

Exploratory Data Mining via Search Strategies Lab #3

Ross Jacobucci & Kevin J. Grimm

Outline

This script will go over Decision Trees and generalizations in R.

Resources:

Basic intro to Decision Trees: <http://www.statmethods.net/advstats/cart.html>

Full list of data mining packages in R:

<http://cran.r-project.org/web/views/MachineLearning.html>

Two packages will be used and their caret equivalents:

- ▶ **rpart** (tree accomplishes very similar thing):<http://cran.r-project.org/web/packages/rpart/vignettes/longintro.pdf>
- ▶ **party**:
<http://cran.r-project.org/web/packages/party/vignettes/party.pdf>

In **caret**, method =

- ▶ “rpart” – tuning = cp (complexity parameter)
- ▶ “rpart2” – tuning = maxdepth
- ▶ “rpartCost” – tuning = cp and cost
- ▶ “ctree” – tuning = mincriterion (p value thresholds)
- ▶ “ctree2” – tuning = maxdepth

Lets load the main packages

```
library(caret)
library(rpart)
library(pROC)
library(randomForest)
library(gbm)
library(ISLR)
library(party)
library(MASS) # for boston data
data(Boston)
```

Regression

Regression (continuous outcome)

Use rpart first with the Boston data use regression first – predicting median value of homes

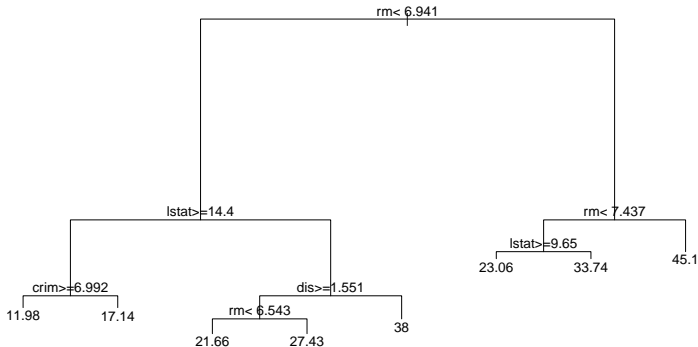
```
#str(Boston)  
  
# lets get a baseline with linear regression  
lm.Boston <- lm(medv ~., data=Boston)  
#summary(lm.Boston)
```

We do pretty well with linear regression R-squared of .74

CART

How about if we just blindly apply Decision Trees

```
rpart.Boston <- rpart(medv ~., data=Boston)
#summary(rpart.Boston)
plot(rpart.Boston);text(rpart.Boston)
```



CART Continued

```
pred1 <- predict(rpart.Boston)
cor(pred1, Boston$medv)**2
```

```
## [1] 0.8075721
```

Doing really well – Rsquared = 0.81

this can be hard to interpret, so I like to look at a different output
rpart.Boston

```
## n= 506
```

```
##
```

```
## node), split, n, deviance, yval
```

```
##      * denotes terminal node
```

```
##
```

```
## 1) root 506 42716.3000 22.53281
```

```
## 2) rm< 6.941 430 17317.3200 19.93372
```

```
## 4) lstat>=14.4 175 3373.2510 14.95600
```

```
## 8) crim>=6.99237 74 1085.9050 11.97838 *
```

```
## 9) crim< 6.99237 101 1150.5370 17.13762 *
```

```
## 5) lstat< 14.4 255 6632.2170 23.34980
```

```
## 10) dis>=1.5511 248 3658.3930 22.93629
```

```
## 20) rm< 6.543 193 1589.8140 21.65648 *
```

```
## 21) rm>=6.543 55 643.1691 27.42727 *
```

```
## 11) dis< 1.5511 7 1429.0200 38.00000 *
```

```
## 3) rm>=6.941 76 6059.4190 37.23816
```

```
## 6) rm< 7.437 46 1899.6120 32.11304
```

Lasso Regression

What if we tried regularized (penalized) regression instead?

Note: for glmnet, both the x's and y have to be in separate matrices

- and all class = numeric

- don't worry about response, doesn't have to be factor for logistic

- just specify “binomial”

```
y.B <- Boston$medv
x.B <- sapply(Boston[,-14],as.numeric)

# alpha =1 for lasso, 0 for ridge
library(glmnet)
cv <- cv.glmnet(x.B,y.B,alpha=1)
lasso.reg <- glmnet(x.B,y.B,alpha=1,family="gaussian",lambda=cv$lambda.min)

lasso.resp <- predict(lasso.reg,newx=x.B)
cor(y.B,lasso.resp)**2
```

```
##              s0
```

```
## [1,] 0.7403028
```

Taking into account cross-validation, we do worse compared to linear regression with no tuning.

