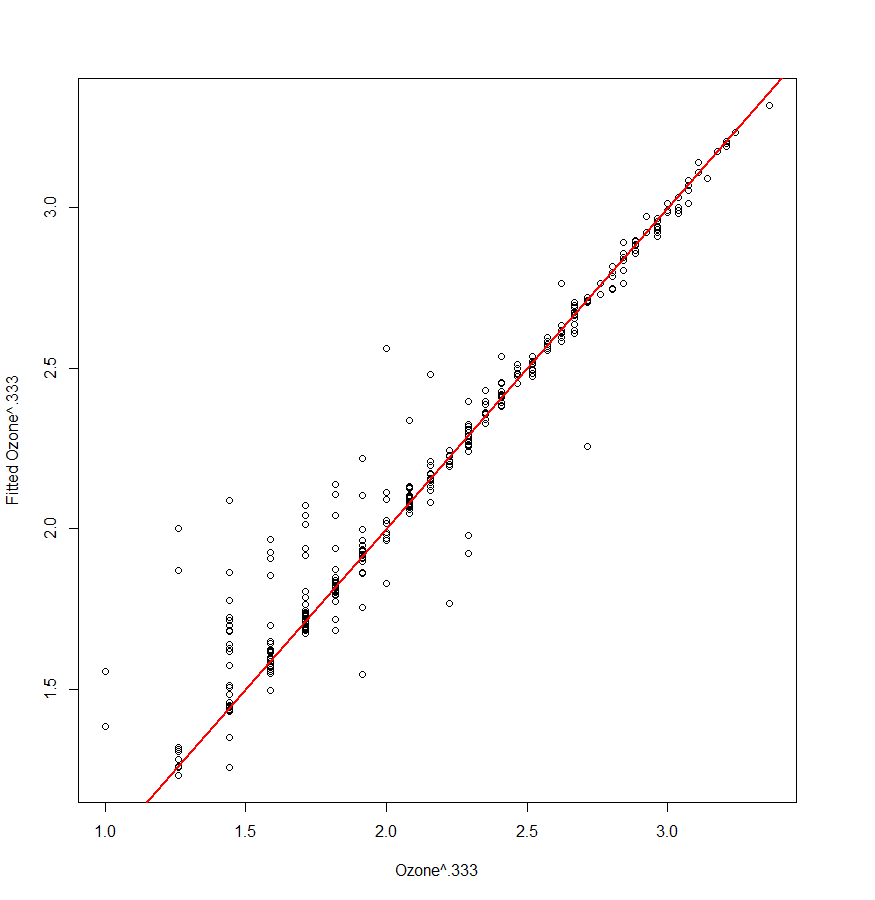
> plot(Ozdata2$tupoz,yhat,xlab="Ozone^.333",ylab="Fitted Ozone^.333")

> abline(0,1,col="red",lwd=2)> cor(Ozdata2$tupoz,yhat)

[1] 0.9700331

> cor(Ozdata2$tupoz,yhat)^2

[1] 0.9409642



> yhat = predict(oz.gbm,n.trees=75) 🡨 Optimal number of iterations from CV

> plot(Ozdata2$tupoz,yhat,xlab="Ozone^.333",ylab="Fitted Ozone^.333")

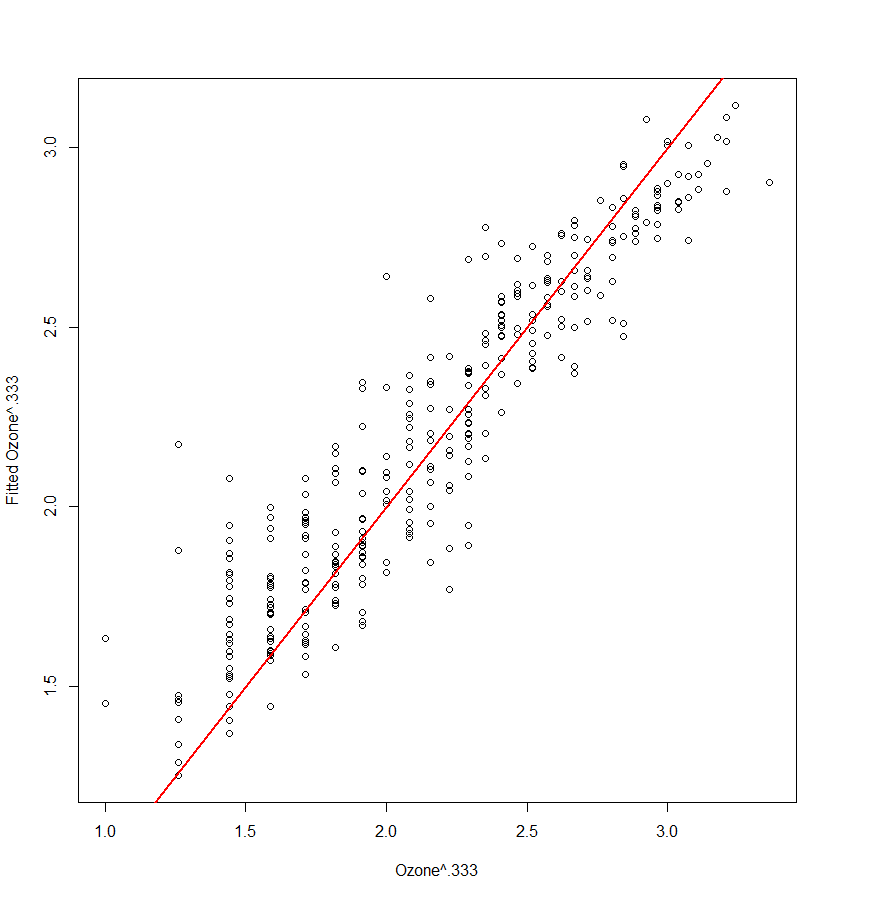
> abline(0,1,col="red",lwd=2)

> cor(Ozdata2$tupoz,yhat)

[1] 0.9304838

> cor(Ozdata2$tupoz,yhat)^2

[1] 0.8658001



**Jm = 6**

> oz.gbm3 = gbm(tupoz~.,data=Ozdata2,distribution="gaussian",

n.trees=2000,shrinkage=.1,interaction.depth=5,bag.fraction=.5,  
train.fraction=.8,n.minobsinnode=10,cv.folds=5,keep.data=T,verbose=T)

> gbm.perf(oz.gbm3,method="test")

[1] 1152

> yhat = predict(oz.gbm,n.trees=1152)

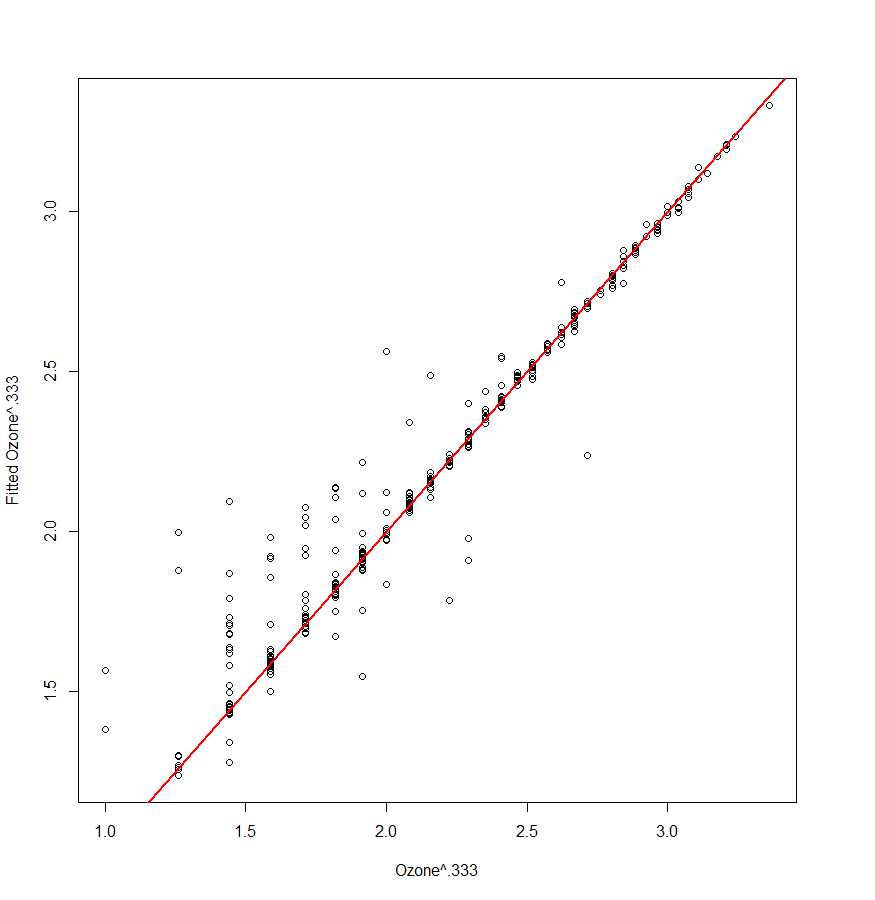
> plot(Ozdata2$tupoz,yhat,xlab="Ozone^.333",ylab="Fitted Ozone^.333")

> cor(Ozdata2$tupoz,yhat)

