SOC 382 Machine Learning Lab

Austin van Loon

March 7, 2019

#Housekeeping

Here are the commands to download all of the packages we'll be using for today's lab.

```
#install.packages("tidyverse")
#install.packages("randomForest")
#install.packages("rpart")
#install.packages("rpart.plot")
#install.packages("glmnet")
#install.packages("partykit")
#install.packages("knitr")
#install.packages("ggplot2")
#install.packages("tufte")
```

Here we're going to load each of the libraries[1], empty our environment, and set the random seed so we can reproduce our results.

[1] the "tufte" library is just so we can use this theme

```
library(tidyverse)
library(randomForest)
library(rpart)
library(rpart.plot)
library(glmnet)
library(partykit)
library(knitr)
library(tidyr)
library(ggplot2)
library(tufte)

remove(list=ls())

set.seed(382)
```

#Data and train/test/dev splitting

Now, below we're going to load in a data set from the Stanford CS 109 course page. This data set contains information on some of the individuals who were aboard the Titanic. It contains some covariate information, the names of which are fairly self-explanatory, and whether they survived. Our goal here is to predict whether each individual survived or not[2]. As an evaluation metric, we'll use the R^2, since we have a pretty good understanding of what that is[3].

[2] Importantly, we're not trying to explain why these individuals survived

[3] In classification tasks such as this, it's normal to use "accuracy", "precision", or the "F1 score"

We're going to eliminate the "Name" variable, though you could imagine extracting useful information from this[4]. We're then going to change the "Survived" variable to be a factor, which is how R best understands dichotomous variables. Lastly, we'll standardize "Age" and "Fare".

[4] Such as whether "Capt." appears in the name

```
titanic <- read.csv('http://web.stanford.edu/class/archive/cs/cs109/cs109.1166/stuff/titanic.csv')
titanic$Name <- NULL
titanic$Survived <- as.factor(titanic$Survived)
titanic$Age <- (titanic$Age - mean(titanic$Age))/sd(titanic$Age)
titanic$Fare <- (titanic$Fare - mean(titanic$Fare))/sd(titanic$Fare)
```

Now we're going to split the data into a training set, a development (dev) set, and a test set. We'll do this by randomly generating numbers between zero and one for every observation, and specifying what proportion of observations (approximately) should be in the train and dev set (with the rest going to the test set)[5]. For our lab today, we'll be putting 60% of our data into our training set and 20% each into our dev set and test set. Remember that we could also do k-fold validation on a larger training set, but having a dev set will allow us to easily examine how much a model is "over-fitting".

[5] There are other ways to do this of course, and a quick Google search will probably reveal several

```
rand = runif(nrow(titanic))

ratio.train = .60
ratio.dev = .20

train <- titanic %>% subset(rand<=ratio.train)
dev <- titanic %>% subset(rand>ratio.train & rand<=(ratio.train+ratio.dev))
test <- titanic %>% subset(rand>(ratio.train + ratio.dev))
```

Now, for some commands in R it will prove useful to have a separate data frame for the covariates associated with a set. To figure our what any specific command is looking for, look at the documentation by typing "?" followed by the command in your console. For instance, to see the documentation for the "glm" command, type "?glm" in your console. For computing how well we're performing, it will be useful to keep a separate vector of the outcome variables, so we'll also store this separately.

```
x.train <- train
x.train$Survived <- NULL
x.train$Sex <- sapply(x.train$Sex, as.factor)
y.train <-train$Survived

x.dev <- dev
```

```
x.dev$Survived <- NULL
x.dev$Sex <- sapply(x.dev$Sex, as.factor)
y.dev <-dev$Survived
```

#Regularized regression

The first class of algorithms we'll examine fall under the umbrella of "regularized regression". These algorithms (to be less than exact) fit an OLS which simultaneously tries to minimize the size of the coefficients. This will make our predictions more generalizeable, but will also bias our specific beta estimates. That means that coefficient values are **NOT ACCURATE**... On purpose! One outcome of this method is that when two variables are highly correlated, one of them tends to be dropped, and any effect of that dropped variable will get "soaked up" by the remaining variable[6]. For comparison, let's first run a logistic regression. We'll print out a summary and then evaluate how well we predict the outcomes in the dev set by predicting those values and comparing to the actual values. We're going to save the performance of all our models and compare them at the end of each section.

[6] If we were playing the "explanation game", we would call this ommitted variable bias

```
logit = glm(Survived ~ ., data=train, family="binomial")
summary(logit)
```

```
##
## Call:
## glm(formula = Survived ~ ., family = "binomial", data = train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.6381  -0.6118  -0.4149   0.5962   2.5101
##
## Coefficients:
##                           Estimate Std. Error z value Pr(>|z|)
## (Intercept)                3.61461    0.49768   7.263 3.79e-13 ***
## Pclass                    -1.04485    0.19286  -5.418 6.04e-08 ***
## Sexmale                   -2.67626    0.25202 -10.619  < 2e-16 ***
## Age                       -0.63279    0.14565  -4.345 1.40e-05 ***
## Siblings.Spouses.Aboard -0.49199    0.15312  -3.213  0.00131 **
## Parents.Children.Aboard -0.06323    0.16012  -0.395  0.69291
## Fare                       0.11115    0.18516   0.600  0.54833
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
```

```
##     Null deviance: 712.81  on 549  degrees of freedom
## Residual deviance: 479.31  on 543  degrees of freedom
## AIC: 493.31
##
## Number of Fisher Scoring iterations: 5
```

```
predicts.train <- predict(logit, train)
predicts.dev <- predict(logit, dev)

r.sq.train <- round(cor(predicts.train,as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(predicts.dev,as.numeric(y.dev))^2, digits=4)

logit.plain <- c('Low-dim Logit', r.sq.train, r.sq.dev)
```

We're going to be using the library glmnet to run our regularized regression[7]. We basically pass in a matrix of covariates, a vector of outcomes, what "family" our regression should be of (for a logit, that's binomial), and then two hyper-parameters called "alpha" and "lambda". The hyper-parameter "alpha" indicates whether you want to run a LASSO or a ridge regression. When alpha is equal to zero (one), the command runs a ridge (LASSO) regression[8]. The hyper-parameter "lambda" indicates how strong the penalty for beta size is; the higher the number, the stronger the penalty. By default (i.e. if we didn't put in a value for lambda) the function actually tests 100 different values of lambda for us. We're going to try just try one at a time here, though, so we can examine each model[9]. When we set lambda equal to zero, our regularized regression should be equivalent to our logistic regression. Let's verify that.

