Introduction to Machine Learning in R

Evan Muzzall and Chris Kennedy

4/23/2018

Table of Contents

Introduction

Visit the UC Berkeley D-Lab to learn more about our services and resources, including the Machine Learning Working Group.

Resources

An Introduction to Statistical Learning - with Applications in R Amazon or free PDF. We encourage you to support the authors by purchasing their textbook!

Package installation

The following packages are required to run the code in this workshop:

```
if (FALSE) {
  # Run this line manually to install the necessary packages.
install.packages(c("caret","chemometrics", "ck37r", "class",
                   "devtools", "gbm", "ggplot2", "gmodels",
                   "pROC", "randomForest", "rpart", "rpart.plot",
                   "SuperLearner"))
}

library(caret)
library(chemometrics)
library(ck37r)
library(class)
library(devtools)
library(gbm)
library(ggplot2)
library(gmodels)
library(pROC)
library(randomForest)
library(rpart)
library(rpart.plot)
library(SuperLearner)
```

Brief history of machine learning

Machine learning evolved from scientific pursuits in computational/information theory, artificial intelligence, and pattern recognition.

How to define machine learning?
1) **In general:** algorithms, computers, and other machines that can "learn" without direct input from a human programmer.
2) **Practically:** sets of tools for investigating/modeling/understanding data.
3) **Specifically:** (see below)

A proto-example:
- Pascal's calculator

Rapid advances:
- McCulloch Pitts neuron model
- Turing test
- Rosenblatt's perceptron
- Samuels and the game of checkers

Modern topics:
- Turing Test: 50 years later

- computer vision
- data cleaning
- machine learning software, tools, and methods
- robotics
- cloud computing

The importance of statistics:
- Welling's commentary
- Srivastava's discussion

Seek "actionable insight":
- "actionable insight"

Supervised machine learning

Selecting a machine learning algorithm depends on the characteristics of the problem being investigated - there is no "best" method applicable to all cases. Machine learning is generally divided into three broad classes of learning: supervised, unsupervised, and reinforcement. In this workshop we will focus on two main subtypes of supervised machine learning: classification and regression.

The syntax for supervised machine learning algorithms can be thought of like this:

$Y \sim X_1 + X_2 + X_3 \ldots X_n$

Y is the dependent/response/target/outcome variable
X are the independent/input/predictor/feature variables

Supervised machine learning methods learn a target function f that best maps X to Y based on a set of training data.

Our function would look like this: $\hat{y} = f(X) + \epsilon$, where f is some function that relates our X predictor variables to Y in an unknown way thus we must estimate it. Therefore, we can predict Y using $\hat{y} = \hat{f}(X)$ for new data (call the test dataset) and evaluate how well the algorithm learned the target function when introduced to new data. Epsilon ϵ is the random error, is independent of X, and averages to zero.

How to define machine learning? (revisited)
More specifically, we can think of machine learning as a bunch of methods to estimate f!

Classification or regression?

Classification is used when the `Y` outcome variable is categorical/discrete. Binary examples generally refer to a yes/no situation where a `1` is prediction of the "yes" category and `0` as the "no". Classification models the probability that the outcome variable is 1 based on the covariates: $Pr(Y = 1|X)$. This can be extended to multi-level classification as well.

Regression is used when the target Y variable is continuous. Regression models the conditional expectation (conditional mean) of the outcome variable given the covariates: $E(Y|X)$.

Data preprocessing

A longstanding first step is to split a dataset into **"training"** and **"test"** subsets. A training dataset usually consists of a majority portion of the original dataset so that an algorithm can learn the model. The remaining portion of the dataset is designated to the test dataset to evaluate model performance on data the model has not yet seen. **Missing data should be handled** before the splitting process commences.

Model performance

Performance metrics are used to see how well a model predicts a specified outcome on training and test datasets.

A model that performs poorly on the training dataset is **underfit** because it is not able to discern relationships between the X and Y variables.

A model that performs well on the training dataset but poorly on the test dataset is saide to be **overfit** because the model performed poorly when given new data - the patterns found in the test data could not be discerned or simply might not exist in the test data.

Common performance metrics

Accuracy
Mean squared error
Sensitivity and specificity
Area under the ROC curve (AUC)
*Cross validated error

Workshop goals

General goals

1) Learn the basics of coding six machine learning algorithms in R:

- k-nearest neighbor
- linear regression
- decision tree
- random forest

- boosting
- SuperLearner

2) Examine the performances of these models

3) Simultaneously compare multiple algorithms in an ensemble

4) Vizualize important information:

- knn accuracy tables
- decision trees
- random forest relative variable importance
- random forest variable importance
- AUC from different boosting models
- SuperLearner cross-validated risk

Specific goals

Use the `PimaIndiansDiabetes2` dataset from the mlbench package to investigate the following questions:

1) **Binary classification examples:** How reliably can machine learning algorithms predict a person's diabetes status using the other variables?

2) **Regression example:** How well can a person's age be predicted using the other variables?

What are these other variables?