[1] 0.9700326

> cor(Ozdata2$tupoz,yhat)^2

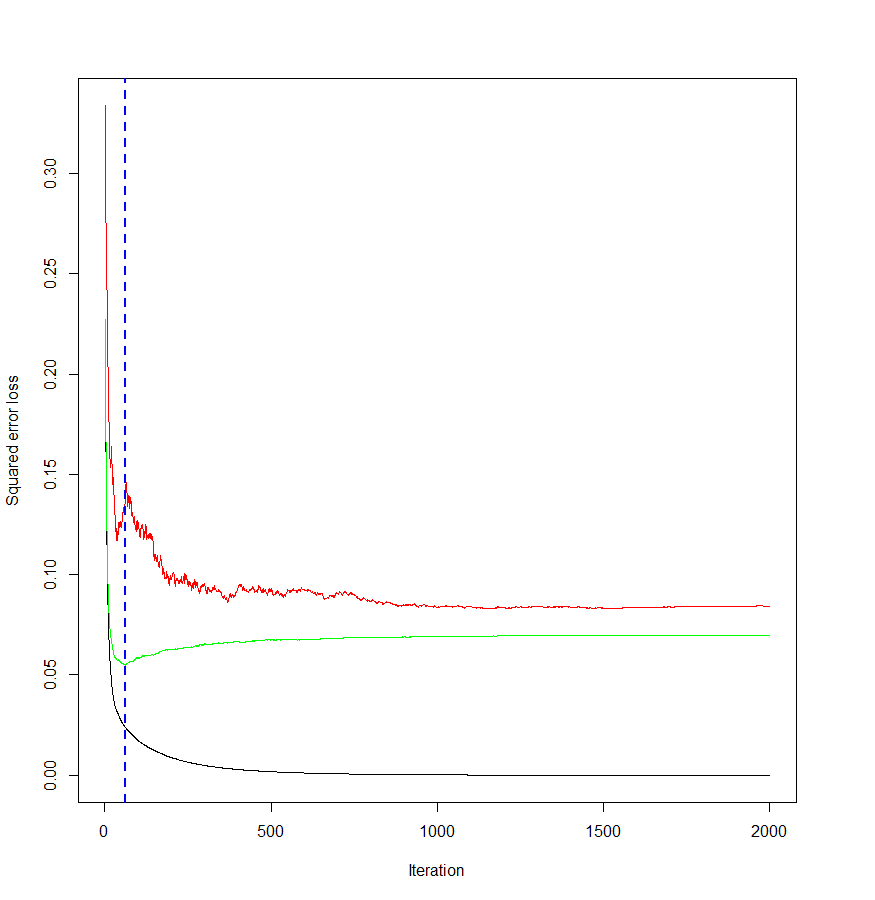
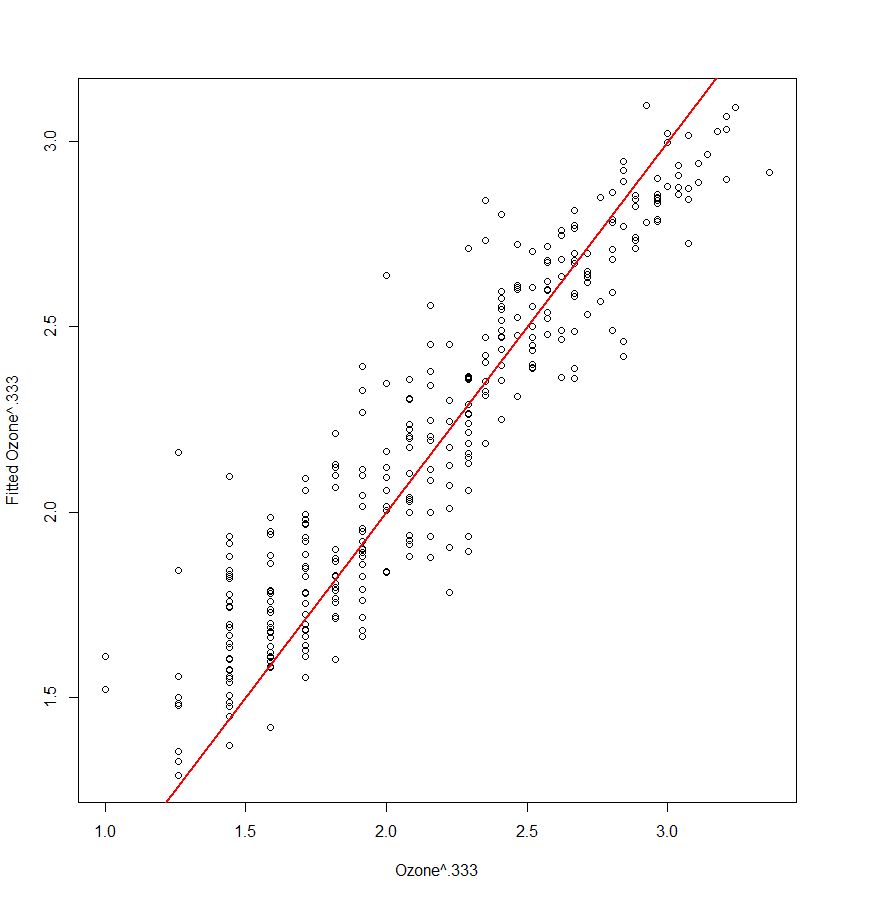
[1] 0.9409633



**OVERFIT?**

> gbm.perf(oz.gbm3,method="cv") 🡨 Using number of iterations from 5-fold CV

[1] 61

> yhat = predict(oz.gbm,n.trees=61)

> plot(Ozdata2$tupoz,yhat,xlab="Ozone^.333",ylab="Fitted Ozone^.333")

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

> cor(Ozdata2$tupoz,yhat)

[1] 0.9269789

> cor(Ozdata2$tupoz,yhat)^2

[1] 0.8592899

> summary(oz.gbm3,n.trees=61)

var rel.inf

safb safb 41.903949

inbt inbt 22.346999

day day 9.535922

inbh inbh 9.147939

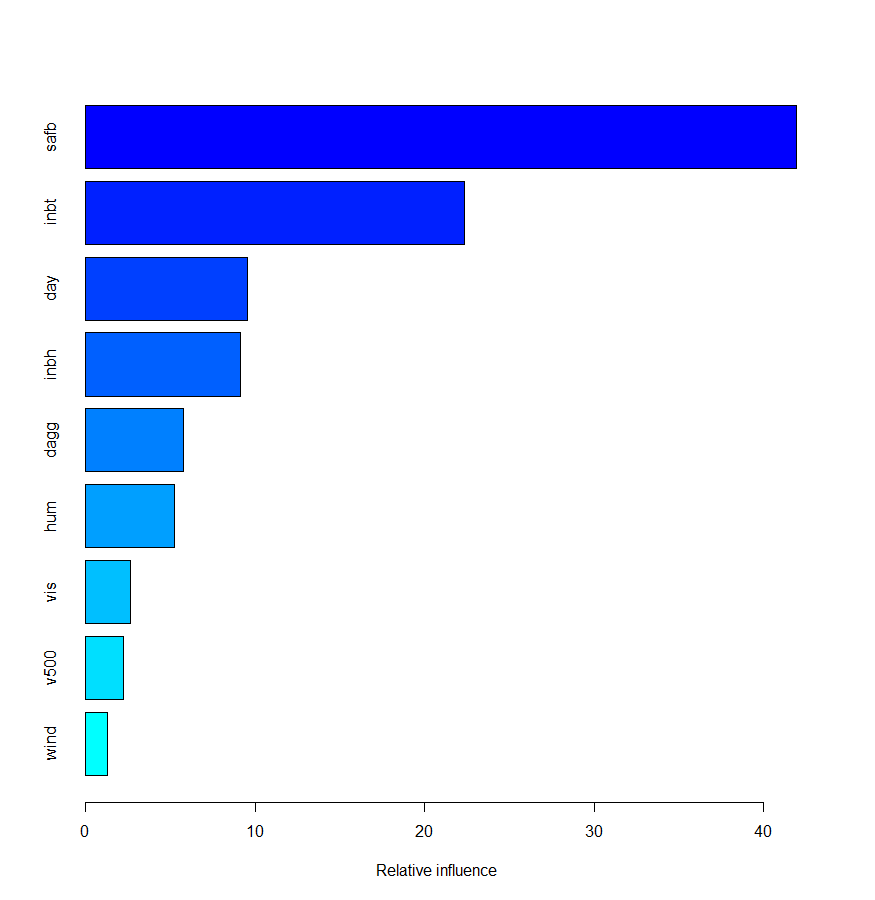
dagg dagg 5.735089

hum hum 5.207820

vis vis 2.615909

v500 v500 2.209559

wind wind 1.296814



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

> plot.gbm(oz.gbm3,1,n.trees=61)

> plot.gbm(oz.gbm3,2,n.trees=61)

> plot.gbm(oz.gbm3,3,n.trees=61)

> plot.gbm(oz.gbm3,4,n.trees=61)

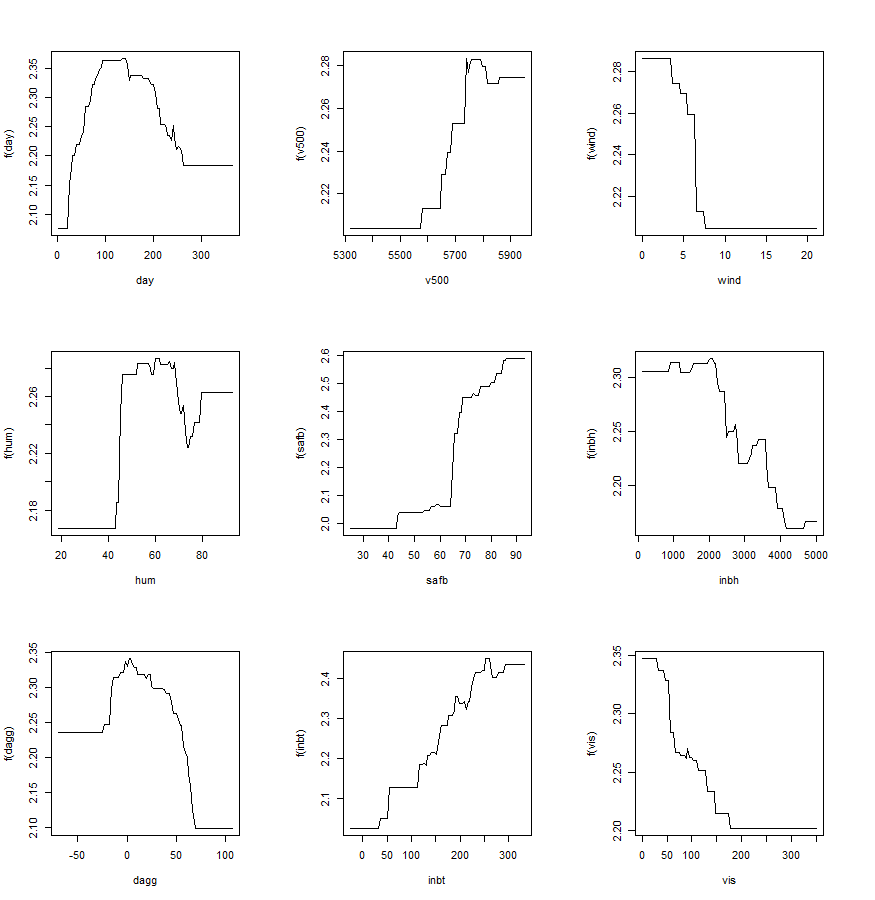
> plot.gbm(oz.gbm3,5,n.trees=61)

> plot.gbm(oz.gbm3,6,n.trees=61)

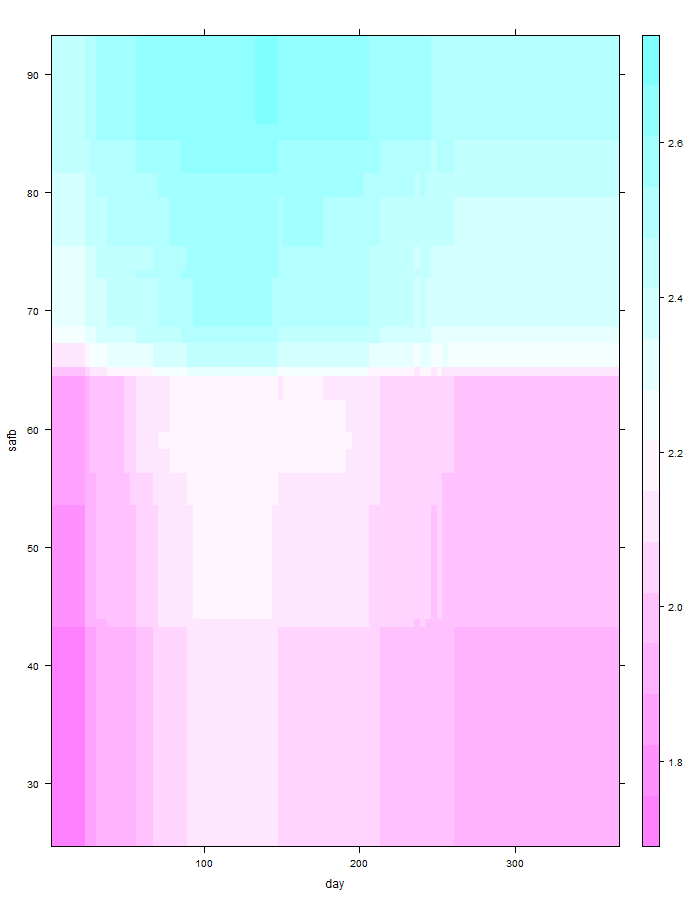
> plot.gbm(oz.gbm3,7,n.trees=61)

> plot.gbm(oz.gbm3,8,n.trees=61)

> plot.gbm(oz.gbm3,9,n.trees=61)