[7] You can type "?glmnet" into your console to see the documentation

[8] In-between values will actually weigh the different penalty terms accordingly (i.e. you can do a "half LASSO, half ridge regression")

[9] If you want to use this in actual research, you would of course prefer to test hundreds of values of lambda and would therefore let the function do this.

```
lasso <- glmnet(x.train, y.train, family="binomial", alpha=1, lambda = 0)
coefficients(lasso)
```

```
## 7 x 1 sparse Matrix of class "dgCMatrix"
##                                s0
## (Intercept)            -1.73821235
## Pclass                 -1.04468232
## Sex                     2.67622527
## Age                    -0.63275955
## Siblings.Spouses.Aboard -0.49198283
## Parents.Children.Aboard -0.06328008
## Fare                    0.11124881
```

Seems to check out (though for some reason the intercept is very different?). Now we're going to compare this baseline model to a LASSO and a ridge regression with non-zero penalty terms so that we can see how they differ.

```
lasso <- glmnet(x.train, y.train, family="binomial", alpha=1, lambda = 0.05)
coefficients(lasso)
```

```
## 7 x 1 sparse Matrix of class "dgCMatrix"
##                                s0
## (Intercept)            -2.2988596
## Pclass                 -0.4336175
## Sex                     1.9216636
## Age                     .
## Siblings.Spouses.Aboard .
## Parents.Children.Aboard .
## Fare                    .
```

```
ridge <- glmnet(x.train, y.train, family="binomial", alpha=0, lambda = 0.75)
coefficients(ridge)
```

```
## 7 x 1 sparse Matrix of class "dgCMatrix"
##                                 s0
## (Intercept)            -0.96588977
## Pclass                 -0.16101569
## Sex                     0.53652427
## Age                    -0.04466451
## Siblings.Spouses.Aboard -0.03547552
## Parents.Children.Aboard  0.03307689
## Fare                    0.10589555
```

So, in LASSO, there's a tendency to send "unimportant" variables to zero, whereas in ridge we tend to send them close to zero. LASSO can be useful in certain situations (we'll talk a little bit about this next week), but when we're doing straight prediction it only matters which one performs better.

##High-dimensional logit

But of course, the real advantage of using regularized regression is when we're in danger of over-fitting to the data. That will often happen when we have a lot of variables. Here we're going to create a data frame that has every possible interaction of our variables[10].

[10] Note that this isn't really "high-dimensional" compared to most machine learning applications where we have hundreds or thousands of features (independent variables), so we should actually expect over-fitting to be rather minimal here.

```
x.train.hd <- model.matrix(~ . * ., data=as.data.frame(x.train))
```

```
x.dev.hd <- model.matrix(~ . * ., data=as.data.frame(x.dev))
```

As a benchmark, we'll run a straight logistic regression with these same terms

```
logit <- glm(Survived ~ (.)^2, data=as.data.frame(train), family="binomial")
summary(logit)
```

```
##
## Call:
## glm(formula = Survived ~ (.)^2, family = "binomial", data = as.data.frame(train))
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4579  -0.6278  -0.4100   0.2627   2.6592
##
## Coefficients:
##                                                Estimate Std. Error z value
## (Intercept)                                    5.607747   1.652503   3.393
## Pclass                                        -1.398264   0.688323  -2.031
## Sexmale                                       -6.617707   1.649708  -4.011
## Age                                            0.455155   0.735277   0.619
## Siblings.Spouses.Aboard                        0.843053   1.199627   0.703
## Parents.Children.Aboard                        0.537386   1.369435   0.392
## Fare                                          -1.567911   0.836890  -1.873
## Pclass:Sexmale                                 1.373267   0.675027   2.034
## Pclass:Age                                    -0.168096   0.270396  -0.622
## Pclass:Siblings.Spouses.Aboard                -0.641706   0.503544  -1.274
## Pclass:Parents.Children.Aboard                -0.315352   0.486069  -0.649
## Pclass:Fare                                    1.471249   0.446942   3.292
## Sexmale:Age                                   -0.572854   0.409413  -1.399
## Sexmale:Siblings.Spouses.Aboard               -0.121790   0.408948  -0.298
## Sexmale:Parents.Children.Aboard                0.671189   0.568885   1.180
## Sexmale:Fare                                  -0.611755   0.844027  -0.725
## Age:Siblings.Spouses.Aboard                   -0.115052   0.290558  -0.396
## Age:Parents.Children.Aboard                   -0.491520   0.260375  -1.888
## Age:Fare                                       0.551776   0.381791   1.445
## Siblings.Spouses.Aboard:Parents.Children.Aboard -0.009297   0.279380  -0.033
## Siblings.Spouses.Aboard:Fare                  -0.066043   0.433248  -0.152
## Parents.Children.Aboard:Fare                  -0.100740   0.484814  -0.208
##                                               Pr(>|z|)
## (Intercept)                                   0.000690 ***
## Pclass                                        0.042214 *
## Sexmale                                       6.03e-05 ***
## Age                                           0.535899
## Siblings.Spouses.Aboard                       0.482204
## Parents.Children.Aboard                       0.694752
## Fare                                          0.061000 .
## Pclass:Sexmale                                0.041912 *
## Pclass:Age                                    0.534161
## Pclass:Siblings.Spouses.Aboard                0.202530
## Pclass:Parents.Children.Aboard                0.516480
```

```
## Pclass:Fare                                        0.000995 ***
## Sexmale:Age                                        0.161750
## Sexmale:Siblings.Spouses.Aboard                    0.765846
## Sexmale:Parents.Children.Aboard                    0.238067
## Sexmale:Fare                                       0.468572
## Age:Siblings.Spouses.Aboard                        0.692128
## Age:Parents.Children.Aboard                        0.059061 .
## Age:Fare                                           0.148393
## Siblings.Spouses.Aboard:Parents.Children.Aboard 0.973453
## Siblings.Spouses.Aboard:Fare                       0.878842
## Parents.Children.Aboard:Fare                       0.835393
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 712.81  on 549  degrees of freedom
## Residual deviance: 422.04  on 528  degrees of freedom
## AIC: 466.04
##
## Number of Fisher Scoring iterations: 7
```

```
predicts.train <- predict(logit, train)
predicts.dev <- predict(logit, dev)

r.sq.train <- round(cor(predicts.train,as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(predicts.dev,as.numeric(y.dev))^2, digits=4)

logit.hd <- cbind('Hi-dim Logit', r.sq.train, r.sq.dev)
```

Now we'll run various LASSO regressions with different values of lambda. Notice how more variables are sent to zero as we increase the lambda value, with very high penalties leading the model to be almost equivalent to a constant-only model.

##Low penalty LASSO

```
lasso <- glmnet(x.train.hd, y.train, family="binomial", alpha=1, lambda = 0.01)
coefficients(lasso)
```

```
## 23 x 1 sparse Matrix of class "dgCMatrix"
##                                                    s0
## (Intercept)                                -0.04010523
## (Intercept)                                 .
## Pclass                                     -0.61820255
## Sexfemale                                   3.02274693
## Age                                        -0.28114023
```

```
## Siblings.Spouses.Aboard                              .
## Parents.Children.Aboard                              .
## Fare                                                 .
## Pclass:Sexfemale                                    -0.20855253
## Pclass:Age                                          -0.07409738
## Pclass:Siblings.Spouses.Aboard                      -0.17051831
## Pclass:Parents.Children.Aboard                       .
## Pclass:Fare                                          0.05506333
## Sexfemale:Age                                        0.45697937
## Sexfemale:Siblings.Spouses.Aboard                    .
## Sexfemale:Parents.Children.Aboard                   -0.02279342
## Sexfemale:Fare                                       0.18120164
## Age:Siblings.Spouses.Aboard                          .
## Age:Parents.Children.Aboard                         -0.32793491
## Age:Fare                                             0.13662590
## Siblings.Spouses.Aboard:Parents.Children.Aboard      .
## Siblings.Spouses.Aboard:Fare                         .
## Parents.Children.Aboard:Fare                         .
```

```
predicts.train <- predict(lasso, x.train.hd)
predicts.dev <- predict(lasso, x.dev.hd)

r.sq.train <- round(cor(predicts.train,as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(predicts.dev,as.numeric(y.dev))^2, digits=4)

lasso.low <- cbind('Low penalty lasso', r.sq.train, r.sq.dev)
```