Load the data

Load the `PimaIndiansDiabetes2` and `iris` datasets

```
library(mlbench)

# load the PimaIndiansDiabetes2 dataset
data("PimaIndiansDiabetes2")

# read variable descriptions
?PimaIndiansDiabetes2
```

```
# rename the dataset to something simpler (pidd = "Pima Indians Diabetes Data
set")
pidd = PimaIndiansDiabetes2

# view the sturcture of pidd
str(pidd)
```

```
## 'data.frame':    768 obs. of  9 variables:
##  $ pregnant: num  6 1 8 1 0 5 3 10 2 8 ...
##  $ glucose : num  148 85 183 89 137 116 78 115 197 125 ...
##  $ pressure: num  72 66 64 66 40 74 50 NA 70 96 ...
##  $ triceps : num  35 29 NA 23 35 NA 32 NA 45 NA ...
##  $ insulin : num  NA NA NA 94 168 NA 88 NA 543 NA ...
##  $ mass    : num  33.6 26.6 23.3 28.1 43.1 25.6 31 35.3 30.5 NA ...
##  $ pedigree: num  0.627 0.351 0.672 0.167 2.288 ...
##  $ age     : num  50 31 32 21 33 30 26 29 53 54 ...
##  $ diabetes: Factor w/ 2 levels "neg","pos": 2 1 2 1 2 1 2 1 2 2 ...
```

```
# also load iris dataset for challenge questions
data(iris)
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1
1 1 1 1 ...
```

```
?iris
```

Data preprocessing

Data peprocessing is an integral first step in machine learning workflows. Because different algorithms sometimes require the moving parts to be coded in slightly different ways, always make sure you research the algorithm you want to implement so that you properly identify your Y and X variables and appropriately split your data into training and test sets if neeeded.

One additional preprocessing aspect to consider: datasets that contain factor (categorical) features should be expanded out into numeric indicators (this is often referred to as one-hot encoding. You can do this manually with the `model.matrix` R function. This makes it easier to code a variety of algorithms to a dataset as many algorithms handle factors poorly (decision trees being the main exception). When we predict age in our regression example below, we will do this manually for the "diabetes" column to practice. However, functions like the `lm` function will internally expand factor variables such as the `diabetes` factor predictor into numeric indicators.

NOTE: Keep in mind that training/test dataset splitting is common, but not always preferred. We will introduce you to cross-validation in the second half of this workshop where *all* the data is used and multiple training/testing configurations are utilized.

Handling missing data

Missing values need to be handled somehow. Listwise deletion (deleting any row with at least one missing case) is common but this method throws out a lot of useful information. Many advocate for mean imputation, but arithmetic means sensitive to outliers. Still, others advocate for Chained Equation/Bayesian/Expectation Maximization imputation (e.g., the mice and Amelia II R packages).

However, median imputation is demonstrated below for the classification examples

```
# First, count the number of missing cases in our pidd dataset
sum(is.na(pidd)) # 652 missing cases
```

```
## [1] 652
```

```
# Then, compute the proportion of missing cases in pidd
sum(is.na(pidd)) / (nrow(pidd) * ncol(pidd)) # ~9% of cases pidd is missing
```

```
## [1] 0.0943287
```

Now, median impute the missing values! We also want to create missingness indicators to inform us about the location of missing data. Thus, we will add some additional columns to our data frame.

Neither the "diabetes" nor "age" columns have any missing cases, so we can go ahead and impute the whole dataset!

```
library(ck37r)

# run the median impute function
result = impute_missing_values(pidd)

# Use the imputed dataframe.
pidd = result$data

# view new columns
str(pidd)
```

```
## 'data.frame':    768 obs. of  14 variables:
##  $ pregnant     : num  6 1 8 1 0 5 3 10 2 8 ...
##  $ glucose      : num  148 85 183 89 137 116 78 115 197 125 ...
##  $ pressure     : num  72 66 64 66 40 74 50 72 70 96 ...
##  $ triceps      : num  35 29 29 23 35 29 32 29 45 29 ...
##  $ insulin      : num  125 125 125 94 168 125 88 125 543 125 ...
##  $ mass         : num  33.6 26.6 23.3 28.1 43.1 25.6 31 35.3 30.5 32.3 ...
##  $ pedigree     : num  0.627 0.351 0.672 0.167 2.288 ...
```

```
##  $ age          : num  50 31 32 21 33 30 26 29 53 54 ...
##  $ diabetes     : Factor w/ 2 levels "neg","pos": 2 1 2 1 2 1 2 1 2 2 ...
##  $ miss_glucose : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ miss_pressure: num  0 0 0 0 0 0 0 1 0 0 ...
##  $ miss_triceps : num  0 0 1 0 0 1 0 1 0 1 ...
##  $ miss_insulin : num  1 1 1 0 0 1 0 1 0 1 ...
##  $ miss_mass    : num  0 0 0 0 0 0 0 0 0 1 ...

# No more missing data!
sum(is.na(pidd)) # 0 missing cases

## [1] 0
```

Defining Y outcome vectors and X feature dataframes

Classification setup

Assign outcome to its own vector for **CLASSIFICATION tasks:** k-nearest neighbor, decision tree, random forest, gradient boosting, and SuperLearner algorithms. However, keep in mind that these algorithms can also perform regression!

```
# View pidd variable names
names(pidd)

##  [1] "pregnant"      "glucose"       "pressure"      "triceps"
##  [5] "insulin"       "mass"          "pedigree"      "age"
##  [9] "diabetes"      "miss_glucose"  "miss_pressure" "miss_triceps"
## [13] "miss_insulin"  "miss_mass"

# 1) Define Y for classification (has diabetes? "pos" or "neg")
Y_fac = pidd$diabetes
head(Y_fac, n = 20)

##  [1] pos neg pos neg pos neg pos neg pos pos neg pos neg pos pos pos pos
## [18] pos neg pos
## Levels: neg pos

# 2) Then, convert "pos" to 1 and "neg" to 0. Many algorithms expect 1's for
the positive class and 0's for the negative class.
Y = ifelse(Y_fac == "pos", 1, 0)
head(Y, n = 20)

##  [1] 1 0 1 0 1 0 1 0 1 1 0 1 0 1 1 1 1 1 0 1

# 3) Finally, define the X feature/predictor dataframe that excludes Y
features = subset(pidd, select = -diabetes)
head(features) # "diabetes" column has been successfully removed

##   pregnant glucose pressure triceps insulin mass pedigree age miss_glucose
## 1        6     148       72      35     125 33.6    0.627  50            0
## 2        1      85       66      29     125 26.6    0.351  31            0
```

```
## 3        8     183       64      29     125 23.3    0.672  32            0
## 4        1      89       66      23      94 28.1    0.167  21            0
## 5        0     137       40      35     168 43.1    2.288  33            0
## 6        5     116       74      29     125 25.6    0.201  30            0
##   miss_pressure miss_triceps miss_insulin miss_mass
## 1             0            0            1         0
## 2             0            0            1         0
## 3             0            1            1         0
## 4             0            0            0         0
## 5             0            0            0         0
## 6             0            1            1         0
```

