Random_Forests_GBM

Neha

10/02/2020

Random Forests

Random forests build lots of bushy trees, and then average them to reduce the variance.

```
require(randomForest)
```

```
## Loading required package: randomForest
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
require(MASS)
```

```
## Loading required package: MASS
```

```
set.seed(101)
dim(Boston)
```

```
## [1] 506  14
```

```
train=sample(1:nrow(Boston),300)
?Boston
```

medv: the median housing value (in $1K dollars)

```
rf.boston=randomForest(medv~.,data=Boston,subset=train)
rf.boston
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, subset = train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##           Mean of squared residuals: 12.30718
##                     % Var explained: 85.13
```

The MSR and % variance explained are based on OOB or out-of-bag estimates, a very clever device in random forests to get honest error estimates. The model reports that mtry=4, which is the number of variables randomly chosen at each split. Since p=13 here, we could try all 13 possible values of mtry. We will do so, record the results, and make a plot.

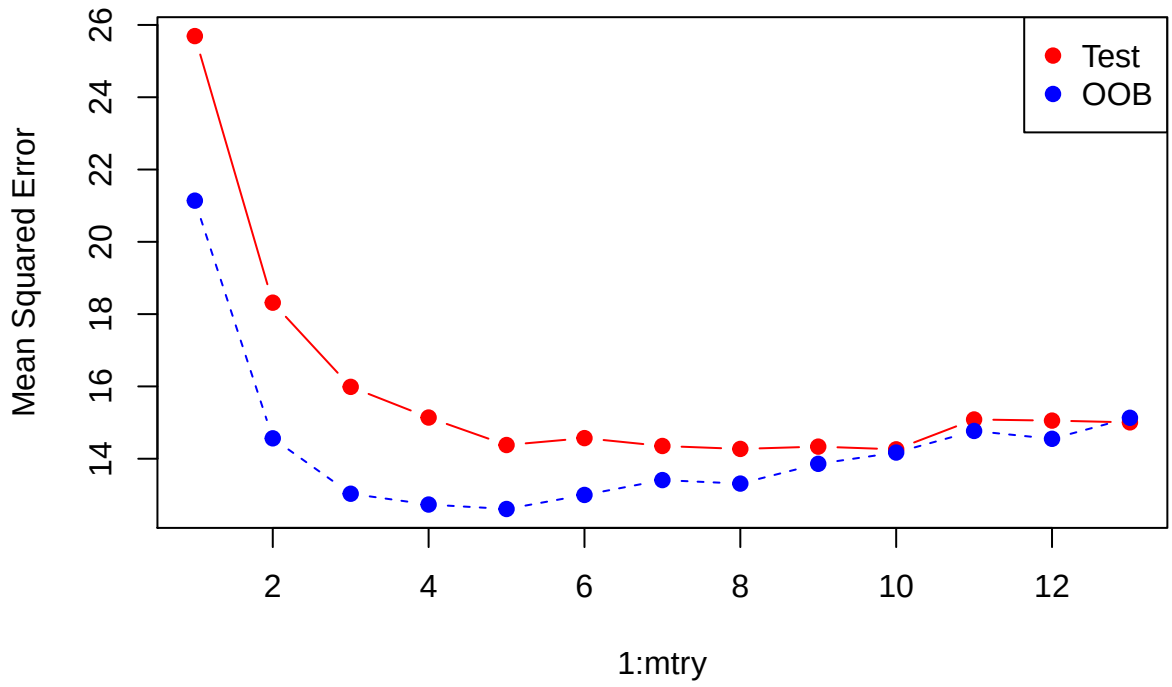
Out of bag error basically seems like a kind of validation set error. So the, pretty much, only tuning parameter in a randomForest is the variable called mtry, the argument called mtry, which is the number of variables that are selected at each split of each tree when you come to make a split. This is how decorrelation is done.

The number of trees will be limited to 400 because thats a lot anyway.

```
oob.err=double(13)
test.err=double(13)
for(mtry in 1:13){
  #mtry = 1 to 13 parameters
  #number of trees = 400
  fit=randomForest(medv~.,data=Boston,subset=train,mtry=mtry,ntree=400)
  oob.err[mtry]=fit$mse[400]
  pred=predict(fit,Boston[-train,])
  test.err[mtry]=with(Boston[-train,],mean((medv-pred)^2))
  cat(mtry," ")
}
```

```
## 1  2  3  4  5  6  7  8  9  10  11  12  13
```

```
matplot(1:mtry,cbind(test.err,oob.err),pch=19,col=c("red","blue"),type="b",ylab="Mean Squared Error")
legend("topright",legend=c("Test","OOB"),pch=19,col=c("red","blue"))
```



so since the out-of-bag was computer on one data set and the test error on a different data set, and they weren't very large, these differences are pretty much well within the standard errors. The error is not too bad, we can see the optimal error is somewhere around 8.