CART Plots

So the plot for rpart didn't come out that well.
Good news, there are better options for plotting.

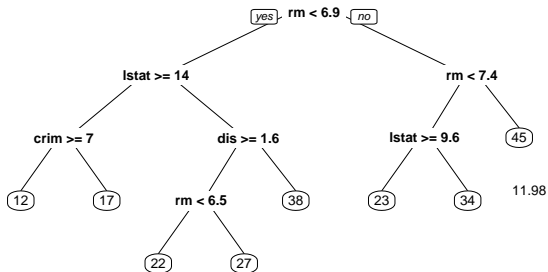
[http://blog.revolutionanalytics.com/2013/06/
plotting-classification-and-regression-trees-with-plotrpart.html](http://blog.revolutionanalytics.com/2013/06/plotting-classification-and-regression-trees-with-plotrpart.html)
Let's load some new packages:

```
library(rattle)
library(rpart.plot)
library(RColorBrewer)
library(partykit)
```

Note: rattle is package that uses a GUI (think SPSS) for data mining applications check out book: <http://www.amazon.com/Data-Mining-Rattle-Excavating-Knowledge/dp/1441998896>
Anyways, let's try some new, prettier plots:

prp()

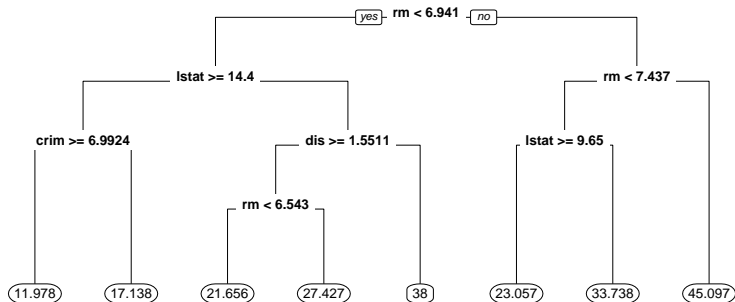
```
# prp(); from rpart.plot  
prp(rpart.Boston);text(rpart.Boston)
```



CART Plotting Continued

Note, `prp()` offers many additional capabilities for tweaking the plot For instance:

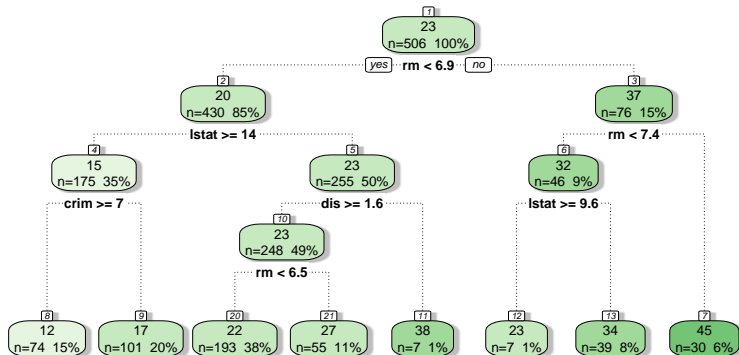
```
# ?prp  
prp(rpart.Boston,varlen=10,digits=5,fallen.leaves=T)
```



CART Plotting Continued

Probably my favorite, `fancyRpartPlot()`

```
#fancyRpartPlot(); from rattle  
fancyRpartPlot(rpart.Boston)
```



Rattle 2016-Aug-12 10:13:47 R.Jacobucci

Conditional Inference Trees

So what about with conditional inference trees?

What if we want a smaller tree? This can be accomplished a number of ways. We can prespecify the maxdepth, the minimum number of people per node, as well as making more restrictive splitting criterion.

Example of prespecifying the depth with ctree()

```
ctree.Boston <- ctree(medv ~., data=Boston)
#plot(ctree.Boston) # too big of a tree
pred2 <- predict(ctree.Boston)
cor(pred2,Boston$medv)**2
```

```
## [1] 0.8746338
```

We do better than rpart, Rsquared = 0.87

Conditional Inference Trees Continued

Biggest difference between `ctree()` and `rpart()` is that `ctree()` does not demonstrate bias with respect to the number of response options, and supposedly had less of a propensity to overfit than `rpart()`.

Note: the models are not optimizing based on `Rsquared`, most likely `MSE`

So what do we think now? Are we happy with results? Remember, decision trees are generally quite robust, so it may not be necessary to check assumptions. – See Table 10.1 ESL

But what about generalizability?

