Decision Trees

This script will go over Decision Trees in R. It will not cover generalization such as SEM Trees or Rasch Trees

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

For longitudinal data:
REEMtree

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
(see "train_model_list" in caret reference manual)

Bonus:
non-greedy tree algorithm:
evtree: http://cran.r-project.org/web/packages/evtree/vignettes/evtree.pdf

Lets load the main packages

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

Regression (continous outcome)

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

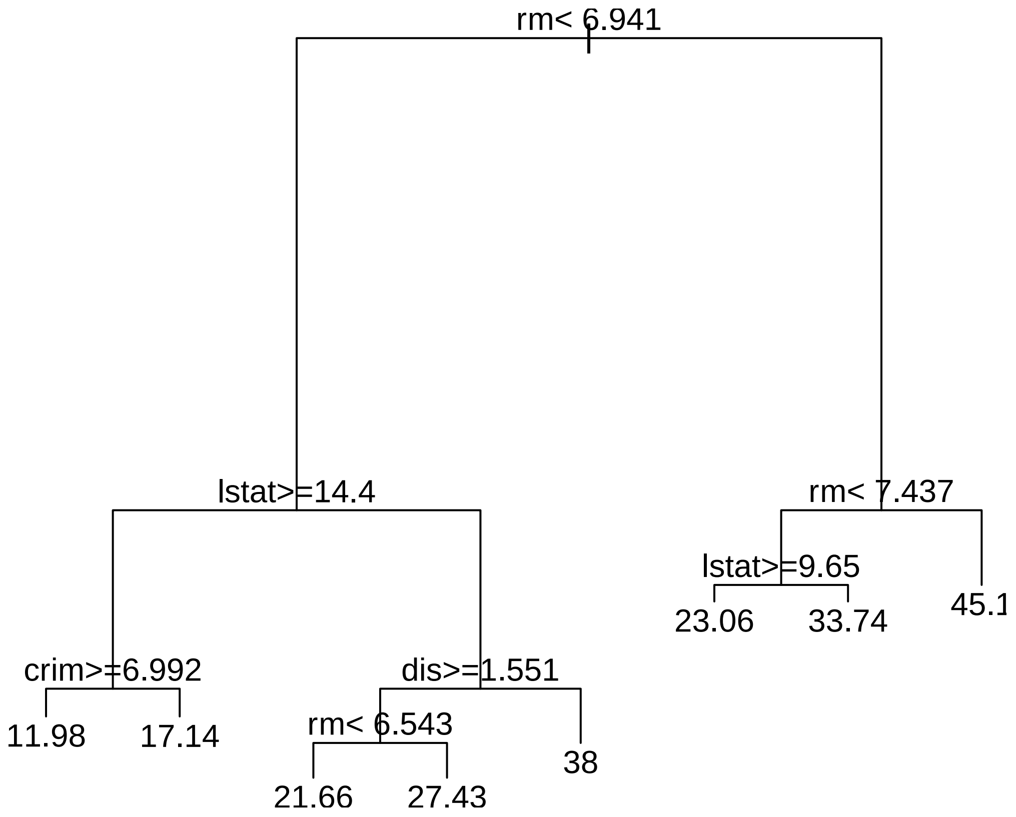
```r
#str(Boston)

# 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)
```



```
# 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
##       12) lstat>=9.65 7   432.9971 23.05714 *
##       13) lstat< 9.65 39   789.5123 33.73846 *
##      7) rm>=7.437 30  1098.8500 45.09667 *
```

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

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

Doing really well – Rsquared = 0.81
## 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”

```r
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,] 0.7402516
```

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

So the plot for rpart didn'tcome 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
Let's load some new packages:

```r
library(rattle)
```

```
## Rattle: A free graphical interface for data mining with R.
## Version 3.3.0 Copyright (c) 2006-2014 Togaware Pty Ltd.
## Type 'rattle()' to shake, rattle, and roll your data.
```

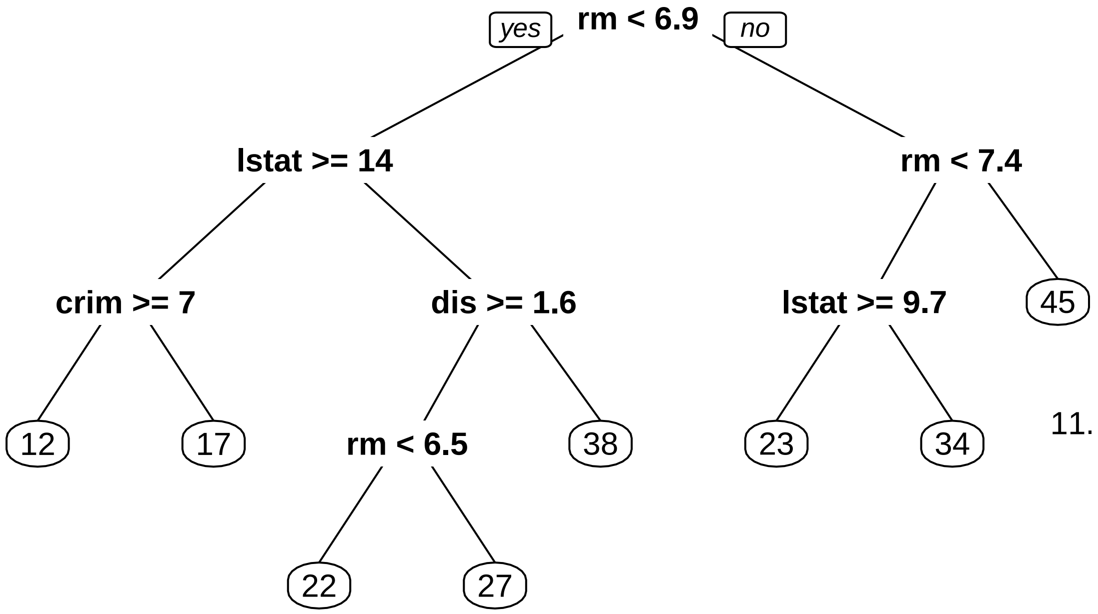
```r
library(rpart.plot)
library(RColorBrewer)
library(partykit)
```

```
##
## Attaching package: 'partykit'
##
## The following objects are masked from 'package:party':
##
##     ctree, ctree_control, edge_simple, mob, mob_control,
##     node_barplot, node_bivplot, node_boxplot, node_inner,
##     node_surv, node_terminal
```