##Medium penalty LASSO

```
lasso <- glmnet(x.train.hd, y.train, family="binomial", alpha=1, lambda = 0.1)
coefficients(lasso)
```

```
## 23 x 1 sparse Matrix of class "dgCMatrix"
##                                                      s0
## (Intercept)                                    -0.8267729
## (Intercept)                                     .
## Pclass                                         -0.1456544
## Sexfemale                                       1.4418379
## Age                                             .
## Siblings.Spouses.Aboard                         .
## Parents.Children.Aboard                         .
## Fare                                            .
## Pclass:Sexfemale                                .
## Pclass:Age                                      .
## Pclass:Siblings.Spouses.Aboard                  .
## Pclass:Parents.Children.Aboard                  .
```

```
## Pclass:Fare                                          .
## Sexfemale:Age                                        .
## Sexfemale:Siblings.Spouses.Aboard                    .
## Sexfemale:Parents.Children.Aboard                    .
## Sexfemale:Fare                                       .
## Age:Siblings.Spouses.Aboard                          .
## Age:Parents.Children.Aboard                          .
## Age:Fare                                             .
## Siblings.Spouses.Aboard:Parents.Children.Aboard      .
## Siblings.Spouses.Aboard:Fare                         .
## Parents.Children.Aboard:Fare                         .
```

```
predicts.train <- predict(lasso, x.train.hd)
predicts.dev <- predict(lasso, x.dev.hd)

r.sq.train <- round(cor(predicts.train,as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(predicts.dev,as.numeric(y.dev))^2, digits=4)

lasso.med <- cbind('Medium penalty lasso', r.sq.train, r.sq.dev)
```

##High penalty LASSO

```
lasso <- glmnet(x.train.hd, y.train, family="binomial", alpha=1, lambda = 0.2)
coefficients(lasso)
```

```
## 23 x 1 sparse Matrix of class "dgCMatrix"
##                                                      s0
## (Intercept)                                  -0.8104141
## (Intercept)                                   .
## Pclass                                        .
## Sexfemale                                     0.5500275
## Age                                           .
## Siblings.Spouses.Aboard                       .
## Parents.Children.Aboard                       .
## Fare                                          .
## Pclass:Sexfemale                              .
## Pclass:Age                                    .
## Pclass:Siblings.Spouses.Aboard                .
## Pclass:Parents.Children.Aboard                .
## Pclass:Fare                                   .
## Sexfemale:Age                                 .
## Sexfemale:Siblings.Spouses.Aboard             .
## Sexfemale:Parents.Children.Aboard             .
## Sexfemale:Fare                                .
## Age:Siblings.Spouses.Aboard                   .
## Age:Parents.Children.Aboard                   .
```

```
## Age:Fare                                          .
## Siblings.Spouses.Aboard:Parents.Children.Aboard   .
## Siblings.Spouses.Aboard:Fare                      .
## Parents.Children.Aboard:Fare                      .
```

```
predicts.train <- predict(lasso, x.train.hd)
predicts.dev <- predict(lasso, x.dev.hd)

r.sq.train <- round(cor(predicts.train,as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(predicts.dev,as.numeric(y.dev))^2, digits=4)

lasso.hi <- cbind('High penalty lasso', r.sq.train, r.sq.dev)
```

##Results

Now we'll make a nice little table to examine all the results of these models

Model	r.squared (train)	r.squared (dev)
Low-dim Logit	0.3744	0.4233
Hi-dim Logit	0.4362	0.3394
Low penalty lasso	0.4138	0.0884
Medium penalty lasso	0.3325	0.3286
High penalty lasso	0.2975	0.383

Notice how, even with only 22 variables, we see pretty significant over-fitting from the logistic regression in high-dimensions. If you just compared the different R-square values for the data we trained on, you'd think the high-dimensional logit would be our best model! You can imagine this is much worse when we have even more variables. Also notice how, when we get to very high levels of lambda, we start to fit even our training data poorly. Remember that our "best model" is the one that predicts the dev set the best. After we pick out best models, we finally evaluate their performance on the test set.

#Decision trees

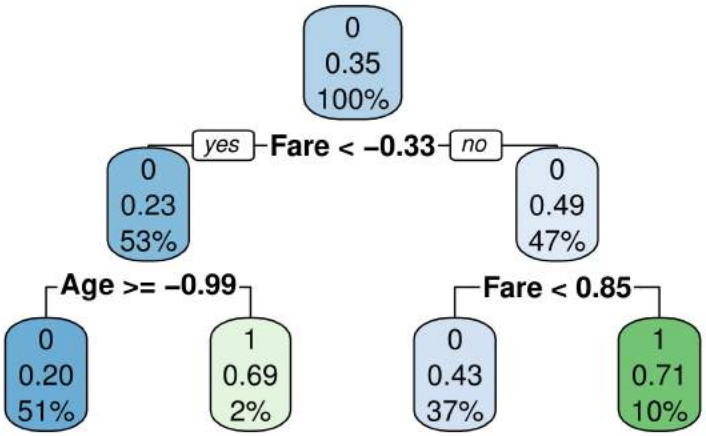
The next algorithm we'll examine is a "decision tree". This looks for ways to "split" the data so as to optimize prediction. To implement this algorithm, we'll be using the rpart library, which is pretty standard for data science or machine learning applications in R[11].

[11] Type "?rpart" in the console to see the documentation for these decision trees, and all the hyper-parameters you can play with

##A simple example

First we're going to "grow" a tree (train the algorithm) using just "Age" and "Fare". For this round, we'll restrict the tree to a "depth" of two.

```
decision.tree <- rpart(Survived ~ Age + Fare, data = train, method="class", maxdepth=2)
rpart.plot(decision.tree)
```