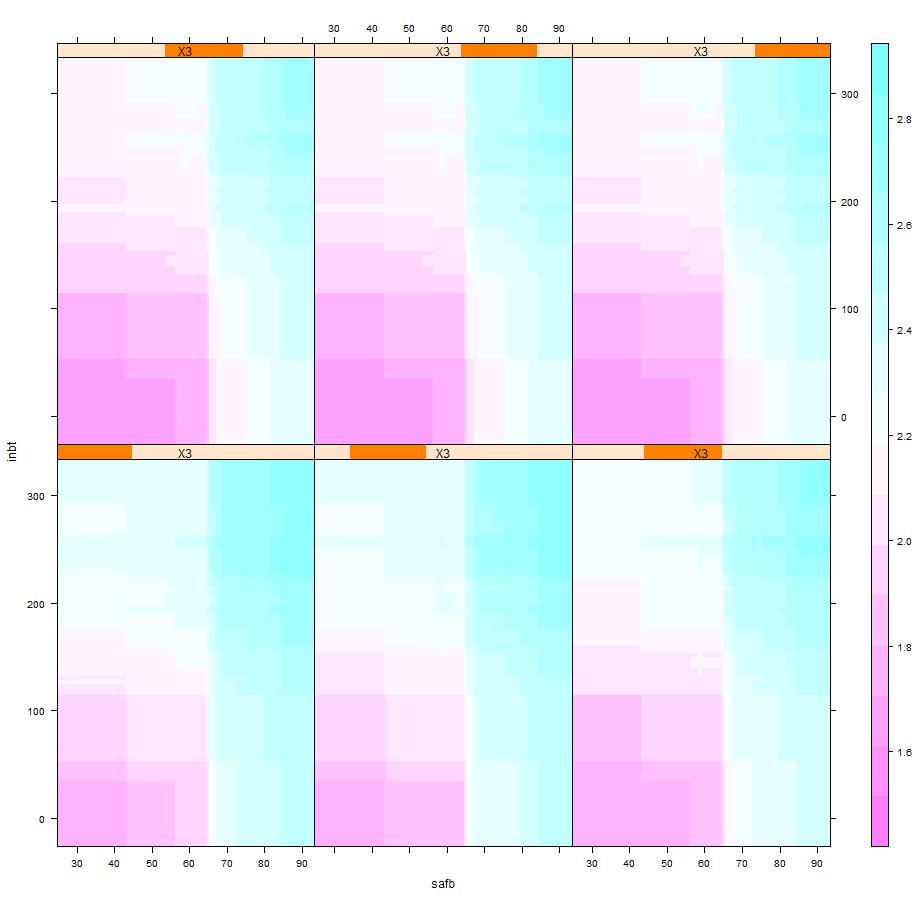


> plot.gbm(oz.gbm3,c(1,5),n.trees=61)



> plot.gbm(oz.gbm3,c(5,8,1),n.trees=45)

Day is in the conditioning plot plot. Warning this takes a very long time to run on even a small dataset!



Based upon my experimentation with gradient boosted regression trees it seems that shrinkage of is a good starting point, even though the literature suggests using a small shrinkage ( the default). Experimenting with larger trees (Jm) and different shrinkage values (larger or smaller) seems to be worthwhile. For example with the L.A. ozone data a shrinkage of with produced a “better” tree than either of the ones above.

A MC split-sample cross-validation function for gradient boosted regression trees should take information about the terminal node size (interaction.depth+1), which method will be used to choose *M* (i.e.. OOB, CV, or Train/Test), and what shrinkage () value to use throughout as arguments. The value for M could therefore vary from one bootstrap sample to the next. We definitely do not need to see verbose output from each loop. We also want to consider relatively small values for B because computation time could be an issue.

> oz.gbm = gbm(tupoz~.,data=Ozdata2,distribution="gaussian",n.trees=2000,  
shrinkage=.1,interaction.depth=4,bag.fraction=0.5,train.fraction=.8,  
n.minobsinnode=10,cv.folds=5,keep.data=T,verbose=F)

Using 5-fold CV value for > set.seed(1)  
> results = gbm.cv(oz.gbm,Ozdata2,B=10,method="cv")

> results = gbm.cv(oz.gbm,data=Ozdata2)

RMSEP = 0.2490317 MAEP= 0.1949044 MAPEP= 0.09710203  
> summary(results)

RMSEP MAEP MAPEP

Min. :0.2147 Min. :0.1679 Min. :0.07944

1st Qu.:0.2328 1st Qu.:0.1830 1st Qu.:0.09003

Median :0.2429 Median :0.1910 Median :0.09673

Mean :0.2480 Mean :0.1949 Mean :0.09710

3rd Qu.:0.2691 3rd Qu.:0.2078 3rd Qu.:0.10420

Max. :0.2820 Max. :0.2278 Max. :0.11966

Using train/test value for > set.seed(1)

> results = gbm.cv(oz.gbm,data=Ozdata2,method="test")

RMSEP = 0.2484639 MAEP= 0.1944672 MAPEP= 0.09703495

> summary(results)

RMSEP MAEP MAPEP

Min. :0.2164 Min. :0.1712 Min. :0.08081

1st Qu.:0.2318 1st Qu.:0.1816 1st Qu.:0.08979

Median :0.2462 Median :0.1941 Median :0.09808

Mean :0.2475 Mean :0.1945 Mean :0.09703

3rd Qu.:0.2688 3rd Qu.:0.2048 3rd Qu.:0.10355

Max. :0.2768 Max. :0.2201 Max. :0.11473

Using Out-of-Bag (OOB) value for   
> set.seed(1)

> results = gbm.cv(oz.gbm,data=Ozdata2,method="OOB")  
RMSEP = 0.2514705 MAEP= 0.1976147 MAPEP= 0.099785  
> summary(results)

RMSEP MAEP MAPEP

Min. :0.2118 Min. :0.1726 Min. :0.08410

1st Qu.:0.2370 1st Qu.:0.1859 1st Qu.:0.09135

Median :0.2462 Median :0.1968 Median :0.10116

Mean :0.2503 Mean :0.1976 Mean :0.09978

3rd Qu.:0.2733 3rd Qu.:0.2058 3rd Qu.:0.10620

Max. :0.2844 Max. :0.2316 Max. :0.12210

gbm.cv = function(fit,data,p=.667,B=10,method="cv",interaction.depth=4,

shrinkage=.1,cv.folds=5) {

n = dim(data)[1]

y = data[,1]

MSE <- rep(0,B)

MAE = rep(0,B)

MAPE = rep(0,B)

for (i in 1:B) {

ss = floor(n\*p)

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

fit2 = gbm(formula(data[sam,]),data=data[sam,],

shrinkage=shrinkage,distribution="gaussian",

n.trees=fit$n.trees,bag.fraction=fit$bag.fraction,

train.fraction=fit$train.fraction,

interaction.depth=interaction.depth,n.minobsinnode=10,

cv.folds=cv.folds,verbose=F)

m = gbm.perf(fit2,method=method,plot.it=F)

ynew = predict(fit2,n.tree=m,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])

}

cat("RMSEP =",sqrt(mean(MSE))," MAEP=",mean(MAE)," MAPEP=",mean(MAPE))

cv = return(data.frame(RMSEP=sqrt(MSE),MAEP=MAE,MAPEP=MAPE))

}

This MCCV function is useful for choosing values for the interaction depth ( and the shrinkage (). Also the function gbm.cv above assesses the boosted tree models based upon the method employed to choose the number of trees to be “averaged” to obtain the final fit, with method “cv” recommended.

Another approach would be to anchor down values for the shrinkage parameter , the number of terminal nodes in the trees used in the boosted fit , and the number of trees (*M*) to average chosen by using the cross-validation features within in the gbm function. We could then write a cross-validation function that takes these tuning parameters as arguments and use bootstrapped training/test sets to assess predictive performance.

A better approach is to set aside a validation set that is NOT used in the model development process at all and then fit models using the wide variety of approaches we have examined. We can then predict the response value for the validation set to get an idea of which method has the best predictive performance. We will demonstrate this using the Boston Housing data with response in the log-scale then back-transformed to compare prediction accuracy in the original scale.

**Formation of Training and Validation Set**

> names(BostonGBM)

[1] "medv" "age" "B" "chas" "crim" "dis" "indus" "lstat"

[9] "nox" "ptratio" "rad" "rm" "tax" "zn"

> Statplot(BostonGBM$medv)  
  
> dim(Boston.yt)

[1] 506 14  
  
> set.seed(1)

> sam = sample(1:506,400,replace=F)

> Boston.train = BostonGBM[sam,]

> Boston.test = BostonGBM[-sam,]

> dim(Boston.train)

[1] 400 14

> dim(Boston.test)

[1] 106 14

> yact = exp(Boston.test$medv)

> length(yact)

[1] 106

**OLS** (log transformed response, step-wise selection, no predictor transformations)  
  
> bos.lm = lm(medv~.,data=Boston.train)  
> bos.step = step(bos.lm)

> ypred = predict(bos.step,newdata=Boston.test)

> ypred = predict(bos.step,newdata=Boston.test)

> ypred = exp(ypred)

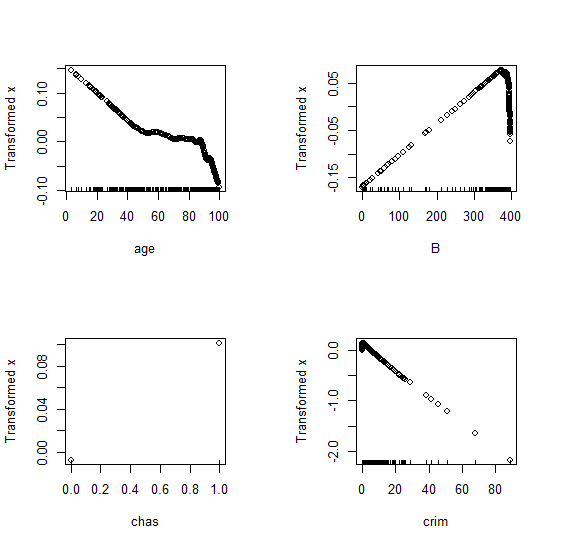
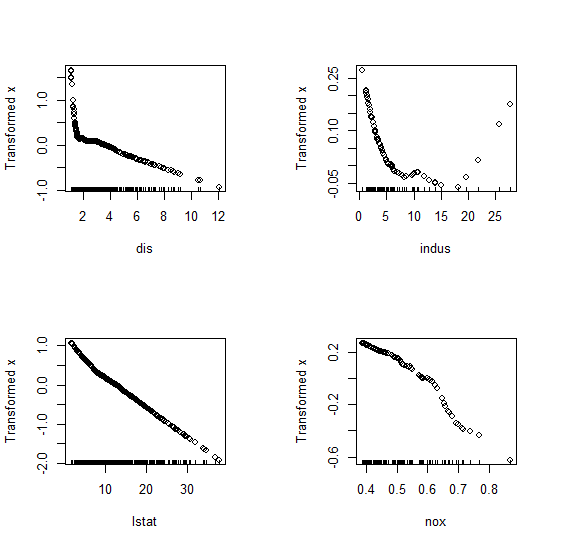
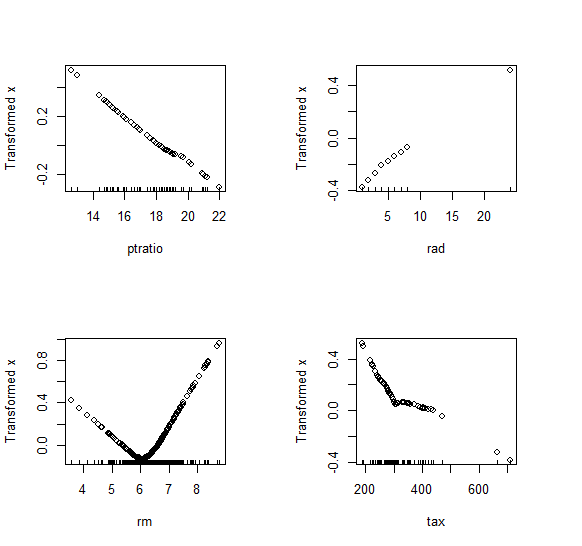
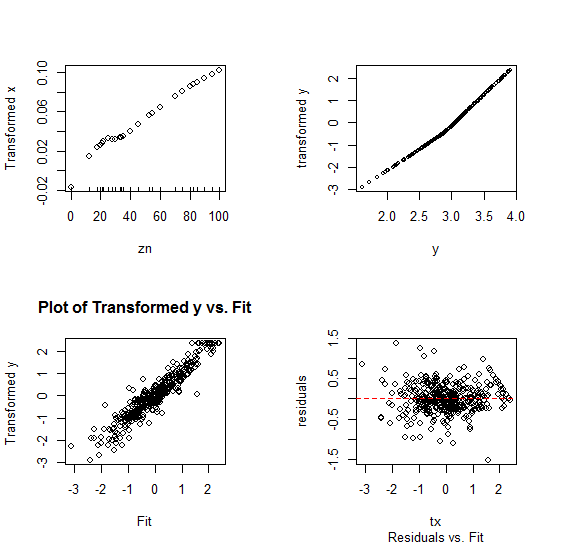
> RMSEP.ols = sqrt(mean((ypred-yact)^2))