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

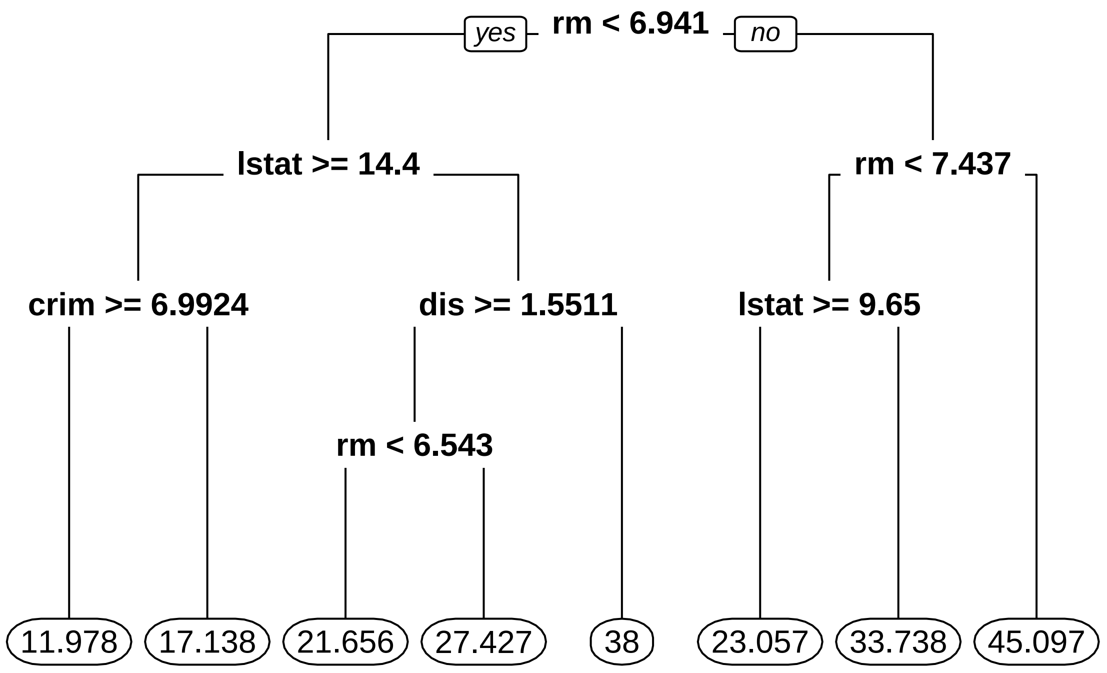
Anyways, lets try some new, prettier plots:

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

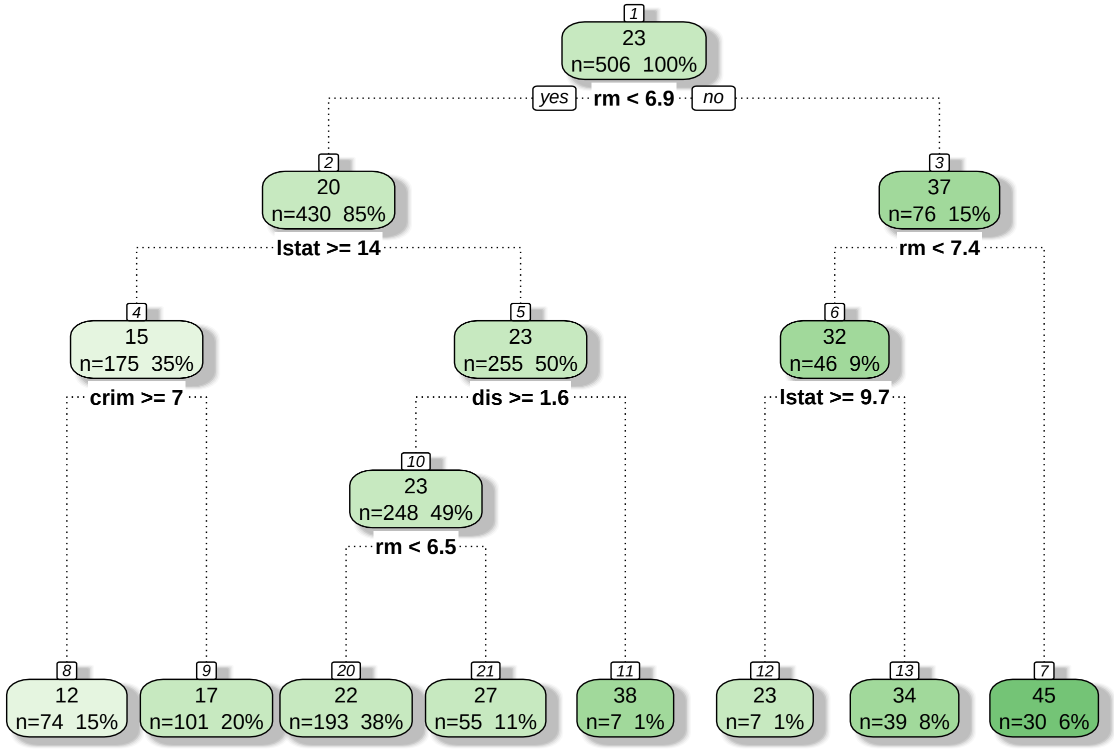


Note, prp() offers many additional capabilities for tweaking the plot For instance:

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



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



Rattle 2015−Jun−09 09:18:55 RJacobucci

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

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, ]
```

Try linear regression first

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

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