Although not as serious as with SVM for instance, Decision Trees have a propensity to overfit, meaning the tree structure won't generalize well

So let's try just creating a simple Training and Test datasets

```
train = sample(dim(Boston)[1], dim(Boston)[1]/2) # half of sample
Boston.train = Boston[train, ]
Boston.test = Boston[-train, ]
```

Linear Regression with CV

```
lm.train <- lm(medv ~., data=Boston.train)

pred.lmTest <- predict(lm.train,Boston.test)
cor(pred.lmTest,Boston.test$medv)**2
```

```
## [1] 0.7068753
```

Note: we are taking our lm object trained on the train dataset, and using these fixed coefficients to predict values on the test dataset.

In SEM, this is referred to as a tight replication strategy No difference in using a test dataset – both R^2 are 0.74

How about with rpart?

CART CV

```
rpart.train <- rpart(medv ~., data=Boston.train)

pred.rpartTest <- predict(rpart.train,Boston.test)
cor(pred.rpartTest,Boston.test$medv)**2
```

```
## [1] 0.7299716
```

Not as good – drops from 0.81 to 0.76 – still better than `lm()`

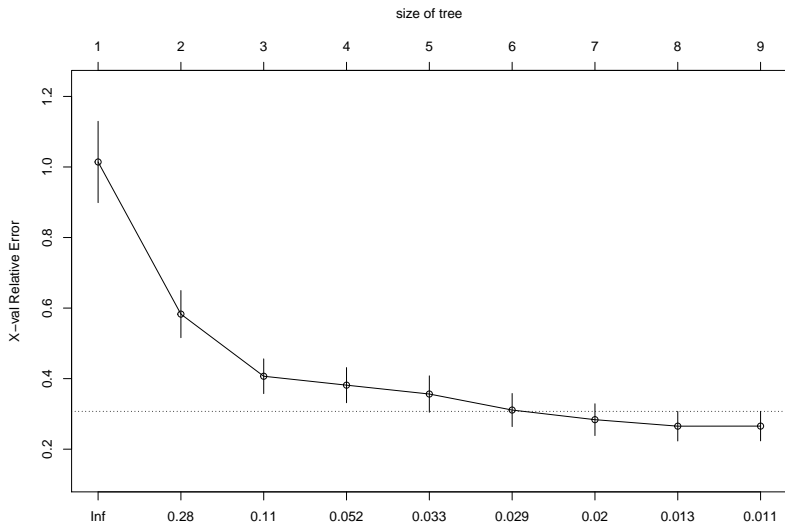
But with `rpart`, it is common to prune trees back. What if we try this, is there less of a drop in R^2 ?

Note: `rpart` automatically does internal CV, varying the complexity paramter (`cp`). If you use the `tree` package instead, you will have to use `cv.tree()`

CART Pruning

With `plotcp()` we are going to choose the error within 1 SE of the lowest cross-validated error. This will be used to prune

```
plotcp(rpart.train)
```



CART Pruning Continued

```
printcp(rpart.train)
```

```
##  
## Regression tree:  
## rpart(formula = medv ~ ., data = Boston.train)  
##  
## Variables actually used in tree construction:  
## [1] age   crim  lstat rm    tax  
##  
## Root node error: 24098/253 = 95.251  
##  
## n= 253  
##  
##      CP nsplit rel error  xerror    xstd  
## 1 0.491102      0  1.00000 1.01420 0.115293  
## 2 0.160787      1  0.50890 0.58289 0.066792  
## 3 0.072844      2  0.34811 0.40677 0.049227  
## 4 0.036758      3  0.27527 0.38152 0.049815  
## 5 0.029632      4  0.23851 0.35634 0.051564  
## 6 0.029057      5  0.20888 0.31094 0.046913  
## 7 0.013260      6  0.17982 0.28364 0.045076  
## 8 0.012100      7  0.16656 0.26523 0.041999  
## 9 0.010000      8  0.15446 0.26539 0.041833
```

Rpart Error Explained

rel error – The error on the whole sample (lower the better)

xerror – Error on the cross-validation folds

xstd – variability in error across CV folds

CART Pruning Continued

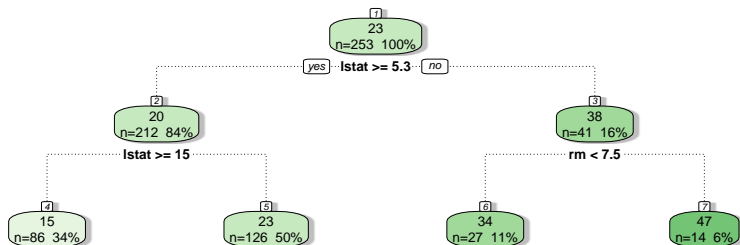
```
# rsq.rpart(rpart.train) ## another way to get cp plots
prune.Bos <- prune(rpart.train,0.053)

pred.prune <- predict(prune.Bos,Boston.test)
cor(pred.prune,Boston.test$medv)
```

```
## [1] 0.7734145
```