**> RMSEP.ols**

**[1] 4.399603**  
**ACE Inspired OLS**> library(acepack)

> X = Boston.train[,-1]

> y = Boston.train[,1]> bos.ace = ace(X,y)  
> maceplot(X,y,bos.ace)

> bos.olsace = lm(medv~poly(age,3)+I((B-350)^2)+poly(dis,2)  
+poly(indus,2)+I(log(lstat))+poly(nox,2)+poly(rm,2)+crim+tax+rad+  
ptratio+chas+zn,data=Boston.train)

> ypred.olsace = predict(bos.olsace,newdata=Boston.test)

> ypred.olsace = exp(ypred.olsace)

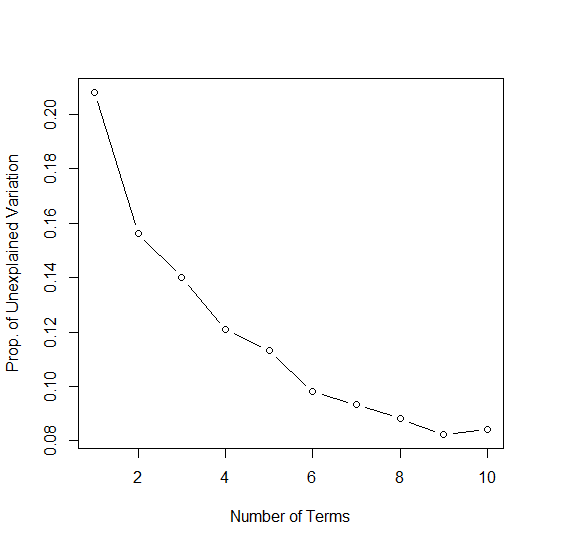
> RMSEP.olsace = sqrt(mean((yact-ypred.olsace)^2))

> RMSEP.olsace

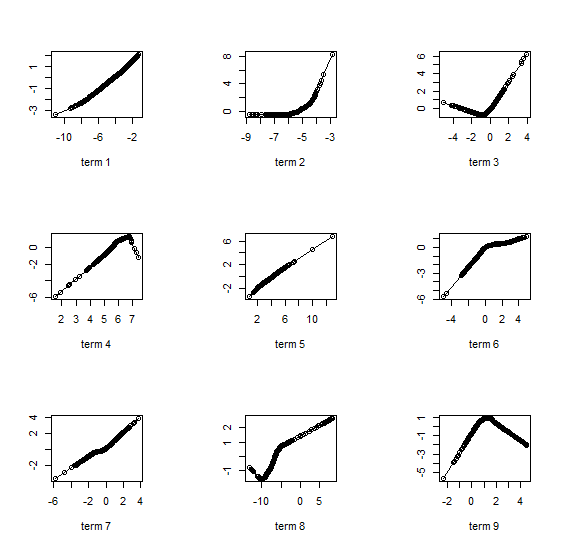
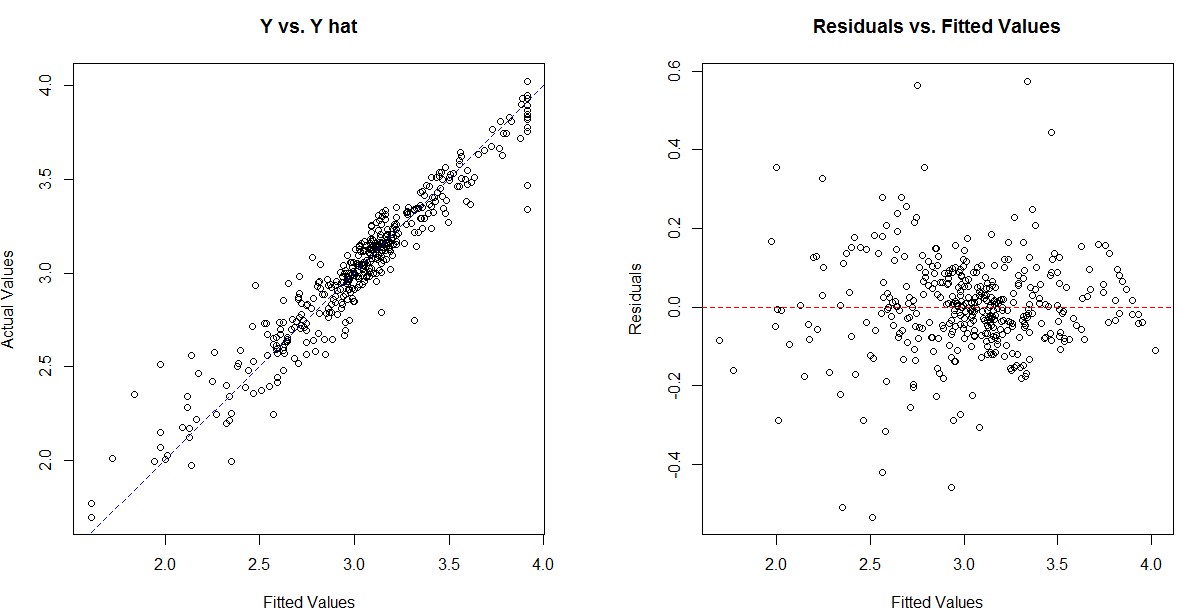
[1] 3.475354

**PPR**  
> bos.ppr = ppr(medv~.,data=Boston.train,nterms=1,max.terms=10,bass=1)

> PPplot(bos.ppr)



Optimal somewhere between 6-10 terms?

Using Monte Carlo Split-Sample cross-validation 3 terms is determined to be optimal.

> bos.ppr = ppr(medv~.,data=Boston.train,nterms=3,max.terms=3,bass=1)

> ypred.ppr = exp(predict(bos.ppr,newdata=Boston.test))

> RMSEP.ppr = sqrt(mean((ypred.ppr-yact)^2))

> RMSEP.ppr

[1] 3.28523

**Neural Networks (h = 1, decay = .001)**  (1 hidden node chosen using nnet.sscv function)

> bos.nn = nnet(logmedv~.,data=Boston.train,size=1,decay=.001,skip=T,linout=T)

> ypred.nn = exp(predict(bos.nn,newdata=Boston.test))

> RMSEP.nn = sqrt(mean((ypred.nn-yact)^2))

> RMSEP.nn

[1] 3.333039

**RPART** (cp chosen using internal cross-validation and 1-SE rule)

> bos.rpart = rpart(medv~.,data=Boston.train,cp=.006,minbucket=5,minsplit=8)

> ypred.rpart = exp(predict(bos.rpart,newdata=Boston.test))

> RMSEP.rpart = sqrt(mean((ypred.rpart-yact)^2))

> RMSEP.rpart

[1] 3.715835

**Bagging** (Num. of average trees (nbagg=1000) chosen via cross-validation using the training data only)  
> bos.bag1000 = bagging(medv~.,data=Boston.train,nbagg=1000,cp=.006,  
minbucket=5,minsplit=8)

> ypred.bag = exp(predict(bos.bag1000,newdata=Boston.test))

> RMSEP.bag = sqrt(mean((ypred.bag-yact)^2))

> RMSEP.bag

[1] 3.345338

**Random Forests** (ntree =100, m = 5) 🡨 chosen using the rf.sscv function above.

> bos.rf = randomForest(medv~.,data=Boston.train,mtry=5,ntree=100)

> ypred.rf = exp(predict(bos.rf,newdata=Boston.test))

> RMSEP.rf = sqrt(mean((ypred.rf-yact)^2))

> RMSEP.rf

[1] 2.973987

**Gradient Boosted Trees**

> bos.gbm = gbm(medv~.,data=Boston.train,distribution="gaussian",n.trees=20000,

interaction.depth=5,shrinkage=0.05,bag.fraction=.5,train.fraction=.8,  
cv.folds=5)

> gbm.perf(bos.gbm,method="cv")

[1] 303

> ypred.boost = exp(predict(bos.gbm,newdata=Boston.test,n.trees=303))

> RMSEP.boost = sqrt(mean((ypred.boost-yact)^2))

> RMSEP.boost

[1] 2.527348

**And the winner is….. Boosted Trees!**

**Discussion:**

As I have mentioned previously, for *kaggle.com* prediction problems the standard benchmark to beat is a random forest, typically run in R or Python. Thus it is not surprising that it was nearly the winner here. It is important to note that I set aside a portion of the data to determine which model had the best predictive performance and these set aside data were not used in the model building process at all! Although not completely shown above, I used cross-validation (####.sscv functions) extensively when working with the training data (Boston.train) in determining the values of the tuning parameters for the different methods I employed. You will notice I did not consider ridge regression, Lasso or Elastic Net regression, PCR, or PLS for these data as the number of predictors for the Boston Housing data is not large (*p = 13*) and there are not any strong correlations amongst the potential predictors.

**10.7 - Boosting with XGBoost**

XGBoost stands for eXtreme Gradient Boosting.

The name xgboost, though, actually refers to the engineering goal to push the limit of computational resources for boosted tree algorithms. Which is the reason why many people use xgboost.

— Tianqi Chen, in answer to the question “What is the difference between the R gbm (gradient boosting machine) and xgboost (extreme gradient boosting)?” on Quora

It is an implementation of gradient boosting machines created by Tianqi Chen, now with contributions from many developers. It belongs to a broader collection of tools under the umbrella of the Distributed Machine Learning Community or DMLC who are also the creators of the popular mxnet deep learning library.

Tianqi Chen provides a brief and interesting back story on the creation of XGBoost in the post Story and Lessons Behind the Evolution of XGBoost found at the link below:  
( <http://homes.cs.washington.edu/~tqchen/2016/03/10/story-and-lessons-behind-the-evolution-of-xgboost.html> )

XGBoost is a software library that you can download and install on your machine, then access from a variety of interfaces. Specifically, XGBoost supports the following main interfaces:

* Command Line Interface (CLI).
* C++ (the language in which the library is written).
* Python interface as well as a model in scikit-learn.
* R interface as well as a model in the caret package. (see Applied Predictive Modeling)
* Julia.
* Java and JVM languages like Scala and platforms like Hadoop.

**Features of XGBoost**

The library is laser focused on computational speed and model performance, as such there are few frills. Nevertheless, it does offer a number of advanced features.