We then can take the "traditional" approach to data splitting and divide our data into training and test sets; 70% of the data will be assigned to the training set and the remaining 30% will be assigned to the holdout, or test, set.

```
library(caret)

## Loading required package: lattice

## Loading required package: ggplot2

# set seed for reproducibility
set.seed(1)

# Create a stratified random split
classification_split = createDataPartition(Y, p=0.70, list=FALSE)

train_X = features[classification_split, ] # partition training dataset
test_X = features[-classification_split, ] # partition test dataset

train_label = Y[classification_split] # partition training Y vector labels
test_label = Y[-classification_split] # partition test Y vector labels

# lengths of our Y label vectors and the  number of rows in our training data
frames are the same for both training and test sets!
dim(train_X)

## [1] 538  13

length(train_label)

## [1] 538

dim(test_X)

## [1] 230  13

length(test_label)

## [1] 230
```

Regression setup

We could do something similar for our lone **REGRESSION task:** linear regression. There are forms of regression that can perform classification (such as `glm`), but the one we use here - `lm` - is not one of them. Hence, we can only predict a continuous/integer Y output variable.

Instead, we will use R's handy `predict` function to simply create a test data set based on predictions on all the data.

Algorithm walkthroughs

1. k-nearest neighbor (knn)

The k-nearest neighbor (knn) algorithm is a good machine learning start point because it makes no assumptions about the underlying distribution of the data. To classify a data point, knn uses the characteristics of the points around it to determine how to classify it. Euclidean distances between points are used in this example.

`k` is the number of neighbors used to classify the point in question. Choosing a proper `k` is integral to finding the best-performing model. **Large k-values** could be bad because the class with the largest size might win regardless of influence of closer points. **Small k-values** might also be bad because neighboring points might become overly influential.

1.1 knn - selecting a starting "k-value"

Different methods exist for choosing a start point for "k". Let's use the square root of the number of rows in the training datset:

```
round(sqrt(nrow(train_X)), digits=2) # 23.19

## [1] 23.19
```

Fit the model! Our goal is to predict the classification accuracy of our `Y` variable (whether or not a person has diabetes based on the other `train_X` predictor variables; 1 = yes/pos, 0 = no/neg:

```
library(class)
set.seed(1)
data_predicted = knn(train = train_X, test = test_X,
                     cl = train_label, k = 23, prob = TRUE)
```

Using accuracy as our performance metric, compute a contingency table to see how well the model predicted yes and no:

```
library(gmodels)
CrossTable(x = test_label, y = data_predicted,
           prop.chisq = F,
```

```
           prop.r = F,
           prop.c = F,
           prop.t = F)

##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |-------------------------|
##
##
## Total Observations in Table:  230
##
##
##              | data_predicted
##   test_label |         0 |         1 | Row Total |
## -------------|-----------|-----------|-----------|
##            0 |       122 |        26 |       148 |
## -------------|-----------|-----------|-----------|
##            1 |        39 |        43 |        82 |
## -------------|-----------|-----------|-----------|
## Column Total |       161 |        69 |       230 |
## -------------|-----------|-----------|-----------|
##
##

# Compute accuracy - how did we do?
mean(test_label == data_predicted) # ~ 0.72

## [1] 0.7173913
```

How did it do?

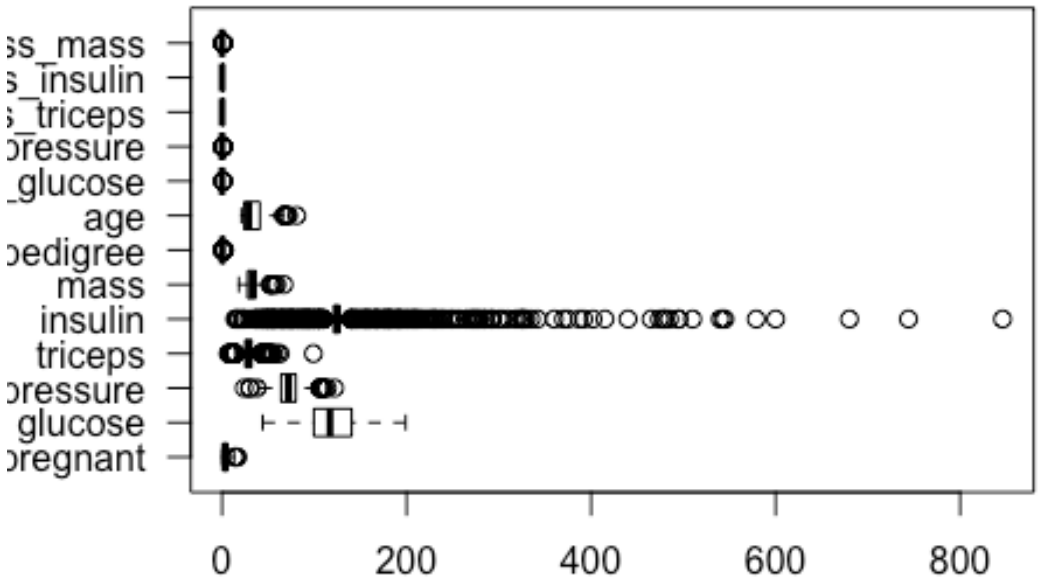
1.2 knn - improving performance metrics

Two common ways of improving model performance are to 1) standardize the data so that scale does not unduly influence classification, and 2) change the k-value.