Plot Pruned Tree

```
#plot(prune.Bos);text(prune.Bos)
fancyRpartPlot(prune.Bos)
```



```
ctree.train <- ctree(medv ~., data=Boston.train)
#plot(ctree.train) # huge tree
pred.ctreeTest <- predict(ctree.train,Boston.test)
cor(pred.ctreeTest,Boston.test$medv)**2
```

```
## [1] 0.6470982
```

It is worth noting how much more of an effect there was for using a test dataset with the tree methods as compared to `lm()`, this is pretty typical, and much more important with more “flexible” methods such as random forests, `gbm`, `svm` etc. . .

Classification

Classification

Two Biggest Things To Remember:

1. Make sure functions outcome variable is categorical; `as.factor(outcome)`
2. Using `predict()` changes. Variable across packages

As a baseline, we will use logistic regression.

```
library(ISLR)
data(Default)
#head(Default)
str(Default)
```

```
## 'data.frame':    10000 obs. of  4 variables:
## $ default: Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
## $ student: Factor w/ 2 levels "No","Yes": 1 2 1 1 1 2 1 2 1 1 ...
## $ balance: num  730 817 1074 529 786 ...
## $ income : num  44362 12106 31767 35704 38463 ...
```

My favorite function in R is `str()`, as it gives the class of each variable and other summary characteristics. Most important thing to note is that the “default” variable is already coded as a factor variable, meaning that R now knows it is categorical, and will change the cost function (thus estimator) accordingly. This is really important because `rpart`, `randomForest` and other packages do not automatically detect whether it is a regression or classification problem. If you don't change the outcome variable to its proper class, you could get a suboptimal answer (use the wrong estimator i.e. regression instead of logistic regression)

Logistic Regression Now let's do logistic regression

ROC Curves

good intro to using ROC:

<https://ccrma.stanford.edu/workshops/mir2009/references/ROCintro.pdf>

These plots are a balance of sensitivity and specificity. Ideally the curve gets as close as possible to the upper left corner.

To get this plot, we need to get our predictions from our logistic model.

```
glm.probs=predict(lr.out,type="response")
#glm.pred00=ifelse(glm.probs>0.5,1,0)

rocCurve <- roc(Default$default,glm.probs)
pROC::auc(rocCurve)
```

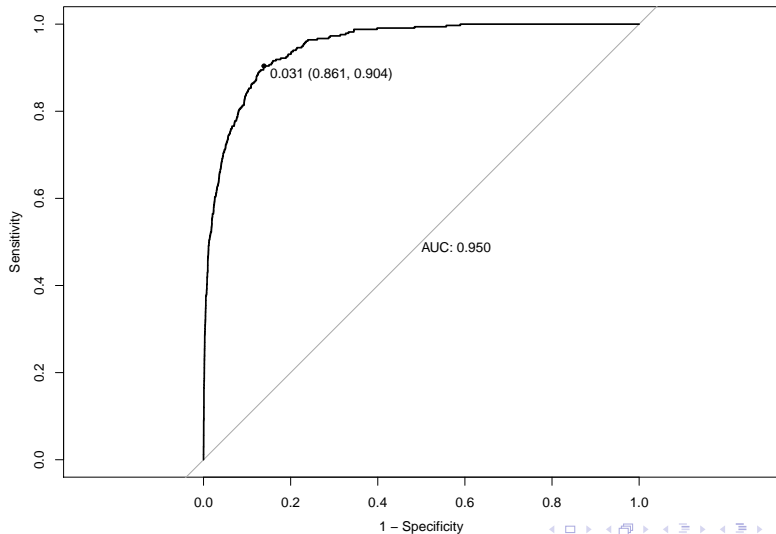
```
## Area under the curve: 0.9496
```

```
pROC::ci(rocCurve)
```

```
## 95% CI: 0.9402-0.959 (DeLong)
```


Plot ROC Curve

```
# quartz()  
plot(rocCurve, legacy.axes = TRUE, print.thres=T, print.auc=T)
```



Lasso Logistic Regression

How about lasso logistic regression?

```
library(glmnet)
yy = as.numeric(Default$default)
xx = apply(Default[,2:4],as.numeric)
lasso.out <- cv.glmnet(xx,yy,family="binomial",alpha=1,nfolds=10) #alpha=1 ==
# find best lambda
ll <- lasso.out$lambda.min

lasso.probs <- predict(lasso.out,newx=xx,s=ll,type="response")
```

Results from lasso using CV

```
rocCurve.lasso <- roc(Default$default,lasso.probs)
pROC::auc(rocCurve.lasso)
```