```
## [1] 0.7037515
```

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 Rsq are 0.74

How about with rpart?

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

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

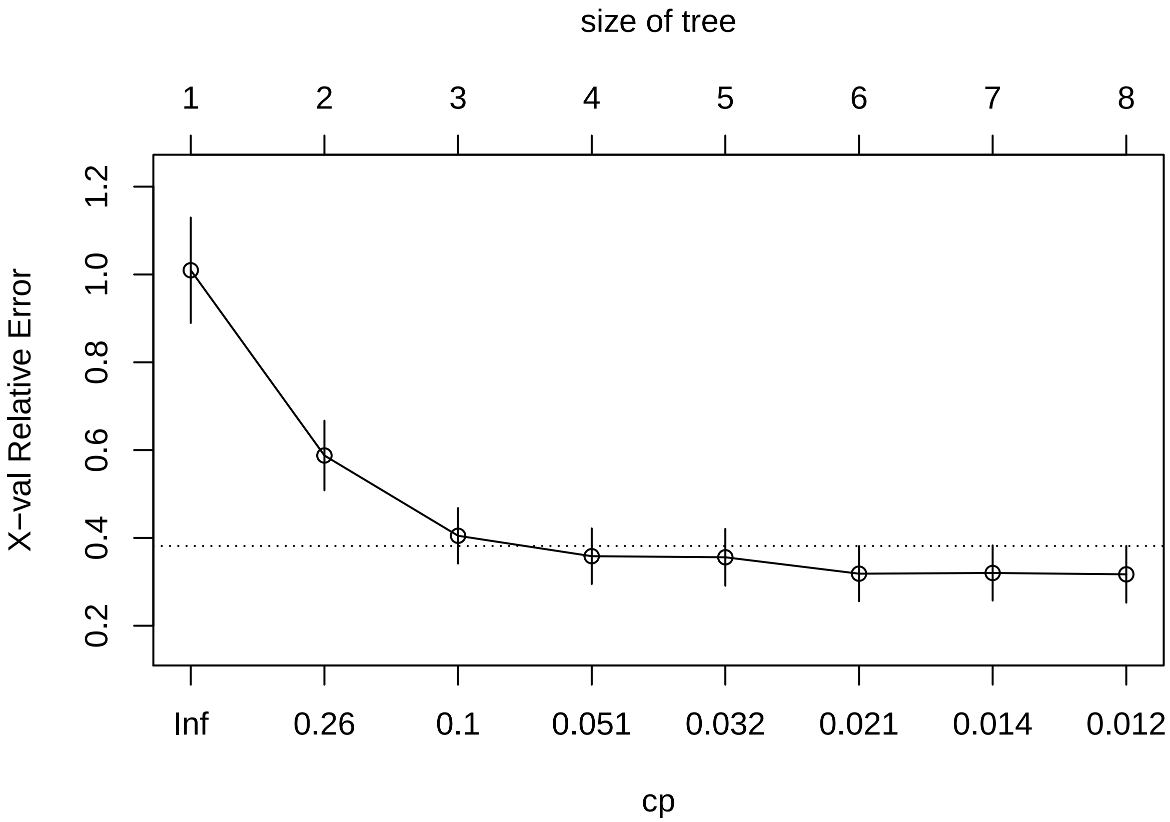
```
## [1] 0.7592042
```

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()

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)
```



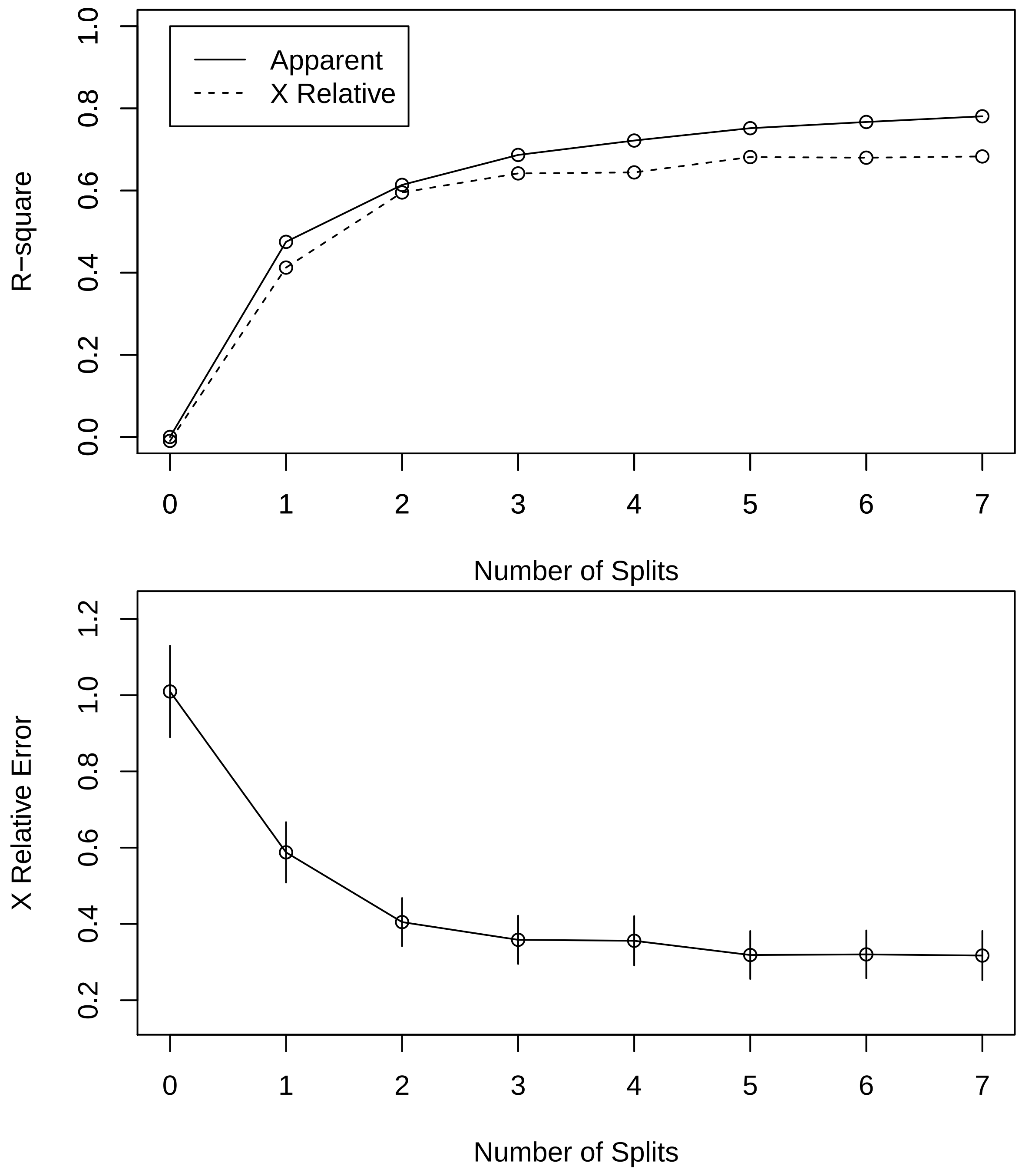
```
printcp(rpart.train)
```

```
##
## Regression tree:
## rpart(formula = medv ~ ., data = Boston.train)
##
## Variables actually used in tree construction:
## [1] crim  indus lstat nox   rm
```