Using all 13 Variables like at the end is bagging.

RandomForests reduce the variance of the trees by averaging. So it grows big bushy trees, and then gets rid of the variance by averaging. Boosting, on the other hand, is really going off to bias.So boosting grows smaller, stubbier trees and goes at the bias.

BOOSTING

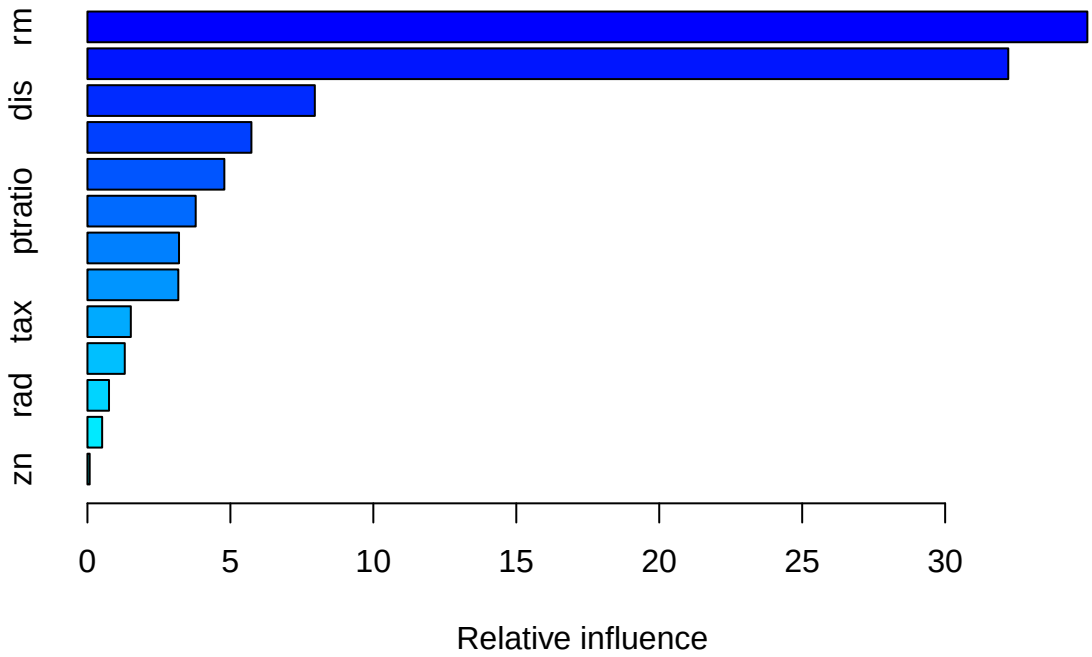
Boosting builds lots of smaller trees. Unlike random forests, each new tree in boosting tries to patch up the deficiencies of the current ensemble.

```
#install.packages('gbm')
require(gbm)
```

```
## Loading required package: gbm
```

```
## Loaded gbm 2.1.5
```

```
#use is similar to Random Forest
boost.boston=gbm(medv~.,data=Boston[train,],distribution="gaussian",n.trees=10000,shrinkage=0.01,intera
#summary gives a variable importance plot
summary(boost.boston)
```