Model Features

The implementation of the model supports the features of the scikit-learn and R implementations, with new additions like regularization. Three main forms of gradient boosting are supported:

* Gradient Boosting algorithm also called gradient boosting machine including the learning rate.
* Stochastic Gradient Boosting with sub-sampling at the row, column and column per split levels.
* Regularized Gradient Boosting with both L1 and L2 regularization.

You can read more about it and watch some excellent videos on the XGBoost on this website:

<https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/>

The library xgboost is the R implementation of XGBoost.

Below are some examples of XGBoost in R.

**Example 1 – QSAR Melting Points**

QSAR.mtp = read.csv(file=”http://course1.winona.edu/bdeppa/Stat%20425/Data/QSARmtp.csv”)

dim(QSAR.mtp)

summary(QSAR.mtp)

sam = sample(1:4401,floor(4401\*.6667),replace=FALSE)

The data format is a bit strange for XGBoost. The predictor matrix must be stored as a matrix, not a data frame, and the response must be a list. Below we form a training and test set with the proper format for both the predictors and the response.

Q.train = QSAR.mtp[sam,]

Q.test = QSAR.mtp[-sam,]

Q.train.x = as.matrix(Q.train[,-1])

Q.test.x = as.matrix(Q.test[,-1])

Q.train.y = as.list(Q.train[,1])

Q.test.y = as.list(Q.test[,1])

There a numerous tuning parameters in XGBoost that can tweaked to improve predictive performance. This can be a nightmare as there are many things that can be tuned. Below is a list of the main tuning parameters.

**Parameters for an XGBoost Tree Booster**

(booster = “gbtree”, “gblinear”,”dart” which is the default)

1. **nrounds[default=100]**
   * It controls the maximum number of iterations. For classification, it is similar to the number of trees to grow.
   * Should be tuned using CV
2. **eta [default=0.3][range: (0,1)]**
   * It controls the learning rate, i.e., the rate at which our model learns patterns in the data. After every round, it shrinks the feature weights to reach the best optimum.
   * Lower eta leads to slower computation. It must be supported by increase in nrounds.
   * Typically, it lies between 0.01 - 0.3
3. **gamma [default=0][range: (0,Inf)]**
   * It controls regularization (or prevents overfitting). The optimal value of gamma depends on the data set and other parameter values.
   * Higher the value, higher the regularization. Regularization means penalizing large coefficients which don't improve the model's performance. default = 0 means no regularization.
   * *Tune trick:* Start with 0 and check CV error rate. If you see train error >>> test error, bring gamma into action. Higher the gamma, lower the difference in train and test CV. If you have no clue what value to use, use gamma=5 and see the performance. Remember that gamma brings improvement when you want to use shallow (low max\_depth) trees.
4. **max\_depth [default=6][range: (0,Inf)]**
   * It controls the depth of the tree (i.e. # of splits and hence # of terminal nodes)
   * Larger the depth, more complex the model; higher chances of overfitting. There is no standard value for max\_depth. Larger data sets require deep trees to learn the rules from data.
   * Should be tuned using CV
5. **min\_child\_weight [default=1][range:(0,Inf)]**
   * In regression, it refers to the minimum number of instances required in a child node. In classification, if the leaf node has a minimum sum of instance weight (calculated by second order partial derivative) lower than min\_child\_weight, the tree splitting stops.
   * In simple words, it blocks the potential feature interactions to prevent overfitting. Should be tuned using CV.
6. **Subsample [default=1][range: (0,1)]**
   * It controls the number of samples (observations) supplied to a tree.
   * Typically, its values lie between (0.5-0.8)
7. **colsample\_bytree [default=1][range: (0,1)]**
   * It control the number of features (variables) supplied to a tree
   * Typically, its values lie between (0.5,0.9)
8. **lambda [default=0]**
   * It controls L2 regularization (equivalent to Ridge regression) on weights. It is used to avoid overfitting.
9. **alpha [default=1]**
   * It controls L1 regularization (equivalent to Lasso regression) on weights. In addition to shrinkage, enabling alpha also results in feature selection. Hence, it's more useful on high dimensional data sets.

**Parameters for a Linear Booster** (booster = “gblinear”, which is NOT the default)

Using linear booster has relatively lesser parameters to tune, hence it computes much faster than gbtree booster. There is also a new booster called dart which has its own set of tuning parameters.

1. **nrounds[default=100]**
   * It controls the maximum number of iterations (steps) required for gradient descent to converge.
   * Should be tuned using CV
2. **lambda[default=0]**
   * It enables Ridge Regression. Same as above
3. **alpha[default=1]**
   * It enables Lasso Regression. Same as above

Below is a basic xgboost call with a few of the tuning parameters set to non-default settings. To improve predictive performance we can try adjusting things like max.depth, eta, and nrounds.

xgboost(data=Q.train.x,label=Q.train.y,max.depth=2,eta=.5,nthread=4,nround=20,verbose=T,objective="reg:squarederror")  
> bst = xgboost(data=Q.train.x,label=Q.train.y,max.depth=2,eta=.5,

nthread=4,nround=20,verbose=T,objective="reg:linear")

[1] train-rmse:101.073021

[2] train-rmse:69.030128

[3] train-rmse:57.174156

[4] train-rmse:53.024014

[5] train-rmse:51.406677

[6] train-rmse:50.501595

[7] train-rmse:49.775265

[8] train-rmse:49.337597

[9] train-rmse:48.830616

[10] train-rmse:48.459248

[11] train-rmse:48.128696

[12] train-rmse:47.809811

[13] train-rmse:47.490067

[14] train-rmse:47.216835

[15] train-rmse:46.855366

[16] train-rmse:46.468288

[17] train-rmse:46.285892

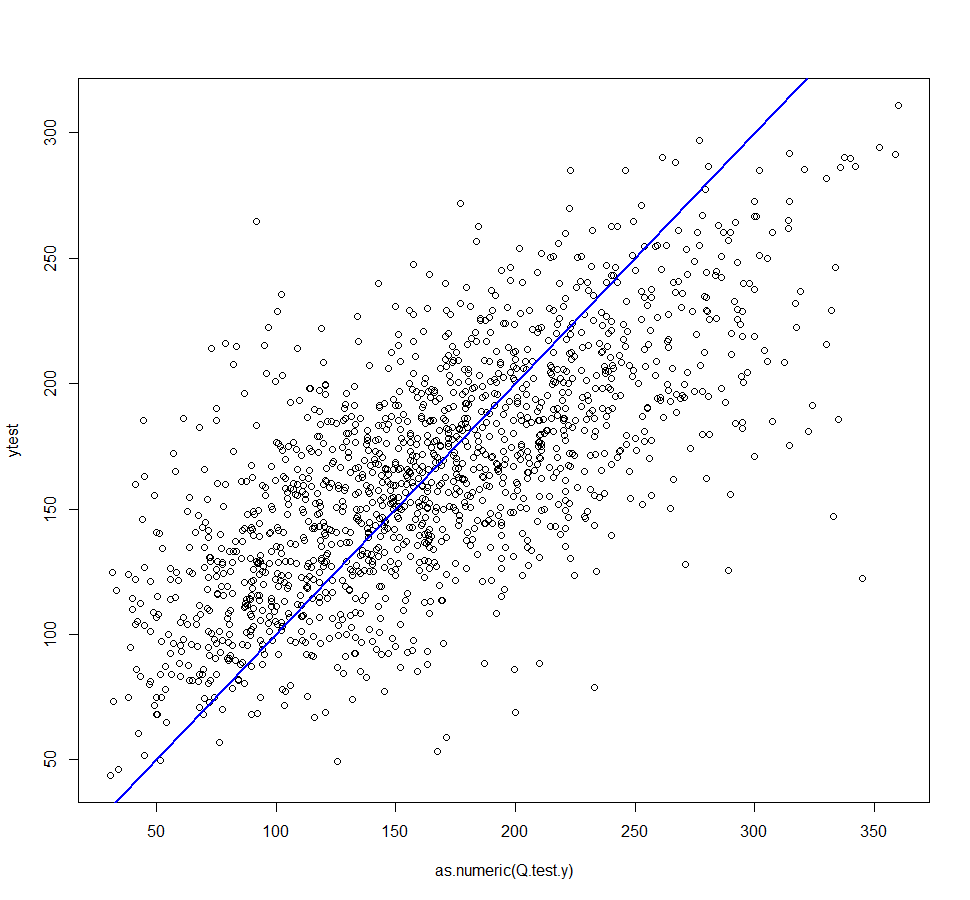
[18] train-rmse:45.989361

[19] train-rmse:45.611977

[20] train-rmse:45.363712

ytest = predict(bst,Q.test.x)

plot(as.numeric(Q.test.y),ytest)

abline(0,1,lwd=2,col="blue")  


RMSEP.test = sqrt(mean((as.numeric(Q.test.y)-ytest)^2))

RMSEP.test  
[1] 46.82609

For practice at this point, you should try adjusting max.depth, eta, and nrounds to see if you can find an “optimal” combination of just these parameters. If you are really feeling ambitious, you could try bringing even more tuning parameters into the mix, but instead we will consider more efficient CV processes.

Alternative approach to a simple train and validation set

First we form objects containing both the predictor matrix and response vector as a list for the training & test sets. To do this we need to form an object containing them using a unique structure for XGBoost. The function xgb.DMatrix is used to do this. Both a training and validation/test set will be supplied to the algorithm through the use of watchlist. We will get model performance measures for both sets as the fitting is done.

dtrain = xgb.DMatrix(data=Q.train.x,label=Q.train.y)

dtest = xgb.DMatrix(data=Q.test.x,label=Q.test.y)

watchlist = list(train=dtrain,test=dtest)

bst2 = xgb.train(data=dtrain,max.depth=5,eta=1,nthread=4,nround=10,

watchlist=watchlist,objective="reg:linear")  
[1] train-rmse:49.224373 test-rmse:55.438202

[2] train-rmse:45.937565 test-rmse:54.394554

[3] train-rmse:44.184669 test-rmse:53.822117

[4] train-rmse:41.813446 test-rmse:54.200741

[5] train-rmse:40.792721 test-rmse:54.267799

[6] train-rmse:38.855255 test-rmse:54.529831

[7] train-rmse:37.078194 test-rmse:54.894909

[8] train-rmse:35.990852 test-rmse:55.287357

[9] train-rmse:35.043217 test-rmse:55.447208

[10] train-rmse:33.291763 test-rmse:55.654099

bst2 = xgb.train(data=dtrain,max.depth=10,eta=.5,nthread=4,nround=10,

watchlist=watchlist,objective="reg:linear")  
[1] train-rmse:97.294868 test-rmse:102.185989

[2] train-rmse:57.760612 test-rmse:71.553329

[3] train-rmse:37.256580 test-rmse:59.576885

[4] train-rmse:26.775166 test-rmse:55.224998

[5] train-rmse:20.255905 test-rmse:53.849976

[6] train-rmse:17.791151 test-rmse:52.989429

[7] train-rmse:14.731245 test-rmse:53.147751

[8] train-rmse:12.850547 test-rmse:53.260628

[9] train-rmse:12.211336 test-rmse:53.262005

[10] train-rmse:11.141236 test-rmse:53.243507

bst2 = xgb.train(data=dtrain,max.depth=5,eta=.5,nthread=4,nround=10,

watchlist=watchlist,objective="reg:linear")  
[1] train-rmse:98.783081 test-rmse:100.669258

[2] train-rmse:63.842075 test-rmse:68.364723

[3] train-rmse:49.309315 test-rmse:56.607391

[4] train-rmse:43.726936 test-rmse:52.487278

[5] train-rmse:41.200855 test-rmse:51.288109

[6] train-rmse:39.460869 test-rmse:50.716461

[7] train-rmse:38.513119 test-rmse:50.169209

[8] train-rmse:37.919258 test-rmse:50.234356

[9] train-rmse:37.033707 test-rmse:50.197178

[10] train-rmse:36.166546 test-rmse:50.101147

bst2 = xgb.train(data=dtrain,max.depth=4,eta=.25,nthread=4,nround=100,

watchlist=watchlist,objective="reg:linear")