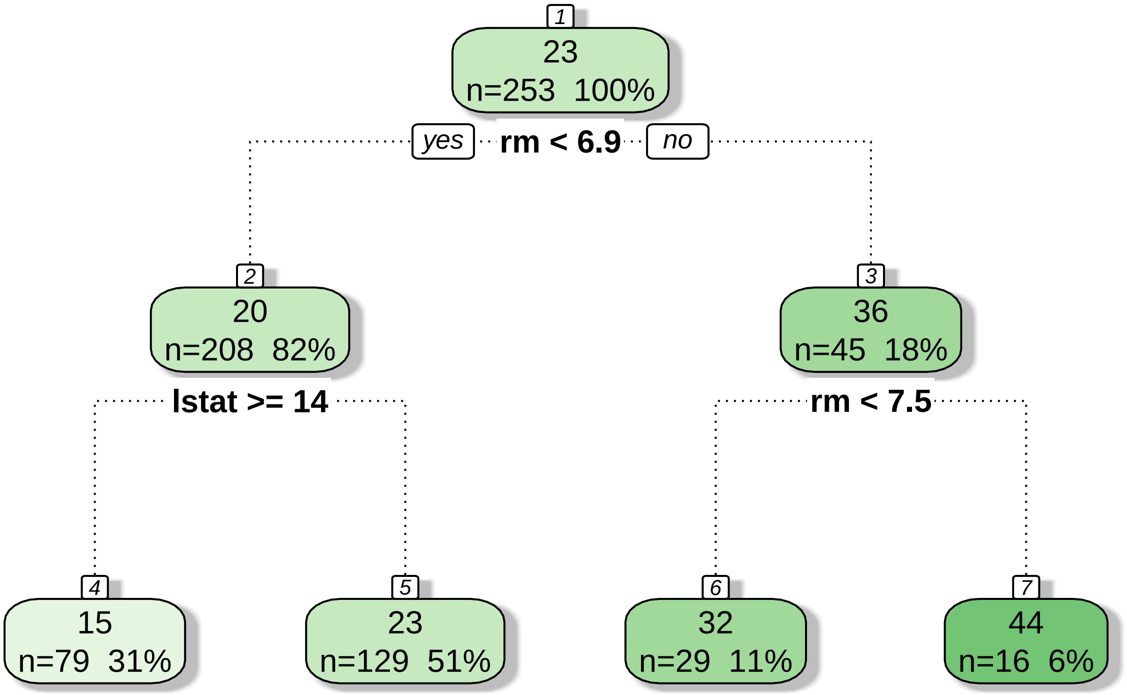
```
##
## Root node error: 20372/253 = 80.523
##
## n= 253
##
##          CP nsplit rel error  xerror     xstd
## 1 0.475006      0   1.00000 1.00958 0.120011
## 2 0.138677      1   0.52499 0.58770 0.079222
## 3 0.072917      2   0.38632 0.40481 0.063126
## 4 0.035061      3   0.31340 0.35835 0.063335
## 5 0.030042      4   0.27834 0.35590 0.064802
## 6 0.015167      5   0.24830 0.31855 0.062875
## 7 0.013772      6   0.23313 0.32013 0.062853
## 8 0.010000      7   0.21936 0.31709 0.064527
```

```r
rsq.rpart(rpart.train)
```

```
##
## Regression tree:
## rpart(formula = medv ~ ., data = Boston.train)
##
## Variables actually used in tree construction:
## [1] crim  indus lstat nox   rm
##
## Root node error: 20372/253 = 80.523
##
## n= 253
##
##          CP nsplit rel error  xerror     xstd
## 1 0.475006      0   1.00000 1.00958 0.120011
## 2 0.138677      1   0.52499 0.58770 0.079222
## 3 0.072917      2   0.38632 0.40481 0.063126
## 4 0.035061      3   0.31340 0.35835 0.063335
## 5 0.030042      4   0.27834 0.35590 0.064802
## 6 0.015167      5   0.24830 0.31855 0.062875
## 7 0.013772      6   0.23313 0.32013 0.062853
## 8 0.010000      7   0.21936 0.31709 0.064527
```



```r
prune.Bos <- prune(rpart.train,0.053)

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



Rattle 2015−Jun−09 09:18:57 RJacobucci

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

```
## [1] 0.8298096
```

caret:

```
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] 0.7791081
```

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 (categorical outcome)

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.