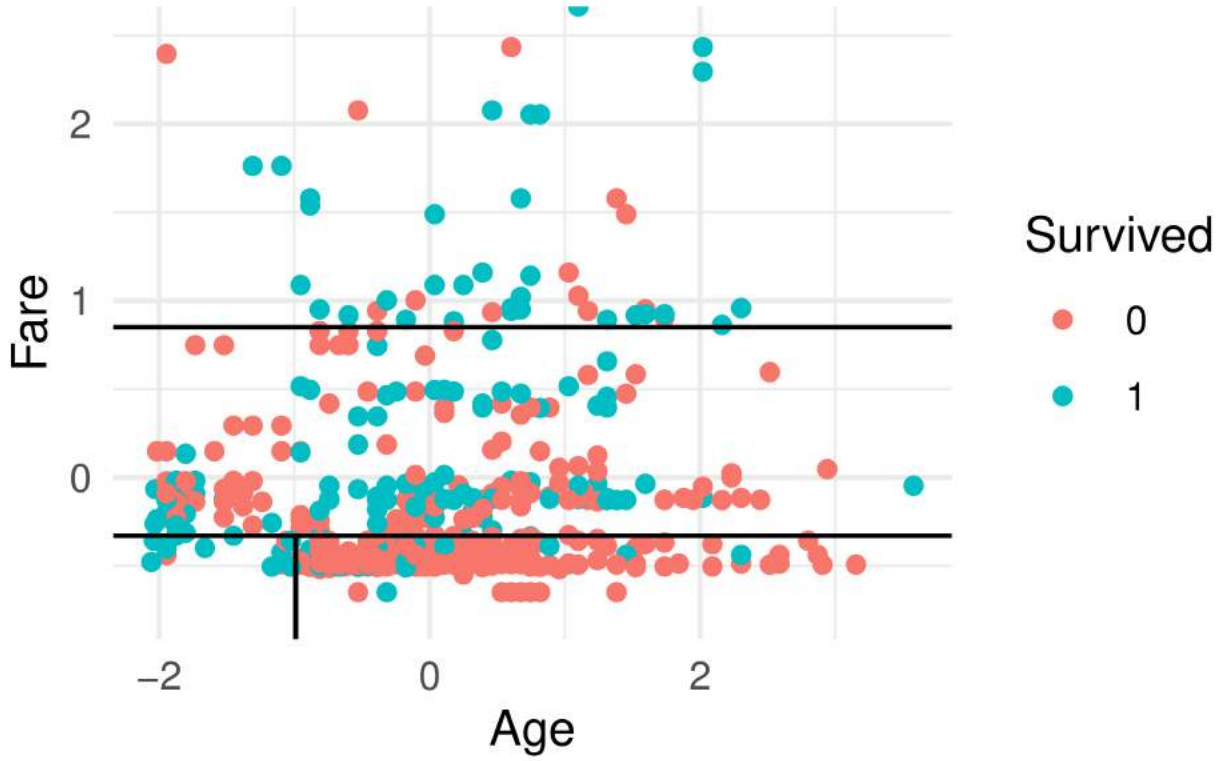


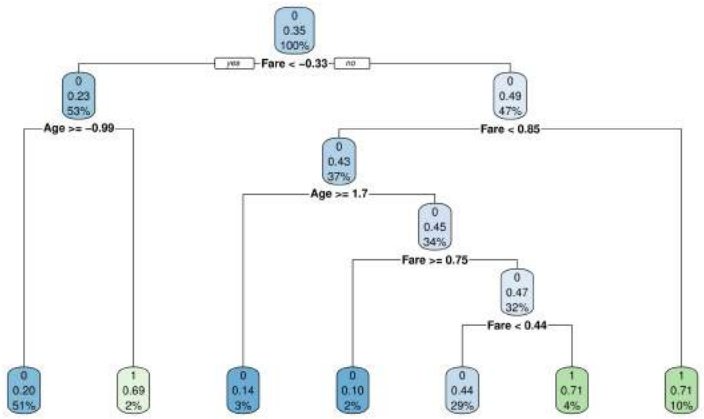
Now we'll plot the covariate space and draw lines where these splits are made so we can see what the algorithm is doing.

```
ggplot(train, aes(x=Age, y=Fare)) +
  geom_point(aes(col=Survived)) +
  geom_hline(yintercept=-0.33) +
  geom_segment(x=-0.99, y=-0.34, xend=-0.99, yend=-1) +
  geom_hline(yintercept=0.85) +
  coord_cartesian(ylim=c(-0.75, 2.5)) +
  theme_minimal()
```

Now let's grow a tree that is allowed to go "deeper" and repeat the same procedure.

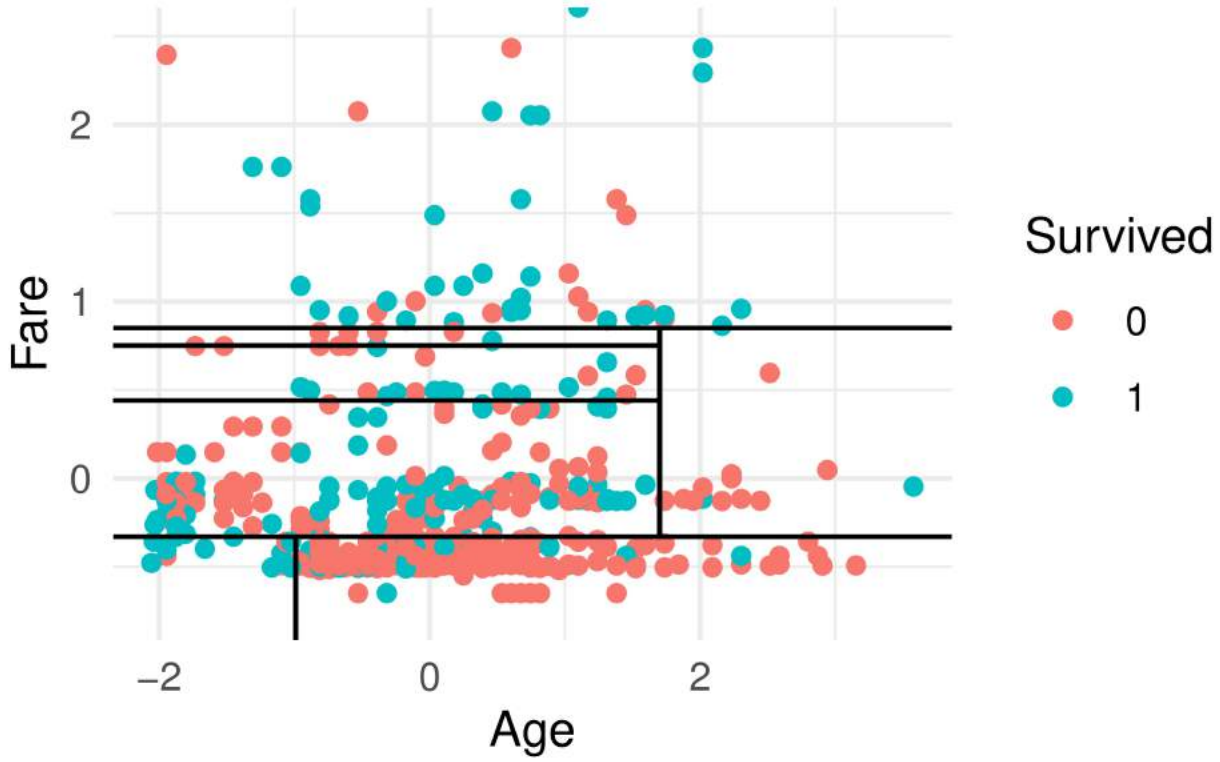
```
decision.tree <- rpart(Survived ~ Age + Fare, data = train, method="class", maxdepth=5)
rpart.plot(decision.tree)
```






```
ggplot(train, aes(x=Age, y=Fare)) +
  geom_point(aes(col=Survived)) +
  geom_hline(yintercept=-0.33) +
  geom_segment(x=-0.99, y=-0.34, xend=-0.99, yend=-1) +
  geom_hline(yintercept=0.85) +
  geom_segment(x=1.7, y=-0.33, xend=1.7, yend=0.85) +
```

```
geom_segment(x=-3, y=0.75, xend=1.7, yend=0.75) +
geom_segment(x=-3, y=0.44, xend=1.7, yend=0.44) +
coord_cartesian(ylim=c(-0.75, 2.5)) +
theme_minimal()
```