[1] train-rmse:137.087204 test-rmse:137.495483

[2] train-rmse:108.254784 test-rmse:109.401009

[3] train-rmse:87.554665 test-rmse:89.308937

[4] train-rmse:73.102173 test-rmse:75.562843

[5] train-rmse:63.046494 test-rmse:66.201591

[6] train-rmse:56.217892 test-rmse:60.152676

[7] train-rmse:51.677689 test-rmse:56.163967

[8] train-rmse:48.683964 test-rmse:53.415508

[9] train-rmse:46.614704 test-rmse:51.834106

[10] train-rmse:45.272217 test-rmse:50.704090

[11] train-rmse:44.114281 test-rmse:49.873360

[12] train-rmse:43.278046 test-rmse:49.358967

[13] train-rmse:42.590752 test-rmse:49.080044

[14] train-rmse:41.921028 test-rmse:48.665916

[15] train-rmse:41.390240 test-rmse:48.460659

[16] train-rmse:40.819225 test-rmse:48.010067

[17] train-rmse:40.469097 test-rmse:47.874275

[18] train-rmse:39.996845 test-rmse:47.717457

[19] train-rmse:39.517906 test-rmse:47.545284

[20] train-rmse:39.126595 test-rmse:47.425121

[21] train-rmse:38.844265 test-rmse:47.359810

[22] train-rmse:38.535809 test-rmse:47.298771

[23] train-rmse:38.155376 test-rmse:47.170902

[24] train-rmse:37.888538 test-rmse:47.161980

[25] train-rmse:37.760956 test-rmse:47.144623

[26] train-rmse:37.513863 test-rmse:47.180191

[27] train-rmse:37.162022 test-rmse:47.015213

[28] train-rmse:36.765438 test-rmse:47.023064

[29] train-rmse:36.570881 test-rmse:46.943287

[30] train-rmse:36.220703 test-rmse:46.856060

[31] train-rmse:35.880997 test-rmse:46.805138

[32] train-rmse:35.612289 test-rmse:46.851452

[33] train-rmse:35.238930 test-rmse:46.679077

[34] train-rmse:35.095661 test-rmse:46.632988

[35] train-rmse:34.675373 test-rmse:46.672993

[36] train-rmse:34.585720 test-rmse:46.667061

[37] train-rmse:34.426010 test-rmse:46.647713

[38] train-rmse:34.297394 test-rmse:46.614658

[39] train-rmse:33.960018 test-rmse:46.580093

[40] train-rmse:33.766003 test-rmse:46.579155

[41] train-rmse:33.466621 test-rmse:46.551735

[42] train-rmse:33.405533 test-rmse:46.549244

[43] train-rmse:33.281002 test-rmse:46.524220

[44] train-rmse:32.927509 test-rmse:46.502731

[45] train-rmse:32.744095 test-rmse:46.448864

[46] train-rmse:32.486454 test-rmse:46.389019

[47] train-rmse:32.208641 test-rmse:46.341278

[48] train-rmse:32.075035 test-rmse:46.370869

[49] train-rmse:32.030785 test-rmse:46.403744

[50] train-rmse:31.885212 test-rmse:46.404575

[51] train-rmse:31.673975 test-rmse:46.329735

[52] train-rmse:31.532917 test-rmse:46.300030

[53] train-rmse:31.274483 test-rmse:46.298008

[54] train-rmse:30.984734 test-rmse:46.348545

[55] train-rmse:30.754656 test-rmse:46.384239

[56] train-rmse:30.509026 test-rmse:46.446995

[57] train-rmse:30.233519 test-rmse:46.465065

[58] train-rmse:30.003153 test-rmse:46.424812

[59] train-rmse:29.776564 test-rmse:46.401161

[60] train-rmse:29.557426 test-rmse:46.380836

[61] train-rmse:29.391209 test-rmse:46.382057

[62] train-rmse:29.259430 test-rmse:46.390362

[63] train-rmse:28.972832 test-rmse:46.347824

[64] train-rmse:28.894812 test-rmse:46.364208

[65] train-rmse:28.784081 test-rmse:46.401184

[66] train-rmse:28.739403 test-rmse:46.383190

[67] train-rmse:28.542549 test-rmse:46.387245

[68] train-rmse:28.349623 test-rmse:46.355984

[69] train-rmse:28.161774 test-rmse:46.354523

[70] train-rmse:27.994087 test-rmse:46.360317

[71] train-rmse:27.808891 test-rmse:46.375362

[72] train-rmse:27.635410 test-rmse:46.376041

[73] train-rmse:27.453598 test-rmse:46.393291

[74] train-rmse:27.247025 test-rmse:46.387558

[75] train-rmse:27.162207 test-rmse:46.394196

[76] train-rmse:26.984289 test-rmse:46.438866

[77] train-rmse:26.765007 test-rmse:46.421642

[78] train-rmse:26.670570 test-rmse:46.427498

[79] train-rmse:26.546865 test-rmse:46.426102

[80] train-rmse:26.388199 test-rmse:46.433708

[81] train-rmse:26.272144 test-rmse:46.415291

[82] train-rmse:26.191458 test-rmse:46.424191

[83] train-rmse:25.987301 test-rmse:46.412895

[84] train-rmse:25.829067 test-rmse:46.412876

[85] train-rmse:25.646105 test-rmse:46.436073

[86] train-rmse:25.483675 test-rmse:46.461494

[87] train-rmse:25.404697 test-rmse:46.501991

[88] train-rmse:25.239088 test-rmse:46.522556

[89] train-rmse:25.156235 test-rmse:46.511398

[90] train-rmse:25.116955 test-rmse:46.521702

[91] train-rmse:25.009691 test-rmse:46.523754

[92] train-rmse:24.908003 test-rmse:46.542919

[93] train-rmse:24.740252 test-rmse:46.521526

[94] train-rmse:24.653141 test-rmse:46.506599

[95] train-rmse:24.565540 test-rmse:46.508568

[96] train-rmse:24.431156 test-rmse:46.525288

[97] train-rmse:24.324286 test-rmse:46.499508

[98] train-rmse:24.268969 test-rmse:46.522537

[99] train-rmse:24.089361 test-rmse:46.550568

[100] train-rmse:23.967646 test-rmse:46.544228

Again we can experiment with different tuning parameter settings to find an “optimal” choice.

After choosing some "optimal" settings based on several iterations of the process above.

yhat = predict(bst2,Q.train.x)

plot(as.numeric(Q.train.y),yhat)

abline(0,1,col="blue",lwd=2)

ytest = predict(bst2,Q.test.x)

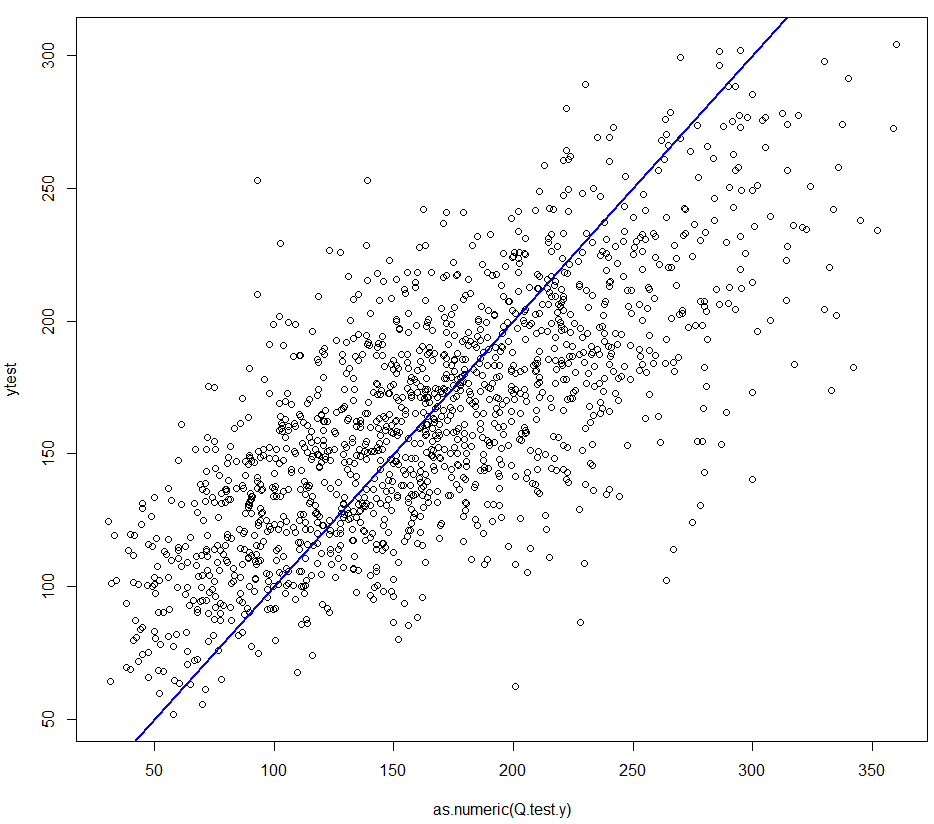
plot(as.numeric(Q.test.y),ytest)

abline(0,1,lwd=2,col="blue")

RMSEP.test = sqrt(mean((as.numeric(Q.test.y)-ytest)^2))

RMSEP.test

[1] 44.98326



Another approach for tuning an XGBoost model

Incorporate more tuning parameters with an automatic k-fold CV function for XGBoost modeling using the function xgb.cv. We can specify model parameters/settings in a separate object called params. These parameter settings are then supplied to the xgb.cv function as the main argument. We can easily make tweaks to the parameter settings this way by altering our script (or upper arrowing at the command line prompt).

params = list(booster="gbtree",objective="reg:linear",eta=0.10,gamma=0,  
max\_depth=6,min\_child\_weight=5,subsample=.75,colsample\_bytree=1,  
lambda=.25,alpha=.75)

cv.results = xgb.cv(params=params,data=dtrain,nrounds=500,nthread=6,nfold=5,  
stratified=F,print\_every\_n=10,early\_stop\_rounds=20,maximize=F)