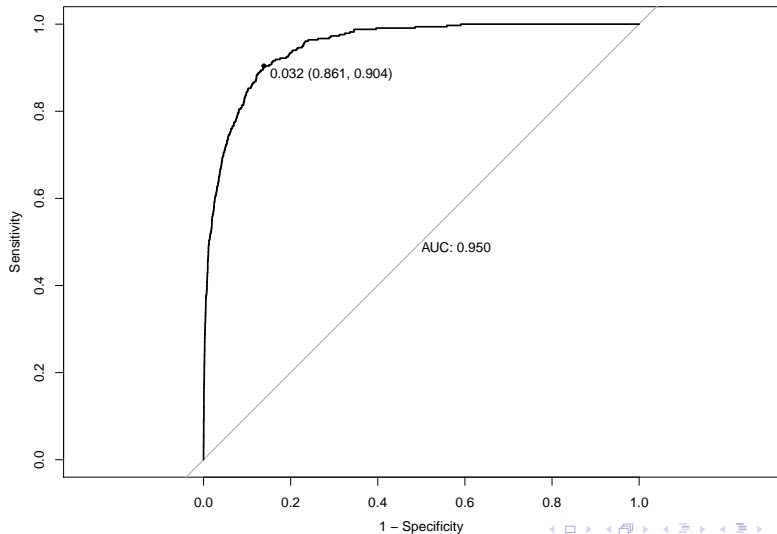
Area under the curve: 0.9495

```
pROC::ci(rocCurve.lasso)
```

95% CI: 0.9401-0.959 (DeLong)

Plot ROC

```
# quartz()  
plot(rocCurve.lasso, legacy.axes = TRUE, print.thres=T, print.auc=T)
```

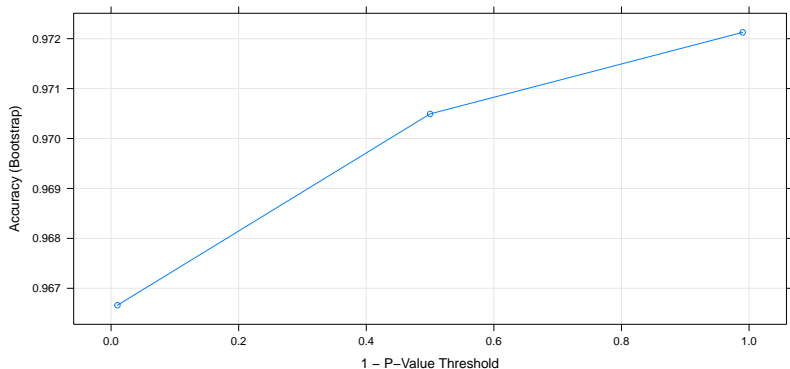


Using Decision Trees for Classification

Instead of demonstrating how to use `rpart()` or `ctree()`, I prefer to use the `train()` from `caret`. This makes it much easier to test out multiple different methods, as well as automatically vary the tuning parameters such as depth, complexity etc..

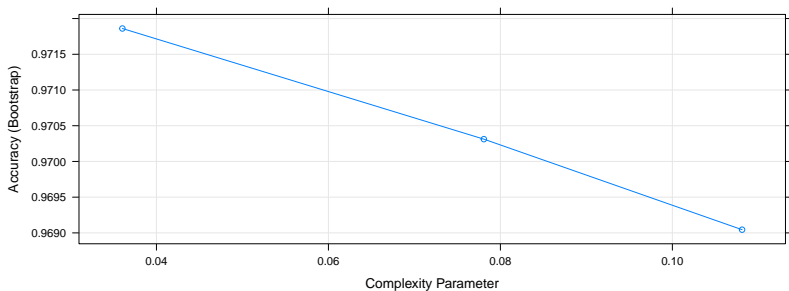
`train()` for `ctree`

```
train.ctree <- train(as.factor(default)~  
                    student+balance+income,data=Default,method="ctree")  
plot(train.ctree)
```



train() for rpart

```
train.rpart <- train(as.factor(default)~  
                     student+balance+income,data=Default,method="rpart")  
plot(train.rpart)
```



train() Continued

In `train()` and through `trainControl()` you can see that it automatically varies different tuning parameters (see caret documentation for the different options for each method), while defaulting to bootstrap estimation to test out each. This is a great way to prevent overfitting.