```r
library(ISLR)
data(Default)
head(Default)
```

```
##   default student   balance    income
## 1      No      No  729.5265 44361.625
## 2      No     Yes  817.1804 12106.135
## 3      No      No 1073.5492 31767.139
## 4      No      No  529.2506 35704.494
## 5      No      No  785.6559 38463.496
## 6      No     Yes  919.5885  7491.559
```

```r
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)
Now let's do logistic regression

```r
lr.out <- glm(default~student+balance+income,family="binomial",data=Default)
summary(lr.out)
```

```
##
## Call:
## glm(formula = default ~ student + balance + income, family = "binomial",
##     data = Default)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4691  -0.1418  -0.0557  -0.0203   3.7383
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.087e+01  4.923e-01 -22.080  < 2e-16 ***
## studentYes  -6.468e-01  2.363e-01  -2.738  0.00619 **
## balance      5.737e-03  2.319e-04  24.738  < 2e-16 ***
## income       3.033e-06  8.203e-06   0.370  0.71152
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
```

```
##
##     Null deviance: 2920.6  on 9999  degrees of freedom
## Residual deviance: 1571.5  on 9996  degrees of freedom
## AIC: 1579.5
##
## Number of Fisher Scoring iterations: 8
```

I always find it much harder to figure out how well I am doing with logistic regression. One of the best ways to assess results in my opinion is the use of receiver operating characteristic curves (ROC curves).

good intro to using ROC:
https://ccrma.stanford.edu/workshops/mir2009/references/ROCintro.pdf
These plots are a balanace 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.

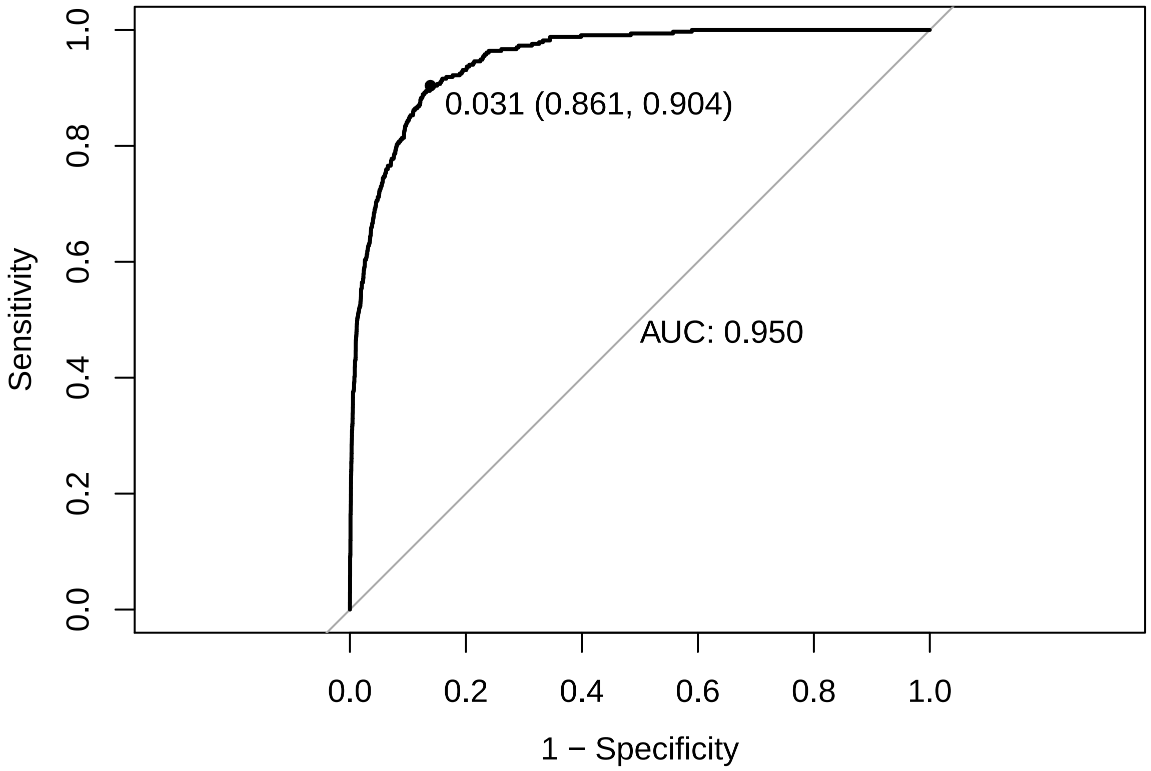
```
glm.probs=predict(lr.out,type="response")
#glm.pred00=ifelse(glm.probs>0.5,1,0)

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)
```

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



```
##
## Call:
## roc.default(response = Default$default, predictor = glm.probs)
##
## Data: glm.probs in 9667 controls (Default$default No) < 333 cases (Default$default Yes).
## Area under the curve: 0.9496
```

For AUC (area under the curve), values of 0.8 and 0.9 are good (the higher the better)

predict() with missing variables

How about lasso logistic regression?

```
library(glmnet)
yy = as.numeric(Default$default)
xx = sapply(Default[,2:4],as.numeric)
lasso.out <- cv.glmnet(xx,yy,family="binomial",alpha=1,nfolds=10) #alpha=1 ==  lasso; 0 =
# find best lambda
ll <- lasso.out$lambda.min

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

Results from lasso using CV

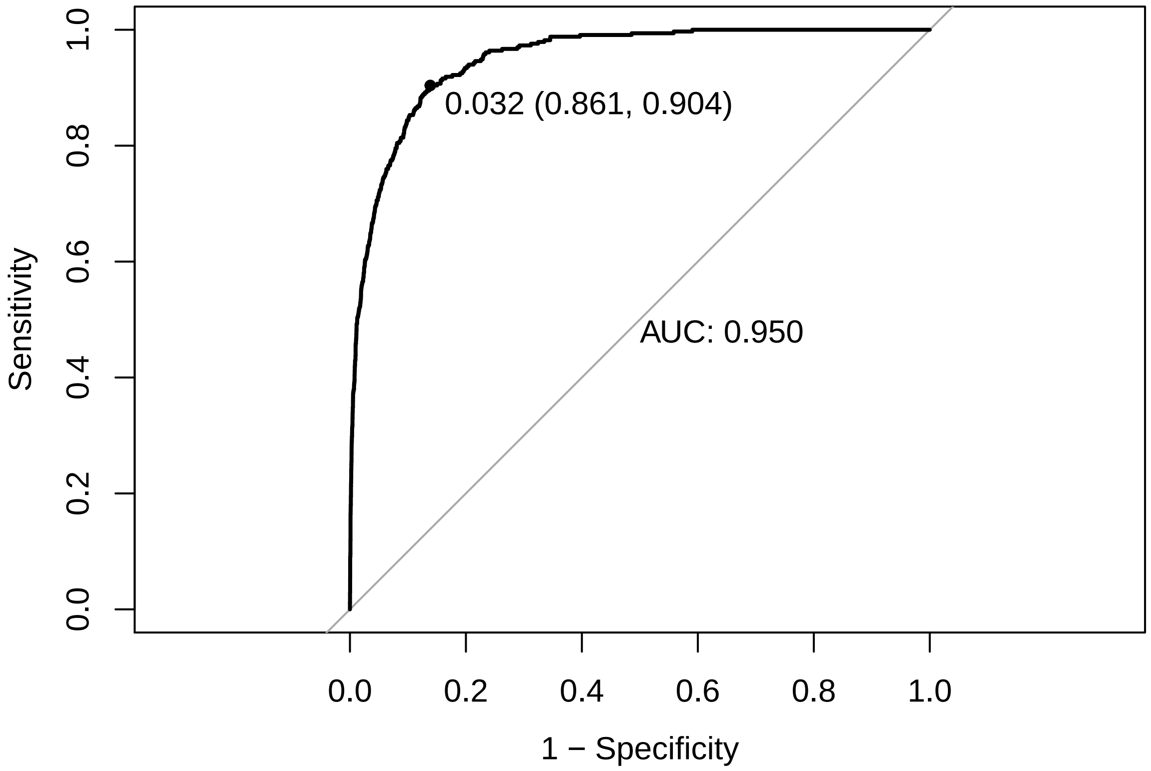
```
rocCurve.lasso <- roc(Default$default,lasso.probs)
pROC::auc(rocCurve.lasso)
```