[1] train-rmse:160.487723+0.510259 test-rmse:160.767731+2.312713

[11] train-rmse:68.123828+0.396307 test-rmse:73.207399+1.813751

[21] train-rmse:40.657574+0.321892 test-rmse:51.652366+1.642452

[31] train-rmse:32.428023+0.253995 test-rmse:47.571361+1.435852

[41] train-rmse:28.703570+0.120143 test-rmse:46.788598+1.300739

[51] train-rmse:26.205735+0.228570 test-rmse:46.540770+1.204045

[61] train-rmse:24.006883+0.216468 test-rmse:46.432765+1.188528

[71] train-rmse:22.260064+0.162466 test-rmse:46.298676+1.179805

[81] train-rmse:20.611996+0.243087 test-rmse:46.213933+1.188451

[91] train-rmse:19.061512+0.373005 test-rmse:46.102297+1.152676

[101] train-rmse:17.665281+0.414063 test-rmse:46.043039+1.172232

[111] train-rmse:16.365365+0.355423 test-rmse:46.010773+1.217890

[121] train-rmse:15.156943+0.346585 test-rmse:45.975129+1.260853

[131] train-rmse:14.072125+0.403166 test-rmse:45.953173+1.266900

[141] train-rmse:13.186142+0.332996 test-rmse:45.951350+1.309663

[151] train-rmse:12.260451+0.387631 test-rmse:45.971460+1.294379

[161] train-rmse:11.451035+0.327721 test-rmse:45.981276+1.281128

[171] train-rmse:10.658594+0.277390 test-rmse:46.023682+1.261616

[181] train-rmse:9.980724+0.282886 test-rmse:46.049023+1.282773

[191] train-rmse:9.315448+0.288140 test-rmse:46.049345+1.282097

[201] train-rmse:8.697107+0.264008 test-rmse:46.100547+1.284265

[211] train-rmse:8.174776+0.229750 test-rmse:46.080152+1.302521

[221] train-rmse:7.655820+0.212821 test-rmse:46.075861+1.310063

[231] train-rmse:7.176794+0.220446 test-rmse:46.103204+1.336674

[241] train-rmse:6.731039+0.203433 test-rmse:46.094331+1.324923

[251] train-rmse:6.329401+0.205350 test-rmse:46.117740+1.336548

[261] train-rmse:5.942507+0.189354 test-rmse:46.111285+1.347797

[271] train-rmse:5.550893+0.174595 test-rmse:46.130218+1.360274

[281] train-rmse:5.224519+0.180137 test-rmse:46.136516+1.358768

[291] train-rmse:4.907075+0.208237 test-rmse:46.135010+1.361304

[301] train-rmse:4.604692+0.181721 test-rmse:46.149315+1.346371

[311] train-rmse:4.330574+0.174409 test-rmse:46.164750+1.347417

[321] train-rmse:4.081842+0.181616 test-rmse:46.148821+1.361858

[331] train-rmse:3.847405+0.192433 test-rmse:46.149587+1.364976

[341] train-rmse:3.645363+0.173178 test-rmse:46.146191+1.352373

[351] train-rmse:3.448408+0.170837 test-rmse:46.154524+1.360163

[361] train-rmse:3.254288+0.173798 test-rmse:46.148927+1.354085

[371] train-rmse:3.090418+0.162472 test-rmse:46.160371+1.348238

[381] train-rmse:2.916860+0.172654 test-rmse:46.170451+1.347166

[391] train-rmse:2.764806+0.186758 test-rmse:46.176733+1.342971

[401] train-rmse:2.625758+0.190420 test-rmse:46.183365+1.347113

[411] train-rmse:2.489339+0.188759 test-rmse:46.188194+1.344799

[421] train-rmse:2.369916+0.190126 test-rmse:46.190626+1.344480

[431] train-rmse:2.259506+0.202255 test-rmse:46.196321+1.345106

[441] train-rmse:2.150337+0.207850 test-rmse:46.203628+1.342734

[451] train-rmse:2.047682+0.209050 test-rmse:46.208388+1.345766

[461] train-rmse:1.950124+0.212687 test-rmse:46.218450+1.353004

[471] train-rmse:1.864919+0.221258 test-rmse:46.214563+1.357860

[481] train-rmse:1.779969+0.227766 test-rmse:46.216630+1.355753

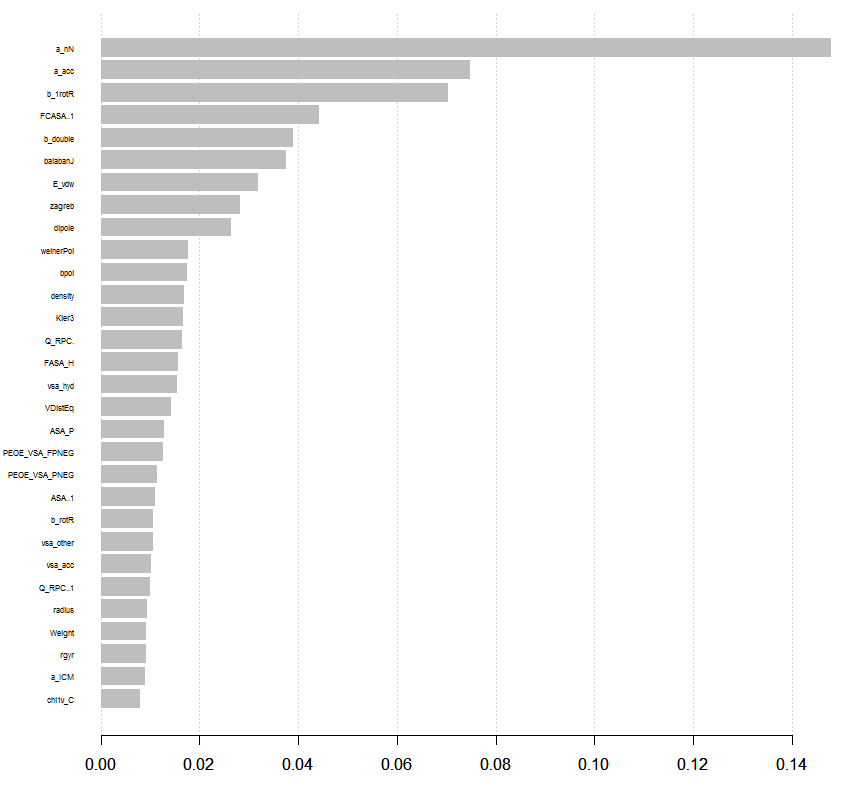
[491] train-rmse:1.701813+0.238945 test-rmse:46.219482+1.348654

[500] train-rmse:1.641478+0.243840 test-rmse:46.221103+1.343747

Assessing variable importance

mat = xgb.importance(feature\_names=colnames(Q.train.x),model=bst2)

xgb.plot.importance(importance\_matrix=mat[1:30])



**10.8 - 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 OLS regression model to the observations in that node, or for a binary classification problem we might fit a logistic regression model in each node, etc.

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 OLS 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 below and is taken from the website *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 basically how it works:

Suppose that  is the case whose unknown target value is to be predicted, and is one of 's nearest neighbors in the training data. The target value of for  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  and  be  and  respectively. The model then predicts that the difference between the target values of  and  is given by . The value of  predicted by neighbor  is adjusted to reflect this difference, so that Cubist uses  instead of 's raw target value, . Once this adjustment has been made for each of nearest neighboring the values, the adjusted response values for each of the neighbors are then averaged to produce the ultimate predicted value for .  
  
Diagram and formulae:

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.

cubist.cv = function(x,y,p=.667,B=10,committees=1,neighbors=0) {

n <- length(y)

MSE <- rep(0,B)

MAE = rep(0,B)

MAPE = rep(0,B)

for (i in 1:B) {

ss <- floor(n\*p)

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

fit2 <- cubist(x[sam,],y[sam],committees=committees,neighbors=neighbors)

ynew <- predict(fit2,newdata=x[-sam,],neighbors=neighbors)

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

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

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

}

cat("RMSEP =",sqrt(mean(MSE))," MAEP=",mean(MAE)," MAPEP=",mean(MAPE))

cv = return(data.frame(RMSEP=sqrt(MSE),MAEP=MAE,MAPEP=MAPE))

}

We now consider the usual examples.

**Example 10.7 - Boston Housing Data (cont’d)**

> names(Boston.train)

[1] "medv" "age" "B" "chas" "crim" "dis" "indus" "lstat"

[9] "nox" "ptratio" "rad" "rm" "tax" "zn"

> bos.x = Boston.train[,-1]

> bos.y = Boston.train[,1]

> bos.cub = cubist(bos.x,bos.y,committees=1)

> bos.cub

Call:

cubist.default(x = bos.x, y = bos.y, committees = 1)

Number of samples: 400

Number of predictors: 13

Number of committees: 1

Number of rules: 6

> summary(bos.cub)

Call:

cubist.default(x = bos.x, y = bos.y, committees = 1)

Cubist [Release 2.07 GPL Edition] Tue Feb 23 15:08:31 2016

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

Target attribute `outcome'

Read 400 cases (14 attributes) from undefined.data

Model:

Rule 1: [24 cases, mean 2.204103, range 1.609438 to 2.70805, est err 0.211997]

if

nox > 0.668

nox <= 0.693

then

outcome = 10.394505 - 10.91 nox - 0.03 lstat + 0.027 dis - 0.0025 crim

+ 0.00014 B + 0.0014 rad - 6e-005 tax - 0.004 ptratio

Rule 2: [63 cases, mean 2.690486, range 1.974081 to 3.314186, est err 0.121927]

if

nox > 0.693

then

outcome = 2.733668 + 0.104 dis - 0.0256 lstat - 0.0192 crim + 0.00056 B

+ 0.27 nox

Rule 3: [24 cases, mean 2.886870, range 2.091864 to 3.912023, est err 0.216421]

if

dis <= 2.072

lstat > 5.12

nox <= 0.668

rm <= 6.315

then

outcome = 4.509502 - 0.418 dis - 0.0273 lstat + 0.0053 rad - 0.00024 tax

- 0.016 ptratio - 0.0024 crim - 0.15 nox - 0.0006 age

+ 0.018 rm + 9e-005 B

Rule 4: [147 cases, mean 2.996866, range 2.476538 to 3.589059, est err 0.084614]

if

dis > 2.072

nox <= 0.668

rm <= 6.315

then

outcome = 3.943254 - 0.0119 lstat - 0.0029 age - 0.032 ptratio

- 0.033 dis - 0.0004 tax + 0.0076 rad + 0.038 rm

Rule 5: [95 cases, mean 3.242678, range 2.646175 to 3.912023, est err 0.088053]

if

lstat > 5.12

nox <= 0.668

rm > 6.315

then

outcome = 2.510734 + 0.261 rm + 0.0178 crim - 0.0209 lstat - 0.00065 tax

- 0.028 dis - 0.022 ptratio - 0.0017 age + 0.0025 rad

- 0.07 nox

Rule 6: [47 cases, mean 3.612082, range 3.113515 to 3.912023, est err 0.109892]

if

lstat <= 5.12

then

outcome = 2.630062 + 0.0673 crim - 0.0472 lstat + 0.191 rm + 0.0028 rad

- 0.01 dis - 0.009 ptratio - 0.15 nox - 0.0001 tax + 0.0001 B

Evaluation on training data (400 cases):

Average |error| 0.108080

Relative |error| 0.36

Correlation coefficient 0.93

Attribute usage:

Conds Model

88% 63% nox

66% 78% rm

43% 100% dis

42% 100% lstat

84% ptratio

84% rad

84% tax

66% age

63% crim

40% B

Time: 0.0 secs

We can also estimate variable importance by using the varImp( ) command from the caret library.  
  