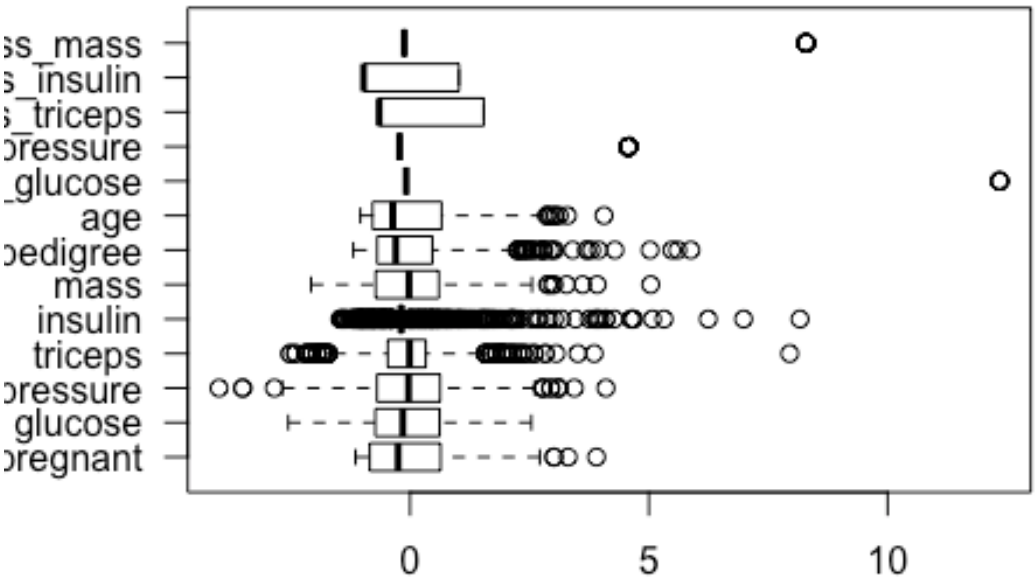
1.3 knn - scale the data

Scaling the data is useful so that variables with large values do not bias the prediction. Create a new copy of the "data" dataset called "data_scaled", which will contain scaled values with means equal to 0 and standard deviations equal to 1. Our Y variable remains unchanged:

```
data_scaled = scale(features, center = TRUE, scale = TRUE)

# Look at the new distributions
boxplot(features, horizontal = T, las = 1)
```




```r
boxplot(data_scaled, horizontal = T, las = 1)
```



Repeat the process:
1) split the scaled data

```
# Use "split" to create the training partition.
set.seed(1)
train_scaled = data_scaled[classification_split, ]

# Create the training set
training_scaled = data_scaled[classification_split, ]

# Create the test set
test_scaled = data_scaled[-classification_split, ]

# Extract outcome data for training set
train_label_scaled = Y[classification_split]

# Extract outcome data for test set
test_label_scaled = Y[-classification_split]
```

2) fit the model on the scaled data

```
library(class)
set.seed(1)
data_predicted_scaled = knn(train = train_scaled,
```

```
                              test = test_scaled,
                              cl = train_label_scaled,
                              k = 23, prob = TRUE)
```

3) examine the accuracy of predictions on the scaled data:

```
library(gmodels)
CrossTable(x = test_label_scaled, y = data_predicted_scaled,
           prop.chisq = F,
           prop.r = F,
           prop.c = F,
           prop.t = F)

##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |-------------------------|
##
##
## Total Observations in Table:  230
##
##
##                    | data_predicted_scaled
## test_label_scaled |         0 |         1 | Row Total |
## ------------------|-----------|-----------|-----------|
##                 0 |       124 |        24 |       148 |
## ------------------|-----------|-----------|-----------|
##                 1 |        42 |        40 |        82 |
## ------------------|-----------|-----------|-----------|
##      Column Total |       166 |        64 |       230 |
## ------------------|-----------|-----------|-----------|
##
##

# Compute accuracy
mean(test_label_scaled == data_predicted_scaled) # ~0.71

## [1] 0.7130435
```