```
## Area under the curve: 0.9495
```

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

```
## 95% CI: 0.9401-0.959 (DeLong)
```

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



```
##
## Call:
## roc.default(response = Default$default, predictor = lasso.probs)
##
## Data: lasso.probs in 9667 controls (Default$default No) < 333 cases (Default$default Yes).
## Area under the curve: 0.9495
```

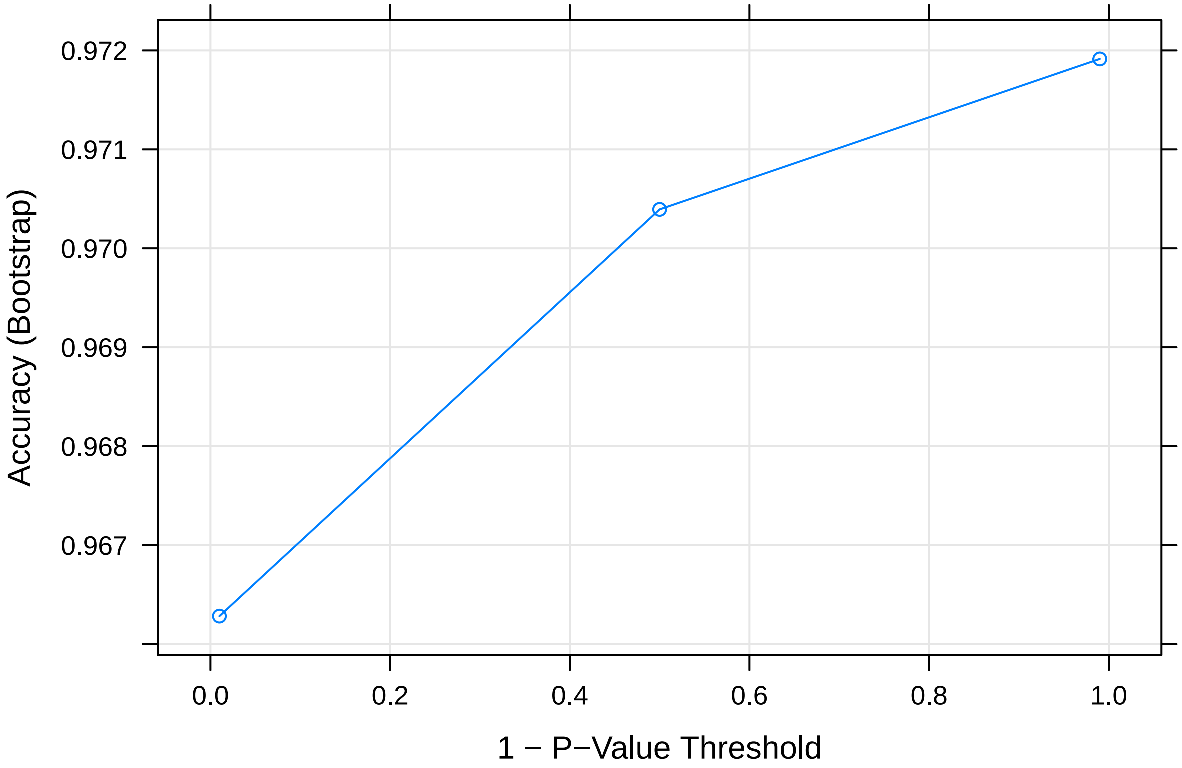
Almost identical results to logistic regression with no penalization.

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..

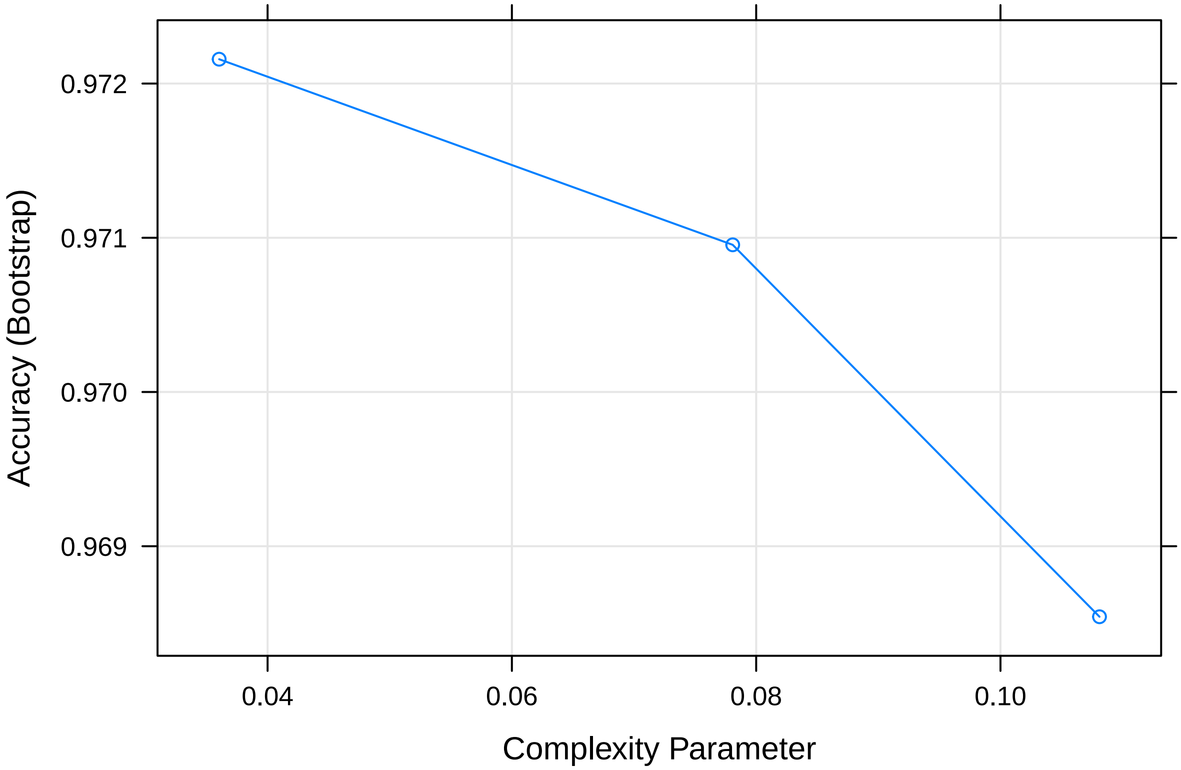
train() for ctree