In examining both plots, it seems as both methods do comparably well, while also they both have different tuning parameters (X-axis). Based on these plots, I would increase the number of values for the tuning parameters, as the accuracy did not reach a maximum necessarily outside of the tails. (`tuneLength = 3` is default)

using a confusion matrix

```
train.class <- predict(train.rpart,Default,type="raw")
confusionMatrix(train.class,Default$default)
```

```
## Confusion Matrix and Statistics
```

```
##
```

```
##           Reference
```

```
## Prediction    No  Yes
```

```
##           No 9639 243
```

```
##           Yes   28   90
```

```
##
```

```
##                Accuracy : 0.9729
```

```
##                95% CI : (0.9695, 0.976)
```

```
##      No Information Rate : 0.9667
```

```
##      P-Value [Acc > NIR] : 0.0002082
```

```
##
```

```
##                Kappa : 0.3885
```

```
## Mcnemar's Test P-Value : < 2.2e-16
```

```
##
```

```
##                Sensitivity : 0.9971
```

```
##                Specificity : 0.2703
```

```
##      Pos Pred Value : 0.9754
```

```
##      Neg Pred Value : 0.7627
```

```
##      Prevalence : 0.9667
```

```
##      Detection Rate : 0.9639
```

```
##      Detection Prevalence : 0.9882
```

```
##      Balanced Accuracy : 0.6337
```

Confusion matrix terminology

Accuracy – $TP + TN / \text{total}$

Specificity – $TN / \text{Actual Negative}$

Sensitivity – $TP / \text{Actual Positive}$

https://en.wikipedia.org/wiki/Confusion_matrix

table()

The typical way of getting a table of classification results:

```
table(train.class,Default$default)
```

```
##  
## train.class    No   Yes  
##           No 9639  243  
##           Yes  28   90
```

I use `confusionMatrix()` because I am lazy and don't want to calculate all of the other statistics

Changing the cutoff for class assignment

This uses the optimal cutoff from the pROC plot

```
train.prob <- predict(train.rpart,Default,type="prob")[,2]
train.class2 <- ifelse(train.prob > .031,1,0)
confusionMatrix(as.numeric(Default$default)-1,train.class2,positive="1")
```

```
## Confusion Matrix and Statistics
```

```
##
```

```
##           Reference
```

```
## Prediction    0    1
```

```
##           0 9541 126
```

```
##           1  171 162
```

```
##
```

```
##           Accuracy : 0.9703
```

```
##           95% CI : (0.9668, 0.9735)
```

```
## No Information Rate : 0.9712
```

```
## P-Value [Acc > NIR] : 0.71715
```

```
##
```

```
##           Kappa : 0.5065
```

```
## Mcnemar's Test P-Value : 0.01068
```

```
##
```

```
##           Sensitivity : 0.5625
```

```
##           Specificity : 0.9824
```

```
## Pos Pred Value : 0.4865
```

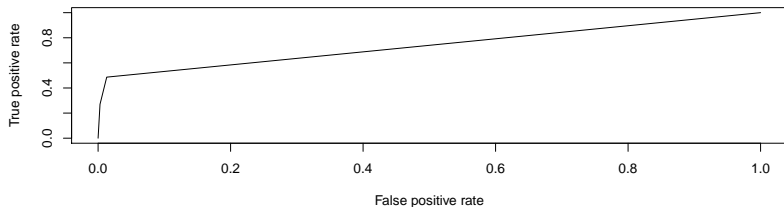
```
## Neg Pred Value : 0.9870
```

```
## Prevalence : 0.0288
```

```
## Detection Rate : 0.0162
```

Another Way to Get Optimal Threshold

```
library(ROCR)
pred <- prediction(train.prob,as.numeric(Default$default)-1)
perf <- performance(pred,"tpr","fpr")
#str(perf)
plot(perf)
```



```
cutoffs <- data.frame(cut=perf@alpha.values[[1]], fpr=perf@x.values[[1]],
                      tpr=perf@y.values[[1]])
```

Boosting and Random Forests

Boosting and RF Setup

For this, we will use the **caret** package as an interface to both the **gbm** and **randomForest** packages.

Because gbm and random forests take much longer to run, we could set up parallelization through caret and other packages.

<http://topepo.github.io/caret/parallel.html>

In my experience, unless your dataset is huge, parallelization with random forests tends to take longer than setting up only serial computation.

In caret, randomForest has two implementations, `method="rf"` and `method="parRF"` with `parRF` being the parallel version. The only tuning parameter for both is `mtry`.

Note, that using the `train()` will take longer, as it is using different tuning parameters and by default using bootstrap sampling to prevent overfitting.

To let `train()` pick the values of `mtry`, just set `tuneLength` to however many different values you want it to try. Default is 3.