> library(caret)  
> varImp(bos.cub)

Overall

nox 75.5

rm 72.0

dis 71.5

lstat 71.0

ptratio 42.0

rad 42.0

tax 42.0

age 33.0

crim 31.5

B 20.0

chas 0.0

indus 0.0

zn 0.0

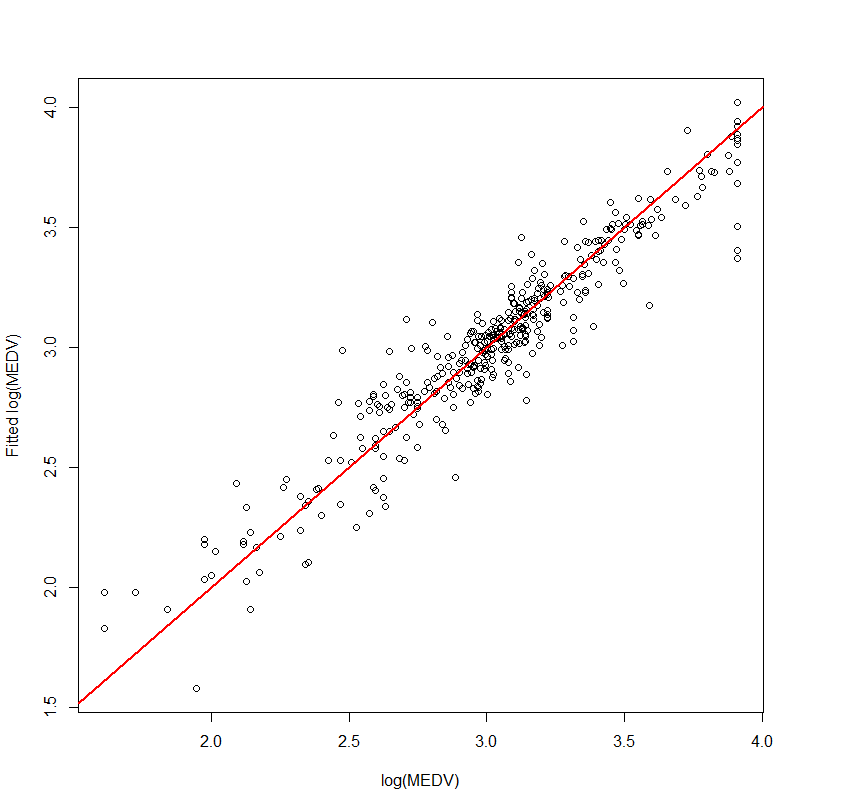
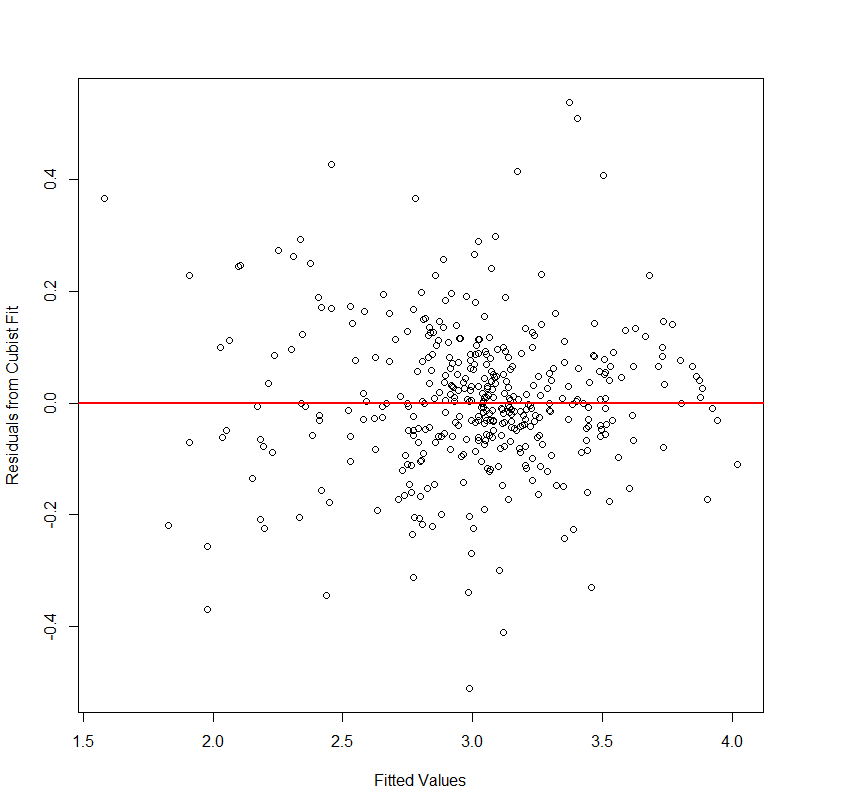
> plot(bos.y,predict(bos.cub,newdata=bos.x),xlab="log(MEDV)",   
ylab="Fitted log(MEDV)")

> abline(0,1,col="red",lwd=2)  
> cor(bos.y,predict(bos.cub,newdata=bos.x))

[1] 0.94906

> cor(bos.y,predict(bos.cub,newdata=bos.x))^2

[1] 0.9007148

> yhat = predict(bos.cub,newdata=bos.x)

> plot(yhat,resid.cub,xlab="Fitted Values",ylab="Residuals from Cubist Fit")

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

Boost the cubist model (*M = 10*)

> bos.cub10 = cubist(bos.x,bos.y,committees=10)

> plot(bos.y,yhat,xlab="log(MEDV)",ylab="Fitted log(MEDV)")

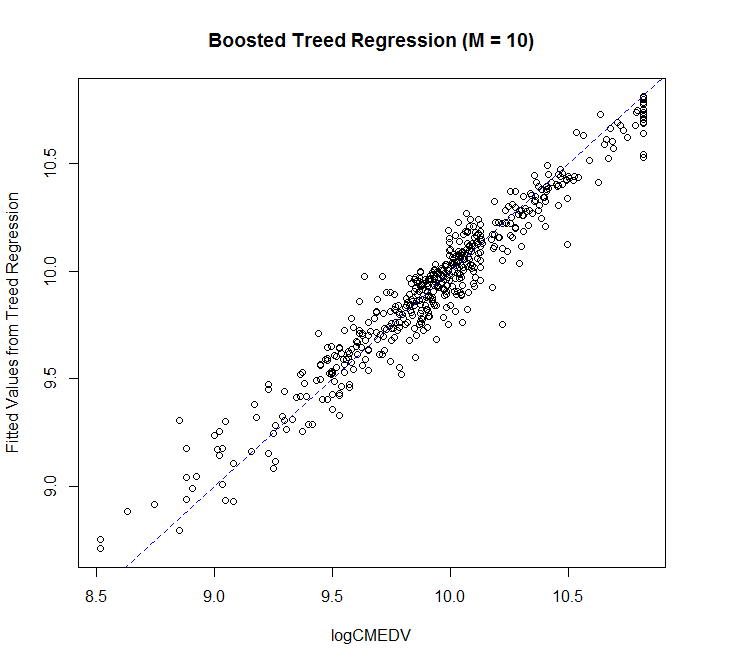
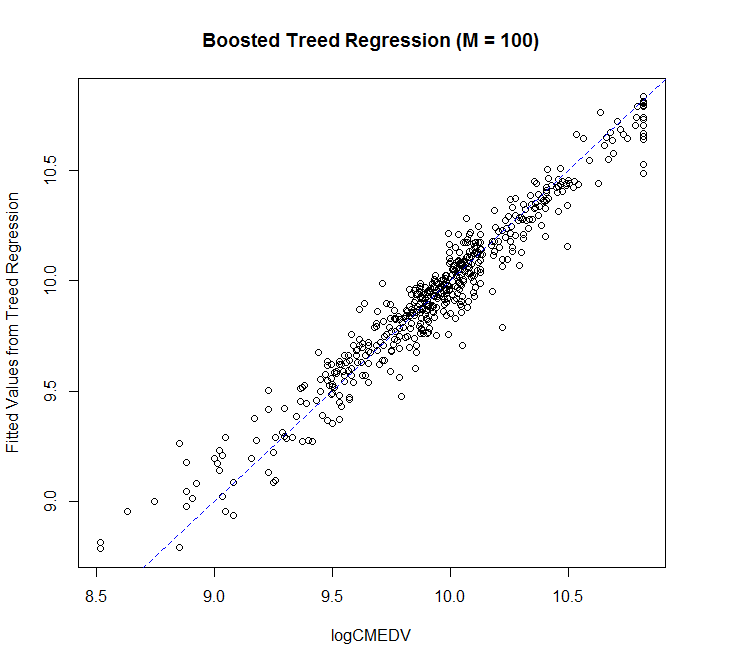
> abline(0,1,col="red",lwd=2)

> cor(bos.y,yhat)

[1] 0.9664181

> cor(bos.y,yhat)^2 🡨

[1] 0.9339639

Boost the cubist (*M = 15*) 🡨 better than *M = 10*  
> bos.cub15 = cubist(bos.x,bos.y,committees=15)

> yhat = predict(bos.cub100,newdata=bos.x)

> cor(bos.y,yhat)

[1] 0.9670039

> cor(bos.y,yhat)^2 🡨

[1] 0.9350965  
  
  
Boost the cubist (*M = 100*) 🡨 worse than *M = 15*  
> bos.cub100 = cubist(bos.x,bos.y,committees=100))

> yhat100 = predict(bos.cub100,newdata=bos.x)

> cor(bos.y,yhat)

[1] 0.9651285

> cor(bos.y,yhat)^2

[1] 0.9314731

**Using cross-validation to fine tune our treed regression model**We can use the cubist.sscv function above to decide what level of boosting to do (M) and whether to use a nearest neighbor adjustment when making predictions.

> results = cubist.sscv(bos.x,bos.y,committees=1,neighbors=4,B=20)

RMSEP = 0.1917183 MAEP= 0.1310246 MAPEP= 0.04574339

> results = cubist.sscv(bos.x,bos.y,committees=5,neighbors=4,B=20)

RMSEP = 0.1809252 MAEP= 0.1235103 MAPEP= 0.0436853

> results = cubist.sscv(bos.x,bos.y,committees=10,neighbors=4,B=20)

RMSEP = 0.1695083 MAEP= 0.1183092 MAPEP= 0.04217953

> results = cubist.sscv(bos.x,bos.y,committees=15,neighbors=4,B=20)

RMSEP = 0.1638071 MAEP= 0.1131867 MAPEP= 0.04036523

> results = cubist.sscv(bos.x,bos.y,committees=20,neighbors=4,B=20)

RMSEP = 0.16147 MAEP= 0.1144339 MAPEP= 0.0403694

> results = cubist.sscv(bos.x,bos.y,committees=50,neighbors=4,B=25)

RMSEP = 0.1568915 MAEP= 0.1060937 MAPEP= 0.03722768

> results = cubist.sscv(bos.x,bos.y,committees=75,neighbors=4,B=25)

RMSEP = 0.1538194 MAEP= 0.104482 MAPEP= 0.03716319

> results = cubist.sscv(bos.x,bos.y,committees=100,neighbors=4,B=25)

RMSEP = 0.1599598 MAEP= 0.108521 MAPEP= 0.03823639  
> results = cubist.sscv(bos.x,bos.y,committees=75,neighbors=5,B=25)

RMSEP = 0.1477676 MAEP= 0.1021941 MAPEP= 0.03596935

> results = cubist.sscv(bos.x,bos.y,committees=75,neighbors=6,B=25)

RMSEP = 0.1581457 MAEP= 0.1075315 MAPEP= 0.03814671

> results = cubist.sscv(bos.x,bos.y,committees=75,neighbors=7,B=25)

RMSEP = 0.1554667 MAEP= 0.106847 MAPEP= 0.03783427

> results = cubist.sscv(bos.x,bos.y,committees=75,neighbors=8,B=25)

RMSEP = 0.1496608 MAEP= 0.1037669 MAPEP= 0.03660336

> results = cubist.sscv(bos.x,bos.y,committees=75,neighbors=9,B=25)

RMSEP = 0.1519383 MAEP= 0.1049568 MAPEP= 0.03664266

“Best” Cubist Model   
  
> bos.cub = cubist(bos.x,bos.y,data=Boston.train,committees=75,neighbors=5)

> bos.cub

Call:

cubist.default(x = bos.x, y = bos.y, committees = 75, data =

Boston.train, neighbors = 5)

Number of samples: 400

Number of predictors: 13

Number of committees: 75

Number of rules per committee: 6, 5, 4, 5, 7, 5, 8, 4, 6, 6, 6, 4, 6, 7, 6, 3, 7, 9, 4, 6 ...

> ypred.cub = exp(predict(bos.cub,newdata=Boston.test[,-1],neighbors=9))

> RMSEP.cub = sqrt(mean((ypred.cub-yact)^2))

> RMSEP.cub

[1] 2.525938

Treed Regression (Cubist) model beats Random Forest and barely beats the Gradient Boosted Trees models for these data. Notice that I increased the number of neighbors to the maximum when making predictions for the test cases.