```
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)
```



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)

Boosting and Random Forests

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.

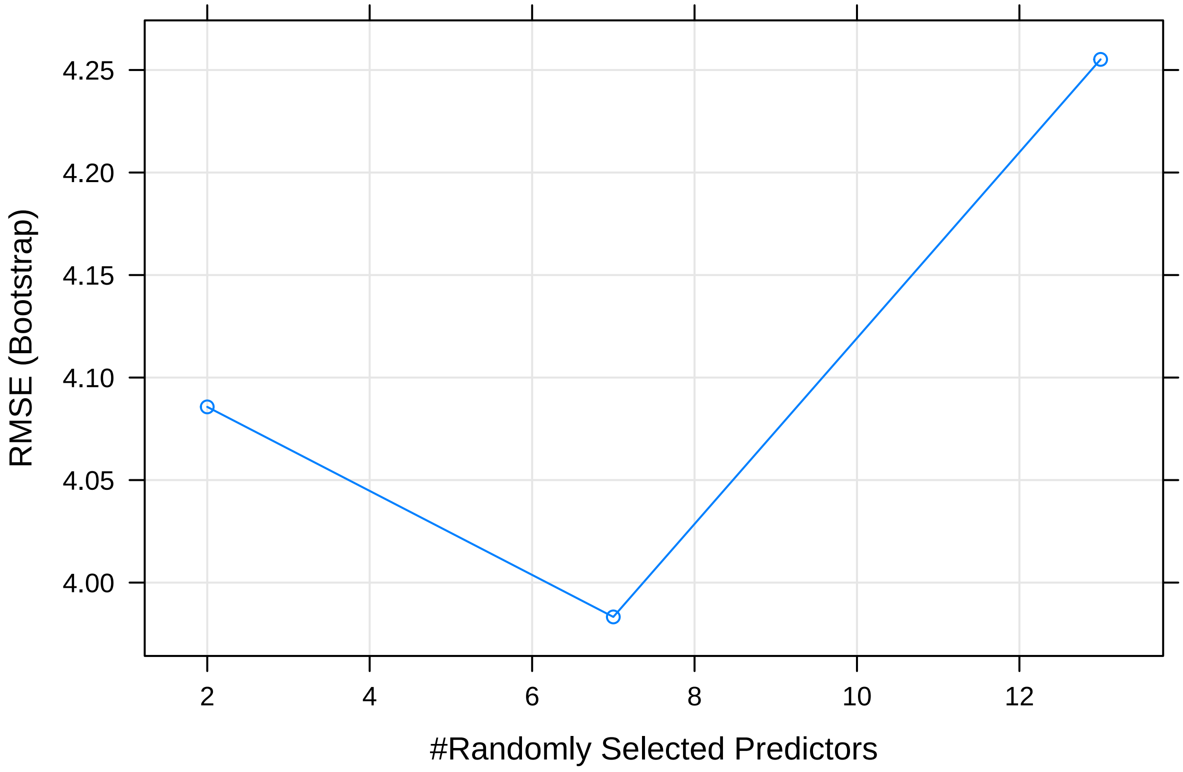
To let train() pick the values of mtry, just set tuneLength to however many different values you want it to try. Default is 3.