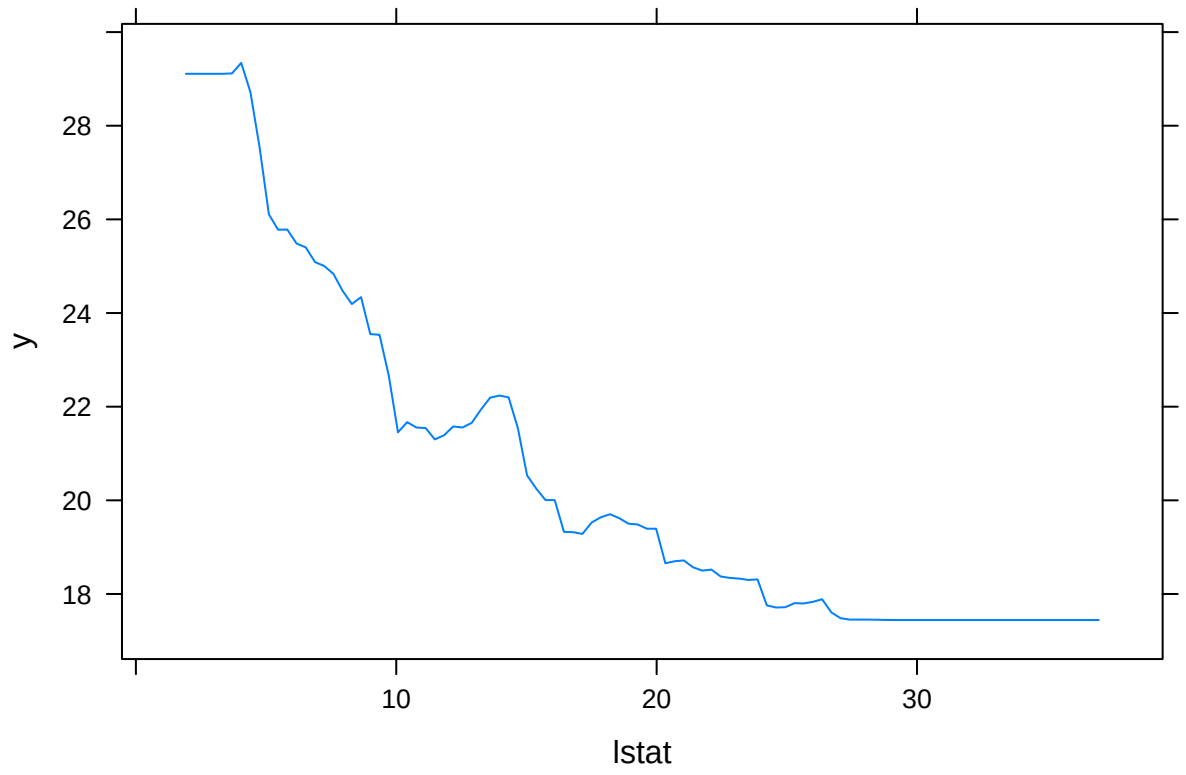

```
##             var     rel.inf
## rm           rm 34.97452489
## lstat     lstat 32.20820578
```

```
## dis         dis  7.95358257
## crim       crim  5.73513691
## nox         nox  4.78842194
## ptratio ptratio  3.78642992
## age         age  3.20324899
## black     black  3.17914690
## tax         tax  1.51752430
## chas       chas  1.30550424
## rad         rad  0.75495021
## indus     indus  0.51457132
## zn           zn  0.07875203
```

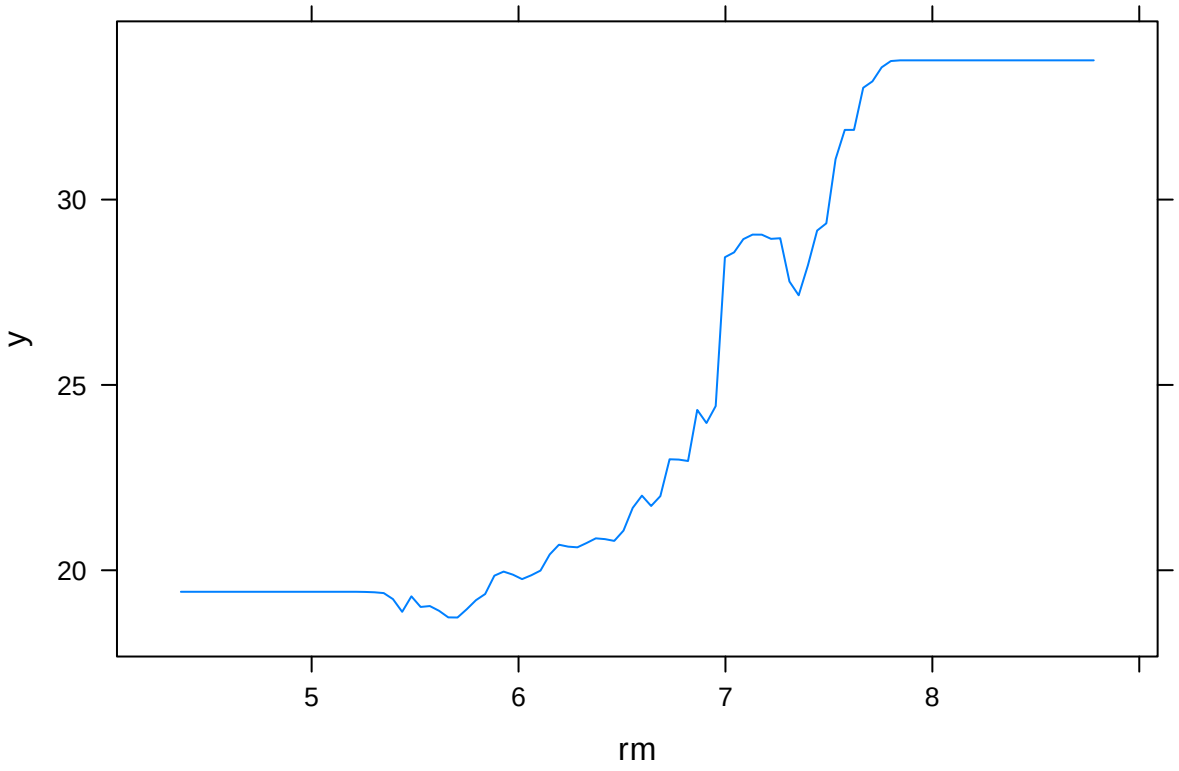
Summary Plot There's two variables that seem to be the most important. There's the number of rooms. The second variable is lstat, which is the percentage of lower economic status people in the community. Those are the two most important variables. They aren't all labeled in this plot here, because of the font size.