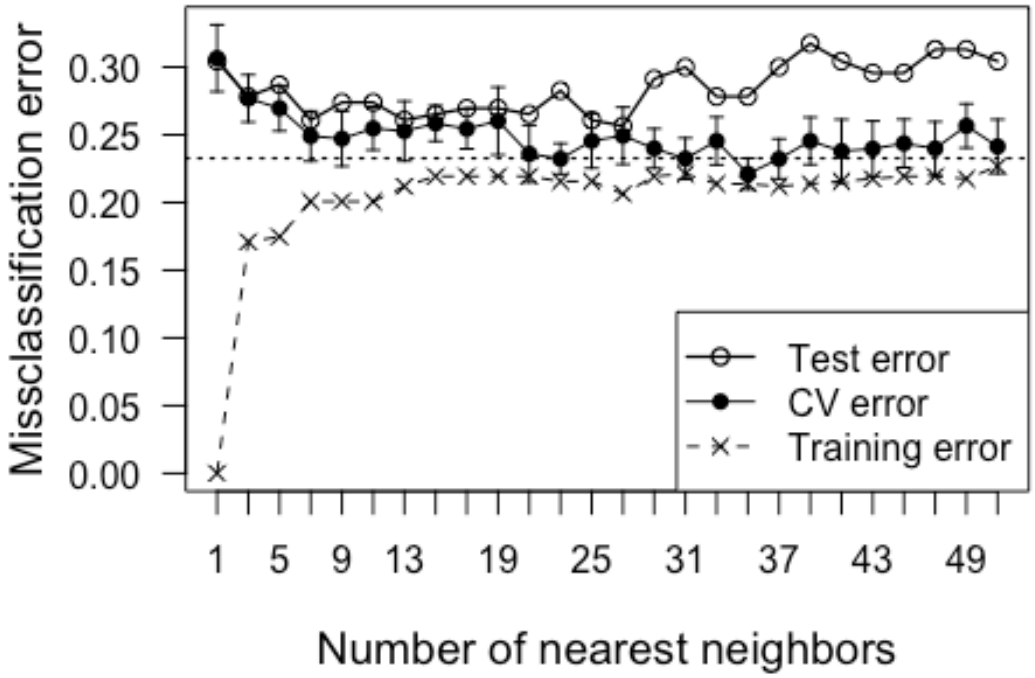
Did scaling the data help?

1.4 knn - improving performance metrics - change k-value

Scaling our data actually *reduced* our accuracy a little. It also helps to investigate the cross-validated errors for multiple k-values at once to see which is ideal. Plot the cross-validated errors:

```
library(chemometrics)

## Loading required package: rpart
```

```
set.seed(1)
knn_k = knnEval(X = pidd[,-9], grp = pidd[,9],
                train = classification_split,
                knnvec=seq(1, 51, by = 2), kfold = 10,
                legpo="bottomright", las = 2)
```



A combination of scaling the data and searching for the best cross-validated "k" can help find the proper k value.

Big question 1: What might you conclude about the k-nearest neighbors algorithm and this particular dataset? (see the solutions file for an answer)

Challenge 1

Using what you learned above, classify knn predictive accuracies of the `Species` variable in the "iris" dataset.

2. Linear regression

Ordinary least squares regression (OLS) can be used when the target Y variable is continuous. Remember that under the hood, `lm` is one-hot encoding factors to indicators.

Recall that we have already defined our Y outcome variable "age" and stored it in the object `Y_reg` as well as our dataframe of features, stored in `data_reg`. Let's fit a model that predicts the `Y_reg` using the other variables in `data_reg` as predictors.

Mean squared error (MSE) will be our performance metric. MSE measures the difference between observed and expected values, with smaller values tending to reflect greater predictive accuracy.

```
# Fit the regression model.
pidd_age = lm(pidd$age ~ ., data = subset(pidd, select = -age))

# View the regression results.
summary(pidd_age)
```

```
##
## Call:
## lm(formula = pidd$age ~ ., data = subset(pidd, select = -age))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -24.146  -5.440  -1.724   3.045  40.358
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)    8.135187   2.565603   3.171 0.001581 **
## pregnant       1.560681   0.104323  14.960  < 2e-16 ***
## glucose        0.051182   0.013761   3.719 0.000215 ***
## pressure       0.186320   0.029948   6.221 8.16e-10 ***
## triceps        0.097062   0.045228   2.146 0.032186 *
## insulin        0.004982   0.004321   1.153 0.249248
## mass          -0.205587   0.061500  -3.343 0.000870 ***
## pedigree       1.847823   1.040058   1.777 0.076028 .
## diabetespos    1.047557   0.846020   1.238 0.216020
## miss_glucose  -1.395688   4.150332  -0.336 0.736751
## miss_pressure -4.300987   1.770415  -2.429 0.015358 *
## miss_triceps   2.796141   1.011765   2.764 0.005856 **
## miss_insulin   1.438815   0.912538   1.577 0.115280
## miss_mass     -1.549375   2.974561  -0.521 0.602607
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 9.158 on 754 degrees of freedom
## Multiple R-squared:  0.4038, Adjusted R-squared:  0.3936
## F-statistic: 39.29 on 13 and 754 DF,  p-value: < 2.2e-16
```

```
# Predict the outcome back onto the data
pidd_age_predicted = predict(pidd_age, pidd)

# Calculate mean-squared error.
MSE = mean((pidd$age - pidd_age_predicted)^2)
```

```
MSE
```

```
## [1] 82.34319
```

```
sqrt(MSE) #RMSE
```

```
## [1] 9.074315
```

Big question 2: What surmise about linear regression and this dataset? (see the solutions file for an answer)

Challenge 2

Code a regression model that predicts one of the numeric variables from the "iris" dataset.

3. Decision trees

Decision trees are recursive partitioning methods that divides the predictor spaces into simpler regions and can be visualized in a tree-like structure. Decesion trees attempt to classify data by dividing it into subsets according to a `Y` output variable and based on some predictors.

Let's see how - Let's see how a tree-based method classifies women's participation in the workforce.