```
# set up parallel
#library(snowfall)
#sfInit(parallel=T,cpus=4)
```

```
cont <- trainControl(allowParallel=TRUE,method="cv")

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

train.rf1
```

```
## Random Forest
##
## 253 samples
##  13 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
##
## Summary of sample sizes: 253, 253, 253, 253, 253, 253, ...
##
## Resampling results across tuning parameters:
##
##   mtry  RMSE      Rsquared   RMSE SD    Rsquared SD
##    2    4.085702  0.7956939  0.6405938  0.06275338
##    7    3.983268  0.7878477  0.7637024  0.08231750
##   13    4.255169  0.7572780  0.8823610  0.09609684
##
## RMSE was used to select the optimal model using  the smallest value.
## The final value used for the model was mtry = 7.
```

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



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

```
## rf variable importance
##
##         Overall
## rm      100.000
## lstat    63.385
## nox      35.523
## crim     31.018
## ptratio  29.844
## dis      29.451
## age      25.831
## indus    22.760
## tax      20.542
## black     9.534
## rad       9.441
## zn        7.769
## chas      0.000
```

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

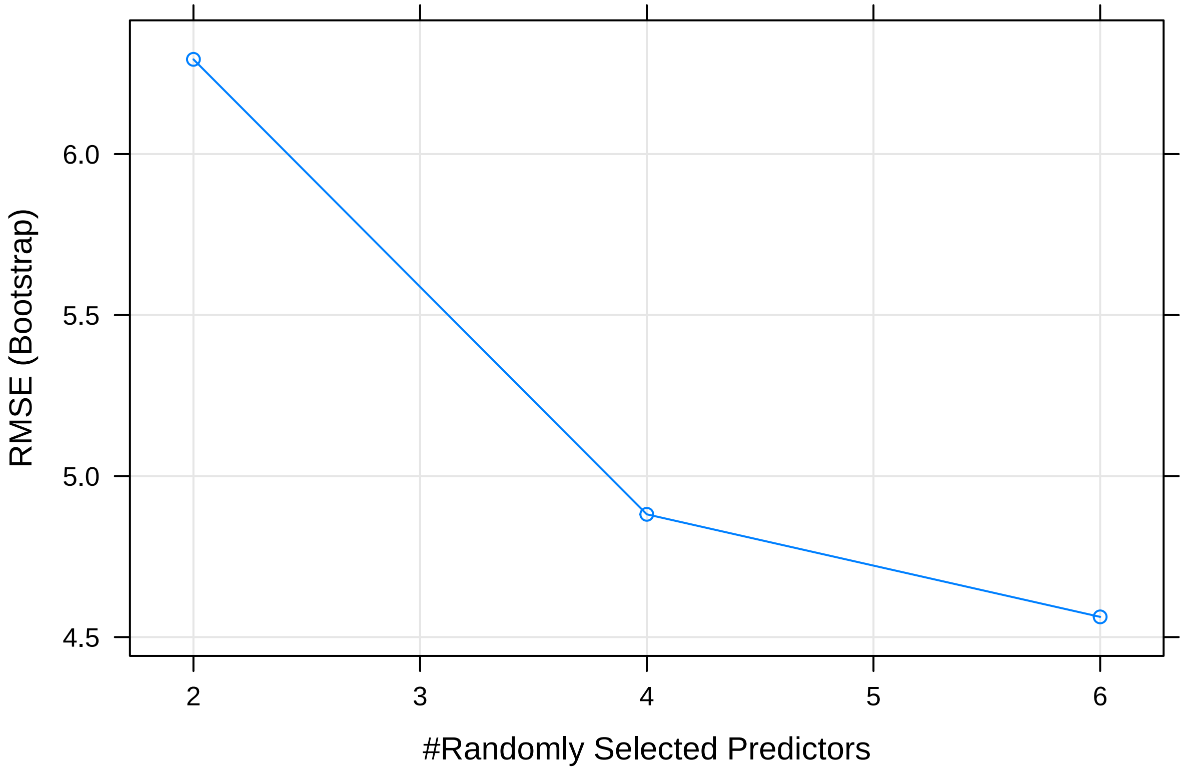
```
## [1] 0.906275
```

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

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

```
## Conditional Inference Random Forest
##
## 253 samples
##  13 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
##
## Summary of sample sizes: 253, 253, 253, 253, 253, 253, ...
##
## Resampling results across tuning parameters:
##
##   mtry  RMSE      Rsquared   RMSE SD    Rsquared SD
##   2     6.294135  0.6491496  0.7105531  0.07299538
##   4     4.881232  0.7531934  0.6671379  0.05998814
##   6     4.562823  0.7619192  0.6519958  0.06035083
##
## RMSE was used to select the optimal model using  the smallest value.
## The final value used for the model was mtry = 6.
```

```r
plot(train.cf1)
```



```r
varImp(train.cf1)
```

```
## cforest variable importance
##
##           Overall
## rm      100.00000
## lstat    79.06662
## crim      7.36433
## ptratio   7.29624
## indus     5.69712
## nox       4.67995
## tax       4.03395
## age       2.18994
## dis       1.37705
## rad       0.61917
## black     0.39727
## chas      0.02918
## zn        0.00000
```

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

```
## [1] 0.8531805
```

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.train$High,rf.probs)
auc(rocCurve22)
plot(rocCurve22)


rf.class=predict(train.rf2,newdata=Carseats.train)
confusionMatrix(Carseats.train$High,rf.class)

# how about on test dataset

rf.classTest=predict(train.rf2,Carseats.test)
confusionMatrix(Carseats.test$High,rf.classTest)


# ROC
rf.probs1=predict(train.rf2,Carseats.test,type="prob")[,2]
rocCurve4 <- roc(Carseats.test$High,rf.probs1)
auc(rocCurve4)
ci(rocCurve4)
plot(rocCurve4)
```

cforest for binary – automatically knows it is binary, unlike randomForest

!! bug with the predict.cforest function

```
train.cf2 <- train(High ~ ., data=Carseats.train,method="cforest",trControl=cont)
train.cf2

cf.probs=predict(train.cf2) # doesn't work. ????
rocCurve33 <- roc(Carseats$High,cf.probs)
auc(rocCurve33)
plot(rocCurve33)
```

```
cf.class=predict(train.cf2$finalModel)
confusionMatrix(Carseats$High,cf.class) # no errors


# how about on test dataset

cf.classTest=predict(train.cf2,newdata=Carseats.test)
confusionMatrix(Carseats.test$High,cf.classTest)
```

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.ada <- train(as.factor(High) ~ ., data=Carseats.train,method="ada",trControl=cont)
train.ada

ada.probs=predict(train.ada,newdata=Carseats.test,type="prob")[,2]
rocCurve3 <- roc(Carseats.test$High,ada.probs)
auc(rocCurve3)
ci(rocCurve3)
plot(rocCurve3,add=T,col=c(2)) # color red is ada
```