PARTIAL DEPENDENCE PLOT

```
plot(boost.boston,i="lstat")
```




```
plot(boost.boston,i="rm")
```



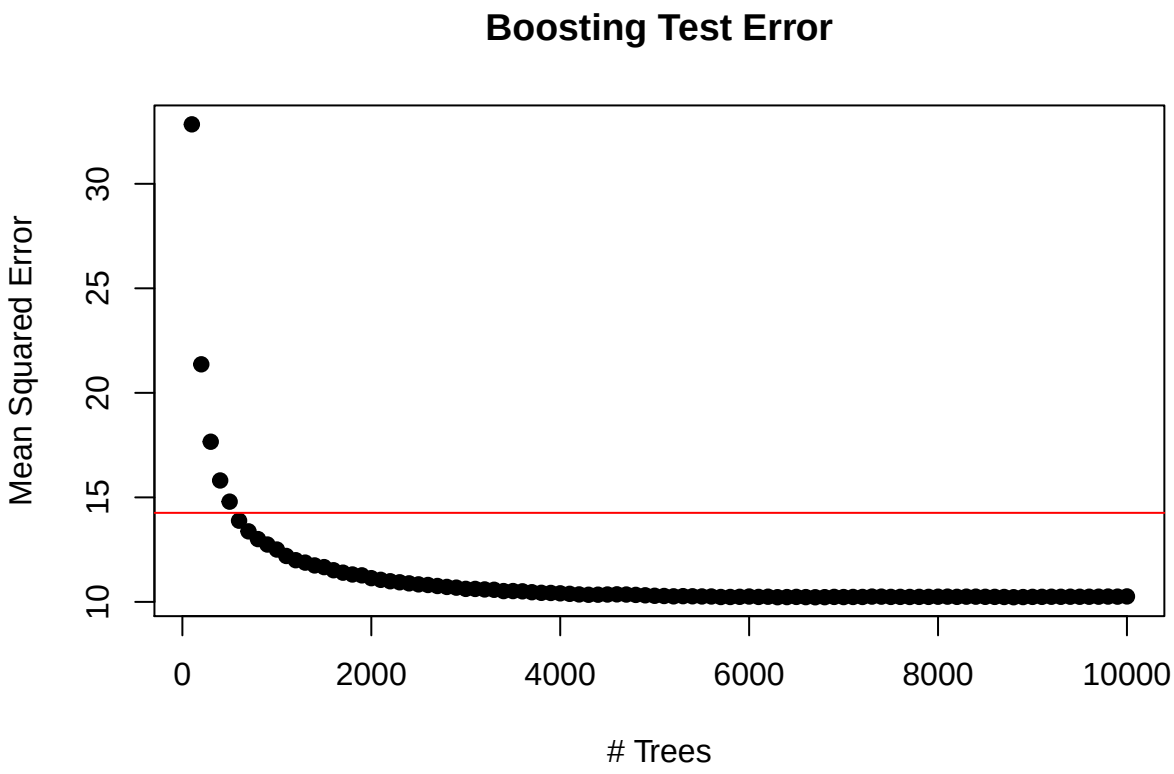
Rough relationship, but it shows us that the higher the proportion of lower status people in the suburb, the lower the value of the house, the housing prices-not a big surprise there. Reversed relationship with the number of rooms. The average number of rooms in the house increases, the price increases.

Lets make a prediction on the test set. With boosting, the number of trees is a tuning parameter, and if we have too many we can overfit. So we should use cross-validation to select the number of trees. We will leave this as an exercise. Instead, we will compute the test error as a function of the number of trees, and make a plot.

```
n.trees=seq(from=100,to=10000,by=100)
predmat=predict(boost.boston,newdata=Boston[-train,],n.trees=n.trees)
dim(predmat)
```

```
## [1] 206 100
```

```
berr=with(Boston[-train,],apply( (predmat-medv)^2,2,mean))
plot(n.trees,berr,pch=19,ylab="Mean Squared Error", xlab="# Trees",main="Boosting Test Error")
abline(h=min(test.err),col="red")
```


Boosting Test Error
Mean Squared Error
10
15
20
25
30
0
2000
4000
6000
8000
10000
Trees