Run Random Forest

```
#library(snowfall);#sfInit(parallel=T,cpus=4)
cont <- trainControl(allowParallel=TRUE,method="cv")
```

```
train.rf1 <- train(medv ~ ., data=Boston.train,method="rf",
                  trControl=cont,
                  importance=T,tuneLength=3)
```

```
train.rf1
```

```
## Random Forest
```

```
##
```

```
## 253 samples
```

```
## 13 predictor
```

```
##
```

```
## No pre-processing
```

```
## Resampling: Cross-Validated (10 fold)
```

```
## Summary of sample sizes: 228, 228, 227, 228, 229, 227, ...
```

```
## Resampling results across tuning parameters:
```

```
##
```

```
##      mtry  RMSE      Rsquared
```

```
##      2    4.059928  0.8485182
```

```
##      7    3.782779  0.8589950
```

```
##     13    3.902765  0.8463466
```

```
##
```

```
## RMSE was used to select the optimal model using the smallest value.
```

```
## The final value used for the model was mtry = 7
```

Performance on a holdout

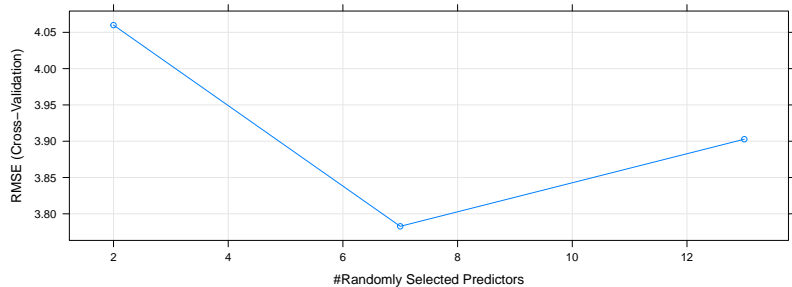
```
# see how we do on hold out sample  
# automatically chooses best model for prediction  
pred.test1 <- predict(train.rf1,Boston.test)  
cor(pred.test1,Boston.test$medv)**2
```

```
## [1] 0.8818734
```

Best practice is to only do this with one model. I.e. choose between best random forest and boosting models, take this one model and check performance on test dataset and report.

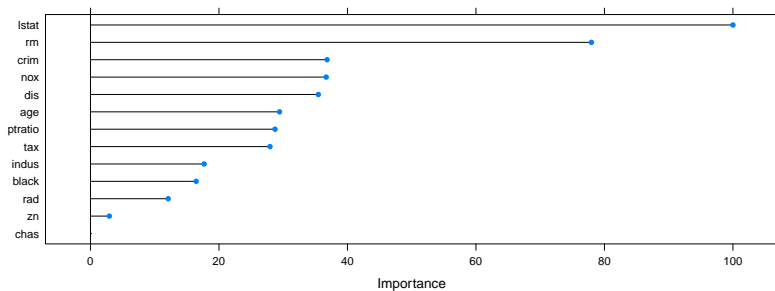
Plot Performance

```
plot(train.rf1)
```



Variable Importance

```
#rf <- train.rf1$finalModel  
imp <- varImp(train.rf1)  
plot(imp)
```



cforest is implemented as method="cforest" with the only tuning parameter being mtry

```
train.cf1 <- train(medv ~., data=Boston.train,method="cforest")
#train.cf1

#plot(train.cf1)

#varImp(train.cf1)

# see how we do on hold out sample
pred.test2 <- predict(train.cf1,Boston.test)
cor(pred.test2,Boston.test$medv)**2
```

```
## [1] 0.7976894
```

Classification

For classification, a couple other things to use paired with train()

```
# set up data
data(Carseats)
#attach(Carseats)
#hist(Carseats$Sales)
High=ifelse(Carseats$Sales<=8,"No","Yes")
Carseats=data.frame(Carseats, High)
Carseats$Sales <- NULL
Carseats$ShelveLoc <- as.numeric(Carseats$ShelveLoc)

train2 = sample(dim(Carseats)[1], dim(Carseats)[1]/2) # half of sample
Carseats.train = Carseats[train2, ]
Carseats.test = Carseats[-train2, ]
```

Now run random forests

```
train.rf2 <- train(as.factor(High) ~ .,  
                  data=Carseats.train,method="rf",trControl=cont)  
#train.rf2  
rf.probs=predict(train.rf2,newdata=Carseats.test,type="prob")[,2]  
rocCurve22 <- roc(Carseats.test$High,rf.probs)  
#auc(rocCurve22);#  
rf.class=predict(train.rf2,newdata=Carseats.train)  
confusionMatrix(Carseats.train$High,rf.class)
```

Confusion Matrix and Statistics

##

Reference

Prediction No Yes

No 115 0

Yes 0 85

##

Accuracy : 1

95% CI : (0.9817, 1)