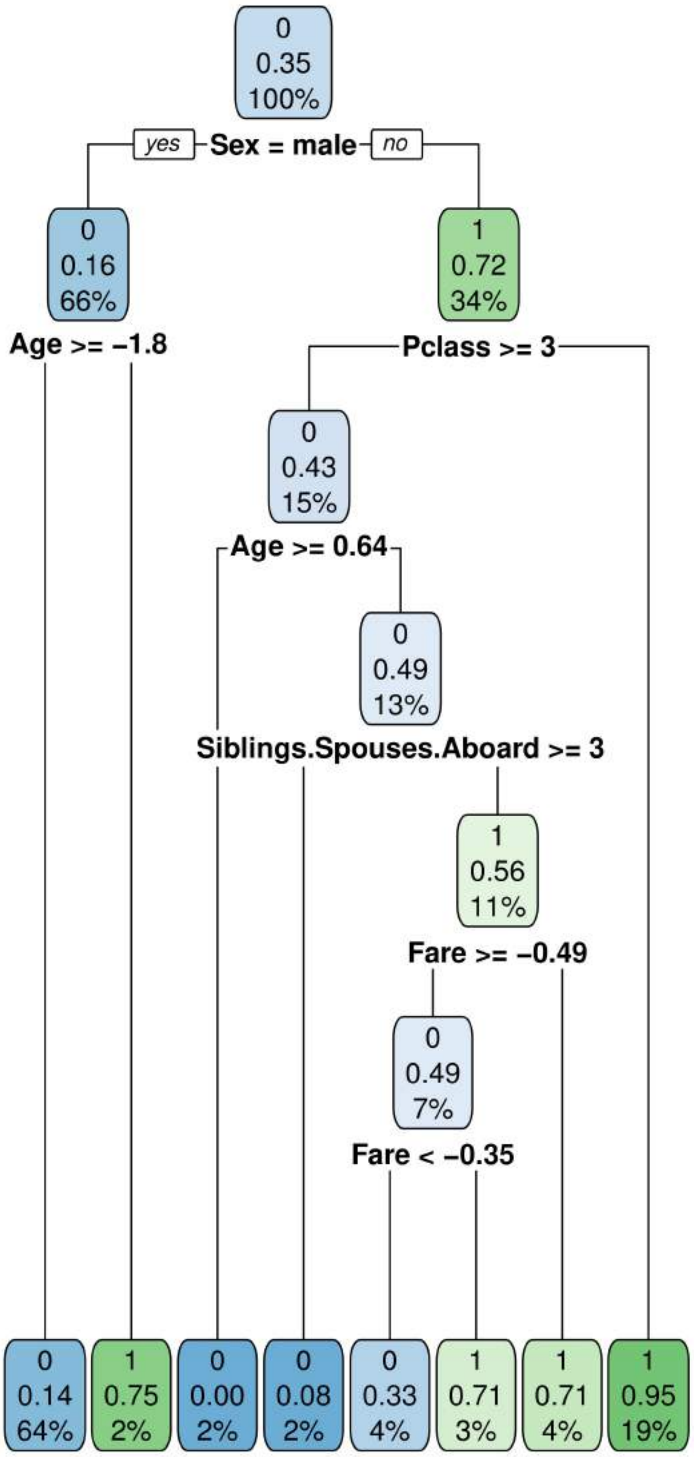
So when we're fitting these trees, imagine it doing the same thing but in a higher-dimensional space (a dataset where we have more variables). Variables that are less important won't tend to be split on automatically[12]. Now let's try to make some competitive models with all of our variables.

##Out-of-the-box decision tree

Let's start with the default values for our decision tree.

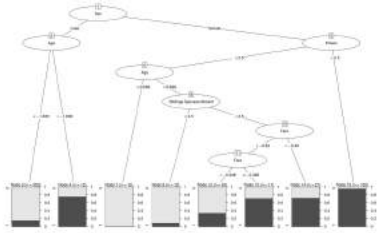
[12] There are ways to examine "variable importance" from these models, but I'm not a huge fan of the very idea so we won't cover that here

```
decision.tree <- rpart(Survived ~ ., data = train, method="class")
rpart.plot(decision.tree)
```


0
0.35
100%
yes
Sex = male
no
0
0.16
66%
1
0.72
34%
Age >= −1.8
Pclass >= 3
0
0.43
15%
Age >= 0.64
0
0.49
13%
Siblings.Spouses.Aboard >= 3
1
0.56
11%
Fare >= −0.49
0
0.49
7%
Fare < −0.35
0
0.14
64%
1
0.75
2%
0
0.00
2%
0
0.08
2%
0
0.33
4%
1
0.71
3%
1
0.71
4%
1
0.95
19%

```
predicts.train <- predict(decision.tree, train, method="class")[,2]
predicts.dev <- predict(decision.tree, dev, method="class")[,2]

r.sq.train <- round(cor(predicts.train, as.numeric(train$Survived))^2, digits=4)
r.sq.dev <- round(cor(predicts.dev, as.numeric(dev$Survived))^2, digits=4)

unrestricted <- cbind("OOB tree", r.sq.train, r.sq.dev)


tree <- as.party(decision.tree)
plot(tree)
```

There is another library partykit which allows you to visualize the decision tree differently (I'm told it looks better on Macs)...

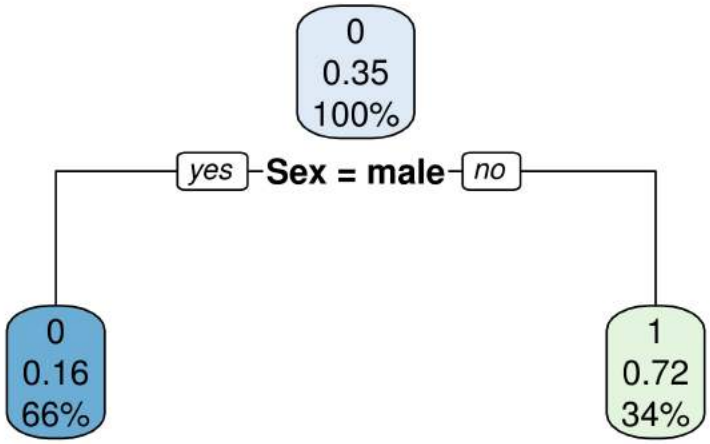


##Restricted Trees

Now, we might be concerned that our decision trees are overfitting to the data, which decision trees have a tendency to do. One way that we can mitigate this is to restrict our decision trees from going a certain depth. Let's try out different values for how deep a tree is allowed to go and compare their performances. We'll also visualize each tree, as this might give you a better sense of what variables are really important here and how these decision trees work.