Note that we do not have to use `model.matrix` for our decision tree algorithm here because it is adept at handling factor variables:

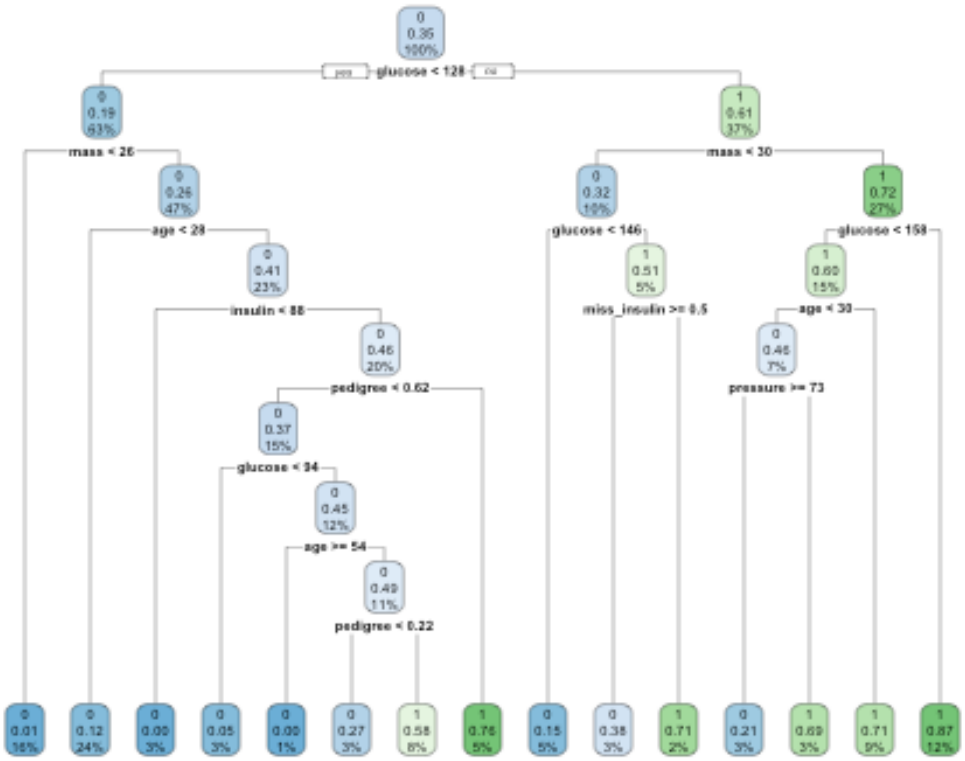
```
library(rpart)

dec_tree = rpart(Y ~ ., data = features,
            method = "class", # or method ="anova" for a regression tree
            parms = list(split = "information")) # or "gini" for gini coeffic
ient

# Here is the text-based display of the decision tree. Yikes!  :^(
print(dec_tree)
```

```
## n= 768
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##   1) root 768 268 0 (0.651041667 0.348958333)
##     2) glucose< 127.5 485  94 0 (0.806185567 0.193814433)
##       4) mass< 26.45 122   1 0 (0.991803279 0.008196721) *
##       5) mass>=26.45 363  93 0 (0.743801653 0.256198347)
##        10) age< 28.5 188  22 0 (0.882978723 0.117021277) *
##        11) age>=28.5 175  71 0 (0.594285714 0.405714286)
```

```
##          22) insulin< 88 22   0 0 (1.000000000 0.000000000) *
##          23) insulin>=88 153  71 0 (0.535947712 0.464052288)
##            46) pedigree< 0.625 116  43 0 (0.629310345 0.370689655)
##              92) glucose< 93.5 22   1 0 (0.954545455 0.045454545) *
##              93) glucose>=93.5 94  42 0 (0.553191489 0.446808511)
##               186) age>=54.5 8   0 0 (1.000000000 0.000000000) *
##               187) age< 54.5 86  42 0 (0.511627907 0.488372093)
##                 374) pedigree< 0.219 26   7 0 (0.730769231 0.269230769) *
##                 375) pedigree>=0.219 60  25 1 (0.416666667 0.583333333) *
##            47) pedigree>=0.625 37   9 1 (0.243243243 0.756756757) *
##     3) glucose>=127.5 283 109 1 (0.385159011 0.614840989)
##       6) mass< 29.95 75  24 0 (0.680000000 0.320000000)
##        12) glucose< 145.5 40   6 0 (0.850000000 0.150000000) *
##        13) glucose>=145.5 35  17 1 (0.485714286 0.514285714)
##          26) miss_insulin>=0.5 21   8 0 (0.619047619 0.380952381) *
##          27) miss_insulin< 0.5 14   4 1 (0.285714286 0.714285714) *
##       7) mass>=29.95 208  58 1 (0.278846154 0.721153846)
##        14) glucose< 157.5 116  46 1 (0.396551724 0.603448276)
##          28) age< 30.5 50  23 0 (0.540000000 0.460000000)
##            56) pressure>=73 24   5 0 (0.791666667 0.208333333) *
##            57) pressure< 73 26   8 1 (0.307692308 0.692307692) *
##          29) age>=30.5 66  19 1 (0.287878788 0.712121212) *
##        15) glucose>=157.5 92  12 1 (0.130434783 0.869565217) *
```

Although interpreting the text can be intimidating, a decision tree's main strength is its tree-like plot, which is much easier to interpret

```
library(rpart.plot)
rpart.plot(dec_tree) # shadow.col = "lightgray", cex = 1, type = 4, extra = 1
01)
```



We can also look inside of `dec_tree` to see what we can unpack. "variable.importance" stands out as one we should check out!

```
names(dec_tree)
```

```
##  [1] "frame"               "where"               "call"
##  [4] "terms"               "cptable"             "method"
##  [7] "parms"               "control"             "functions"
## [10] "numresp"             "splits"              "variable.importance"
## [13] "y"                   "ordered"
```

```
dec_tree$variable.importance
```

```
##      glucose         mass          age      insulin     pressure
##    94.835611    53.485216    38.950283    34.504995    17.966184
##     pedigree     pregnant      triceps miss_insulin miss_triceps
##    17.748828    14.919632    10.278076     7.077762     5.396164
```

In decision trees the main hyperparameter (configuration setting) is the **complexity parameter** (CP), but the name is a little counterintuitive; a high CP results in a simple decision tree with few splits, whereas a low CP results in a larger decision tree with many splits.