No Information Rate : 0.575

P-Value [Acc > NIR] : < 2.2e-16

##

Kappa : 1

McNemar's Test P-Value : NA

##

Sensitivity : 1.000

Specificity : 1.000

Confusion Matrix on Test Data

```
#auc(rocCurve4);#ci(rocCurve4);#plot(rocCurve4)
rf.classTest=predict(train.rf2,Carseats.test)
confusionMatrix(Carseats.test$High,rf.classTest)
```

```
## Confusion Matrix and Statistics
```

```
##
```

```
##           Reference
```

```
## Prediction No Yes
```

```
##           No  98  23
```

```
##           Yes 28  51
```

```
##
```

```
##           Accuracy : 0.745
```

```
##           95% CI : (0.6787, 0.8039)
```

```
##           No Information Rate : 0.63
```

```
##           P-Value [Acc > NIR] : 0.0003656
```

```
##
```

```
##           Kappa : 0.4605
```

```
##           Mcnemar's Test P-Value : 0.5754030
```

```
##
```

```
##           Sensitivity : 0.7778
```

```
##           Specificity : 0.6892
```

```
##           Pos Pred Value : 0.8099
```

```
##           Neg Pred Value : 0.6456
```

```
##           Prevalence : 0.6300
```

```
##           Detection Rate : 0.4900
```

```
##           Detection Prevalence : 0.6050
```

cforest for binary

automatically knows it is binary, unlike randomForest

```
train.cf2 <- train(High ~ .,  
                  data=Carseats.train,method="cforest",trControl=cont)  
#train.cf2  
cf.probs=predict(train.cf2)  
rocCurve33 <- roc(Carseats.train$High,as.numeric(cf.probs));#auc(rocCurve33)  
#plot(rocCurve33,add=T,col=3)  
cf.class=predict(train.cf2$finalModel)  
confusionMatrix(Carseats.train$High,cf.class) # no errors
```

```
## Confusion Matrix and Statistics
```

```
##
```

```
##           Reference
```

```
## Prediction  No  Yes
```

```
##           No  110   5
```

```
##           Yes   8  77
```

```
##
```

```
##           Accuracy : 0.935
```

```
##           95% CI : (0.8914, 0.9649)
```

```
##           No Information Rate : 0.59
```

```
##           P-Value [Acc > NIR] : <2e-16
```

```
##
```

```
##           Kappa : 0.8664
```

```
##           Mcnemar's Test P-Value : 0.5791
```

CForest on Test Data

```
cf.classTest=predict(train.cf2,newdata=Carseats.test)
cf.probs2=predict(train.cf2,newdata=Carseats.test,type="prob")[,2]
rocCurve333 <- roc(Carseats.test$High,as.numeric(cf.probs2));#auc(rocCurve333)
#plot(rocCurve333,add=T,col=3)
```

CForest Test Confusion Matrix

```
confusionMatrix(Carseats.test$High,cf.classTest)
```

```
## Confusion Matrix and Statistics
```

```
##
```

```
##           Reference
```

```
## Prediction No Yes
```

```
##           No  96  25
```

```
##           Yes  28  51
```

```
##
```

```
##           Accuracy : 0.735
```

```
##           95% CI : (0.6681, 0.7948)
```

```
##           No Information Rate : 0.62
```

```
##           P-Value [Acc > NIR] : 0.0003995
```

```
##
```

```
##           Kappa : 0.4419
```

```
## Mcnemar's Test P-Value : 0.7835305
```

```
##
```

```
##           Sensitivity : 0.7742
```

```
##           Specificity : 0.6711
```

```
##           Pos Pred Value : 0.7934
```

```
##           Neg Pred Value : 0.6456
```

```
##           Prevalence : 0.6200
```

```
##           Detection Rate : 0.4800
```

```
##           Detection Prevalence : 0.6050
```

```
##           Balanced Accuracy : 0.7226
```

```
##
```


Boosting

packages: “gbm” and “ada” – both can be accessed through caret

Example tuning parameters for “gbm”:

<http://topepo.github.io/caret/training.html>

For this, I am going to use method=“ada” which is one of the forms of boosting.

Currently, problem with method=“gbm”

Basic set up to compare results to RF:

```
#modelLookup("ada")
train.gbm <- train(as.factor(High) ~ ., verbose=F,
                   data=Carseats.train,method="gbm",trControl=cont)
#train.gbm

gbm.probs=predict(train.gbm,newdata=Carseats.test,type="prob")[,2]
rocCurve3 <- roc(Carseats.test$High,gbm.probs)
#auc(rocCurve3)
#ci(rocCurve3)
```

Compare ROC Curves

```
plot(rocCurve22,col=c(1)) # color black is rf  
plot(rocCurve333,add=T,col=c(2)) # color red is cforest  
plot(rocCurve3,add=T,col=c(3)) # color green is gbm
```