###Restricted to 1 split

```
decision.tree <- rpart(Survived ~ ., data = train, method="class", maxdepth=1)
rpart.plot(decision.tree)
```

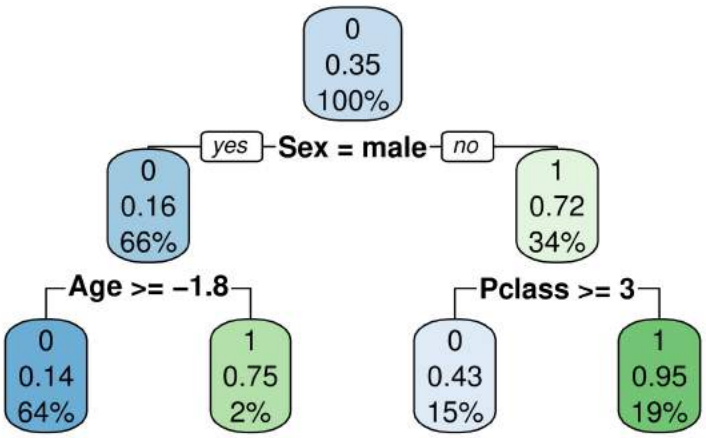



```
predicts.train <- predict(decision.tree, train, method="class")[,2]
predicts.dev <- predict(decision.tree, dev, method="class")[,2]

r.sq.train <- round(cor(predicts.train, as.numeric(y.train))^2, digits=4)
```

```
r.sq.dev <- round(cor(predicts.dev, as.numeric(y.dev))^2, digits=4)

depth.1 <- cbind("Tree depth 1", r.sq.train, r.sq.dev)
```

###Restricted to 2 splits

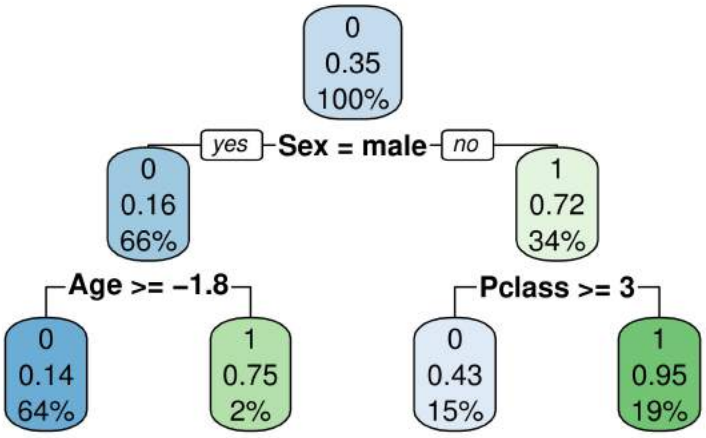
```
decision.tree <- rpart(Survived ~ ., data = train, method="class", maxdepth=2)
rpart.plot(decision.tree)
```




```
predicts.train <- predict(decision.tree, train, method="class")[,2]
predicts.dev <- predict(decision.tree, dev, method="class")[,2]

r.sq.train <- round(cor(predicts.train, as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(predicts.dev, as.numeric(y.dev))^2, digits=4)

depth.2 <- cbind("Tree depth 2", r.sq.train, r.sq.dev)
```

###Restricted to 3 splits

```
decision.tree <- rpart(Survived ~ ., data = train, method="class", maxdepth=3)
rpart.plot(decision.tree)
```

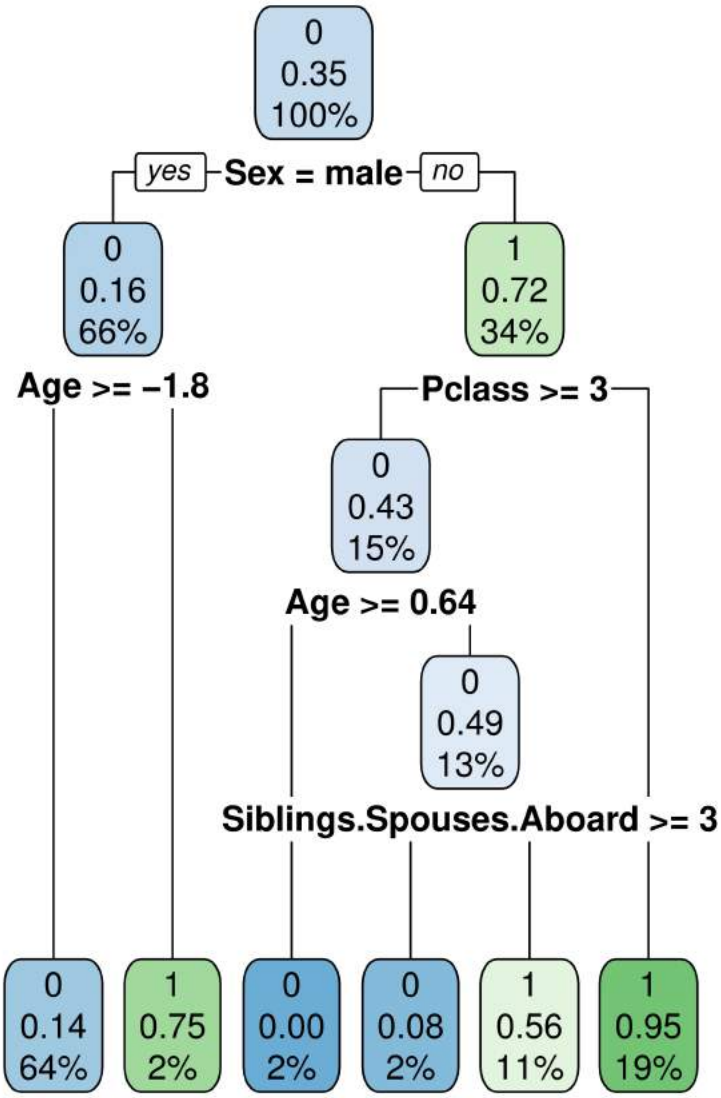



```
predicts.train <- predict(decision.tree, train, method="class")[,2]
predicts.dev <- predict(decision.tree, dev, method="class")[,2]

r.sq.train <- round(cor(predicts.train, as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(predicts.dev, as.numeric(y.dev))^2, digits=4)

depth.3 <- cbind("Tree depth 3", r.sq.train, r.sq.dev)
```

This would be a good spot to note that these decision trees have other restrictions applied to them. For instance, each tree has a "complexity factor", which is how much it "prefers" to make fewer splits. Check out the rpart documentation for default hyper-parameter values.