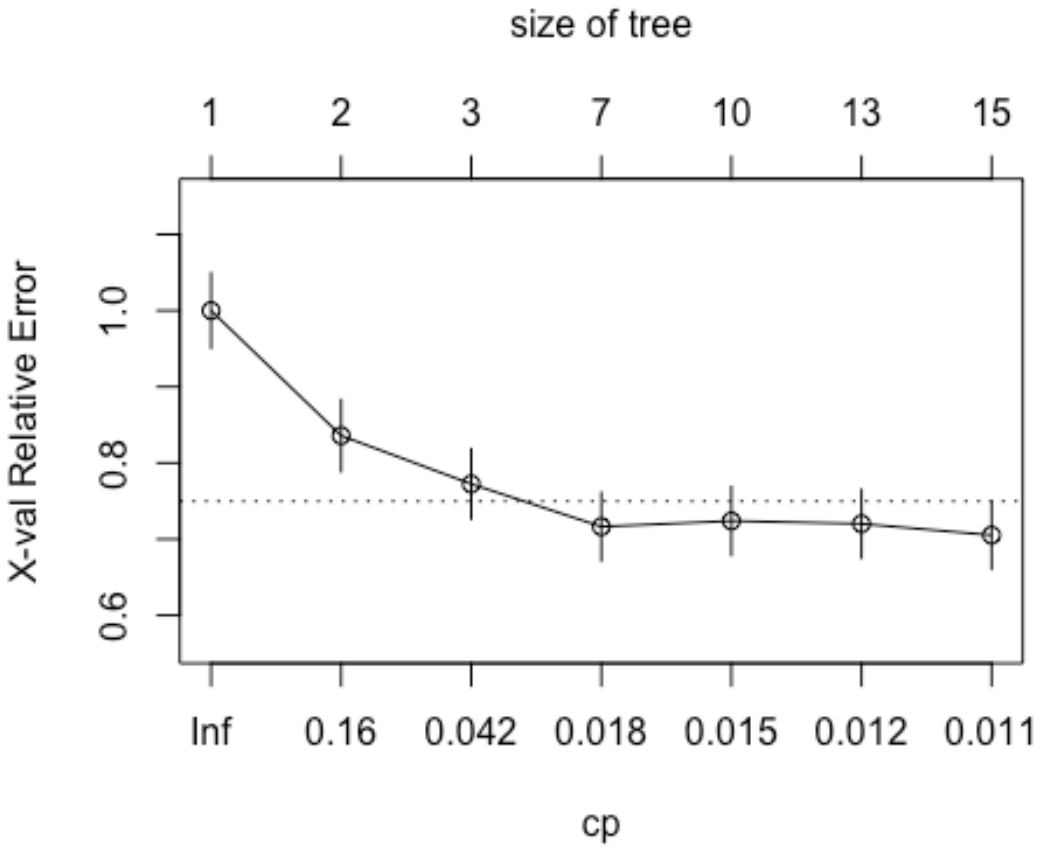
rpart uses cross-validation internally to estimate the accuracy at various CP settings. We can review those to see what setting seems best.

Print the results for various CP settings - we want the one with the lowest "xerror". We can also plot the performance estimates for different CP settings.

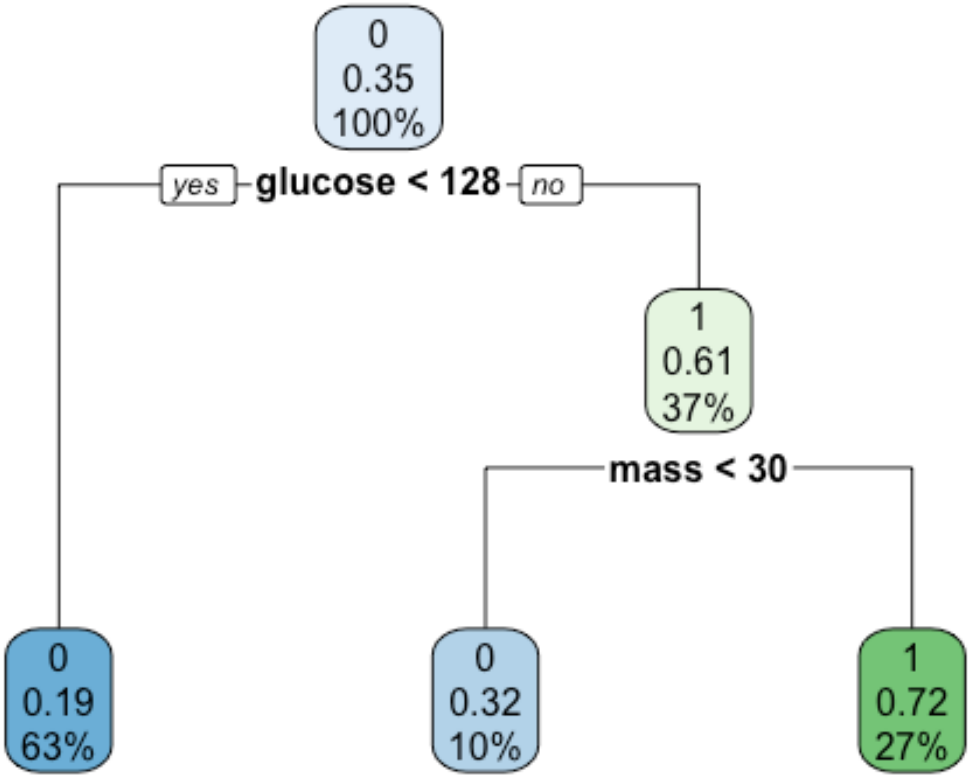
```
printcp(dec_tree) # print various CP settings
```

```
##
## Classification tree:
## rpart(formula = Y ~ ., data = features, method = "class", parms = list(spl
it = "information"))
##
## Variables actually used in tree construction:
## [1] age          glucose      insulin      mass         miss_insulin
## [6] pedigree     pressure
##
## Root node error: 268/768 = 0.34896
##
## n= 768
##
##         CP nsplit rel error  xerror     xstd
## 1 0.242537      0   1.00000 1.00000 0.049288
## 2 0.100746      1   0.75746 0.83582 0.047001
## 3 0.017724      2   0.65672 0.77239 0.045883
## 4 0.017413      6   0.58582 0.71642 0.044776
## 5 0.012438      9   0.53358 0.72388 0.044931
## 6 0.011194     12   0.49627 0.72015 0.044854
## 7 0.010000     14   0.47388 0.70522 0.044540
```