###Restricted to 4 splits

```
decision.tree <- rpart(Survived ~ ., data = train, method="class", maxdepth=4)
rpart.plot(decision.tree)
```

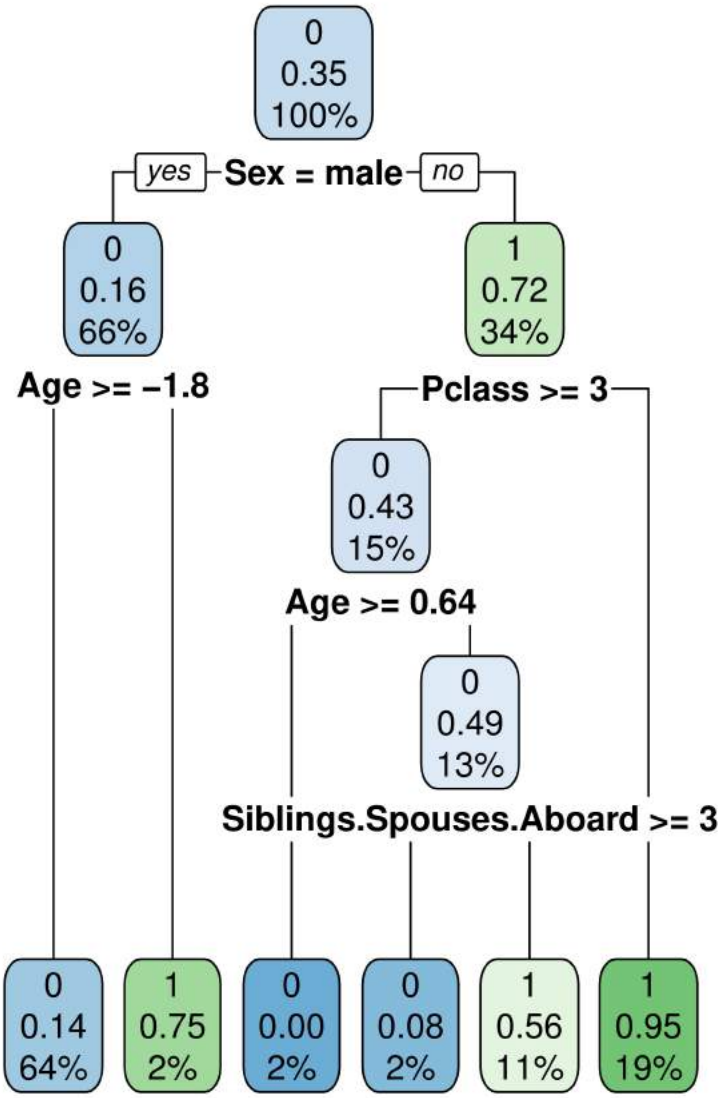



```
predicts.train <- predict(decision.tree, train, method="class")[,2]
predicts.dev <- predict(decision.tree, dev, method="class")[,2]

r.sq.train <- round(cor(predicts.train, as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(predicts.dev, as.numeric(y.dev))^2, digits=4)

depth.4 <- cbind("Tree depth 4", r.sq.train, r.sq.dev)
```

###Restricted to 5 splits

```
decision.tree <- rpart(Survived ~ ., data = train, method="class", maxdepth=5)
```

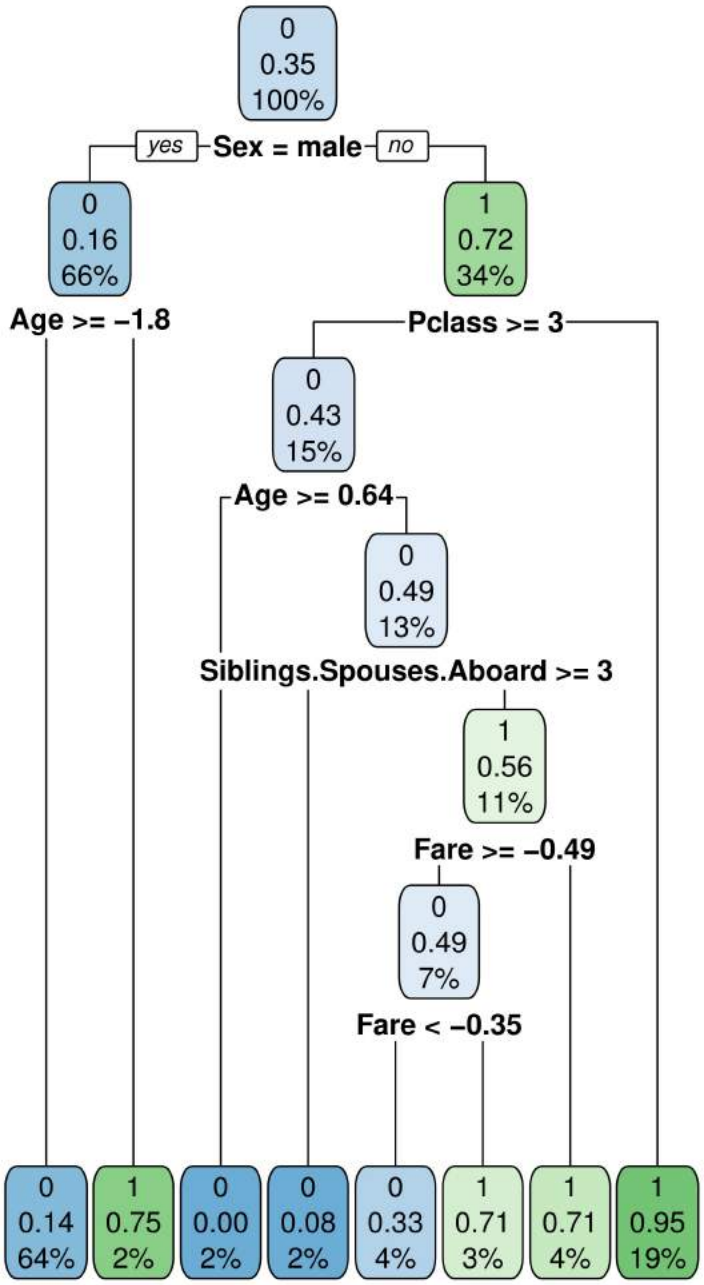
```
rpart.plot(decision.tree)
```




```
predicts.train <- predict(decision.tree, train, method="class")[,2]
predicts.dev <- predict(decision.tree, dev, method="class")[,2]

r.sq.train <- round(cor(predicts.train, as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(predicts.dev, as.numeric(y.dev))^2, digits=4)

depth.5 <- cbind("Tree depth 5", r.sq.train, r.sq.dev)
```

###Restricted to 6 splits

```
decision.tree <- rpart(Survived ~ ., data = train, method="class", maxdepth=6)
rpart.plot(decision.tree)
```



```
predicts.train <- predict(decision.tree, train, method="class")[,2]
predicts.dev <- predict(decision.tree, dev, method="class")[,2]

r.sq.train <- round(cor(predicts.train, as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(predicts.dev, as.numeric(y.dev))^2, digits=4)

depth.6 <- cbind("Tree depth 6", r.sq.train, r.sq.dev)
```

##Results

Now let's see how these different decision trees did on our training and dev set.

```
results <- as.data.frame(rbind(unrestricted, depth.1, depth.2, depth.3, depth.4, depth.5, depth.6))
colnames(results) <- c('Model','r.squared (train)', 'r.squared (dev)')
kable(results)
```

Model	r.squared (train)	r.squared (dev)
OOB tree	0.4837	0.391
Tree depth 1	0.2975	0.383
Tree depth 2	0.4318	0.3502
Tree depth 3	0.4318	0.3502
Tree depth 4	0.467	0.3773
Tree depth 5	0.467	0.3773
Tree depth 6	0.4837	0.391

So, while we were "over-fitting" the data in the sense that we consistently perform worse on our dev set than our training set, it looks like allowing the tree to go as deep as it would want to by default is the winning strategy. Keep in mind that this is going to be different for different tasks and different data, especially when we have many more observations and many more variables.

#Random Forests

A random forest is where we implement the following procedure:

For as many trees as you plan to grow:

1. take a random sample from your training data
2. (optional) take a random sample of your variables
3. "grow" a decision tree on that data

Then, to predict the outcome for an observation, get the prediction according to every tree you've grown and take a majority vote (or, in the case of regression trees, the average prediction across all trees). This is an implementation of a **meta algorithm**, which means this same procedure could be applied to regressions, neural nets,

or whatever (though it's not called a "random forest" then). There are also other meta-algorithms that are really powerful including **gradient boosting**, but we won't get around to talking about that unfortunately.

There are of course many hyper-parameters to be set (type "?randomForest" to see all of them and the default values), but here we'll just vary how many "trees" are in our "forest". As before, we'll record all of their performances and compare at the end.

##10 trees

```
rf <- randomForest(Survived ~ .,data=train, ntree=10)

predicts.train <- predict(rf, train)
predicts.dev <- predict(rf, dev)

r.sq.train <- round(cor(as.numeric(predicts.train), as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(as.numeric(predicts.dev), as.numeric(y.dev))^2, digits=4)

rf.10 <- cbind('10 tree forest', r.sq.train, r.sq.dev)
```

##100 trees

```
rf <- randomForest(Survived ~ .,data=train, ntree=100)

predicts.train <- predict(rf, train)
predicts.dev <- predict(rf, dev)

r.sq.train <- round(cor(as.numeric(predicts.train), as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(as.numeric(predicts.dev), as.numeric(y.dev))^2, digits=4)

rf.100 <- cbind('100 tree forest', r.sq.train, r.sq.dev)
```

##500 trees

```
rf <- randomForest(Survived ~ .,data=train, ntree=500)

predicts.train <- predict(rf, train)
predicts.dev <- predict(rf, dev)

r.sq.train <- round(cor(as.numeric(predicts.train), as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(as.numeric(predicts.dev), as.numeric(y.dev))^2, digits=4)

rf.500 <- cbind('500 tree forest', r.sq.train, r.sq.dev)
```

##1000 trees

```
rf <- randomForest(Survived ~ .,data=train, ntree=1000)

predicts.train <- predict(rf, train)
predicts.dev <- predict(rf, dev)

r.sq.train <- round(cor(as.numeric(predicts.train), as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(as.numeric(predicts.dev), as.numeric(y.dev))^2, digits=4)

rf.1k <- cbind('1k tree forest', r.sq.train, r.sq.dev)
```

##5000 trees This one will take a second (but not as long as you might expect when you think about it!)

```
rf <- randomForest(Survived ~ .,data=train, ntree=5000)

predicts.train <- predict(rf, train)
predicts.dev <- predict(rf, dev)

r.sq.train <- round(cor(as.numeric(predicts.train), as.numeric(y.train))^2, digits=4)
r.sq.dev <- round(cor(as.numeric(predicts.dev), as.numeric(y.dev))^2, digits=4)

rf.5k <- cbind('5k tree forest', r.sq.train, r.sq.dev)
```

##Results

```
results <- as.data.frame(rbind(rf.10, rf.100, rf.500, rf.1k, rf.5k))
colnames(results) <- c('Model','r.squared (train)', 'r.squared (dev)')
kable(results)
```

Model	r.squared (train)	r.squared (dev)
10 tree forest	0.6247	0.46
100 tree forest	0.6452	0.4646
500 tree forest	0.6516	0.4804
1k tree forest	0.6383	0.4646
5k tree forest	0.6516	0.4646

So it looks like we don't get much more predictive power by going with more than 500 trees. We'll just implement the 1k trees forest, then.

#Testing on the test set

So now we've gone through all of the hyper-parameter values we want to test[13]. Let's quickly review the results from all of our models...

##Results on train and dev sets

[13] If you were actually doing a project with machine learning, you'd want to run thousands of models testing as many hyper-parameter values as you could to find the best model before testing on the test set

```
results <- rbind(logit.plain, logit.hd,
                 lasso.low, lasso.med, lasso.hi,
                 unrestricted, depth.1, depth.2, depth.3, depth.4, depth.5, depth.6,
                 rf.10, rf.100, rf.500, rf.1k, rf.5k)

rownames(results) <- c()
colnames(results) <- c('Model', 'r.squared (train)', 'r.squared (dev)')

kable(results)
```

Model	r.squared (train)	r.squared (dev)
Low-dim Logit	0.3744	0.4233
Hi-dim Logit	0.4362	0.3394
Low penalty lasso	0.4138	0.0884
Medium penalty lasso	0.3325	0.3286
High penalty lasso	0.2975	0.383
OOB tree	0.4837	0.391
Tree depth 1	0.2975	0.383
Tree depth 2	0.4318	0.3502
Tree depth 3	0.4318	0.3502
Tree depth 4	0.467	0.3773
Tree depth 5	0.467	0.3773
Tree depth 6	0.4837	0.391
10 tree forest	0.6247	0.46
100 tree forest	0.6452	0.4646
500 tree forest	0.6516	0.4804
1k tree forest	0.6383	0.4646
5k tree forest	0.6516	0.4646

So, our final test will be for "Medium penalty LASSO", "OOB tree", and "1k forest".

#Final testing

Now we'll re-train our models (just because they got over-written), predict our values for the test set, and see how we did.

```
y.test <- as.numeric(test$Survived)

x.test <- test
x.test$Survived <- NULL
x.test.hd <- model.matrix(~ . * ., data=as.data.frame(x.test))

final.lasso <- glmnet(x.train.hd, y.train, family="binomial", alpha=1, lambda = 0.1)
final.tree <- rpart(Survived ~ ., data = train, method="class")
```

```
final.forest <- randomForest(Survived ~ .,data=train, ntree=100)

test.predicts.lasso <- predict(final.lasso, x.test.hd)
test.predicts.tree <- predict(final.tree, test, type = 'prob')
test.predicts.forest <- predict(final.forest, test, type='prob')

perf.lasso <- round(abs(cor(test.predicts.lasso,y.test)), digits=4)
perf.tree <- round(cor(test.predicts.tree,y.test), digits=4)[2]
perf.forest <- round(cor(test.predicts.forest,y.test), digits=4)[2]

lasso <- cbind('LASSO', perf.lasso)
tree <- cbind('Decision tree', perf.tree)
forest <- cbind('Forest', perf.forest)
```

And the final results table!

```
results <- rbind(lasso, tree, forest)

rownames(results) <- c()
colnames(results) <- c('Model', 'r.squared (test)')

kable(results)
```

Model	r.squared (test)
LASSO	0.3714
Decision tree	0.5692
Forest	0.6343

And it looks like the random forest performed the best! Now, importantly, you shouldn't walk away form this thinking "random forests are better than LASSO", because it completely depends on the data and the problem. You should always be checking these models, as well as many others. Remember that when we're playing the "prediction game", it is ALL ABOUT increasing performance. You get into the proceedings of a computer science conference by performing better than everyone else. There are other ways, but if you get state-of-the-art results, you pretty much guaranteed what is essentially a publication to computer scientists.