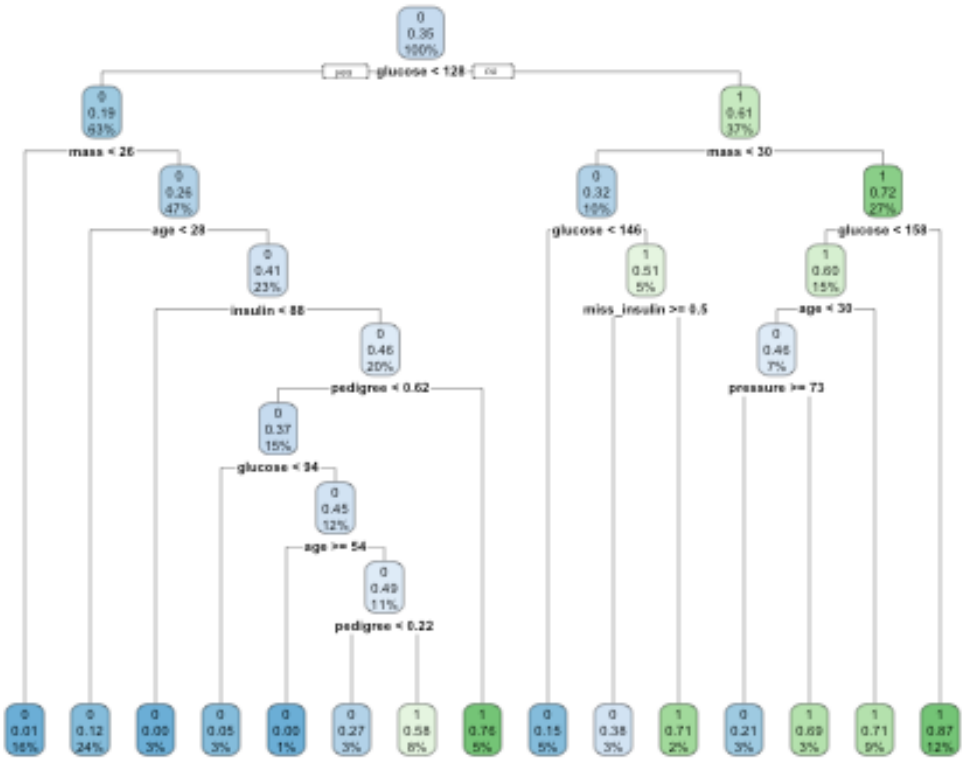
```
plotcp(dec_tree) # Plot their performance estimates
```




```
# 2 or 14 splits appear to be tied for lowest "xerror", but a tree with fewer
splits might be easier to interpret. However, a tree with 14 splits has a low
er relative error.
tree_pruned2 = prune(dec_tree, cp = 0.017724) # 2 splits

tree_pruned14 = prune(dec_tree, cp = 0.010000) # 14 splits

# Print detailed results, variable importance, and summary of splits.

summary(tree_pruned2)

rpart.plot(tree_pruned2)
```




```
summary(tree_pruned14)

rpart.plot(tree_pruned14)
```



You can also get more fine-grained control by checking out the "control" argument inside the rpart function. Type ?rpart to learn more.

Big question 3: What do you notice about the tree with 2 splits and the tree with 14 splits? Are there any parts that are identical?

Challenge 3

What are the "minsplit", "cp", and "minbucket" hyperparameters within the "control" parameter? Use the iris dataset to construct a decision tree that utilized the `rpart.control` hyperparameter.

HINT: the syntax might look like this: `ctrl = rpart.control(minsplit = 20, minbucket = 5, cp = 0.001)`

4. Random forests

The random forest algorithm seeks to improve on the performance of a single decision tree by taking the average of many trees. Thus, a random forest is an **ensemble** method, or model averaging approach. The algorithm was invented by Berkeley's own Leo Breiman in 2001, who was also a co-creator of decision trees (see his 1984 CART book).

Random forests are an extension of **bagging**, in which multiple samples of the original data are drawn with replacement (aka "bootstrap samples"), an algorithm is fit separately to each sample, then the average of those estimates is used for prediction. While bagging can be used any algorithm, random forest uses decision trees as its base learner. Random forests add another level of randomness by also randomly sampling the features (or covariates) at each split in each decision tree. This makes the decision trees use different covariates and therefore be more unique. As a result, the average of these trees tends to be more accurate overall.

Fit a random forest model that tries to predict the number of people with diabetes using the other variables as our X predictors. If our Y variable is a factor, randomForest will by default perform classification; it is is numeric/integer regression will be performed and if it is omitted it will become unsupervised!

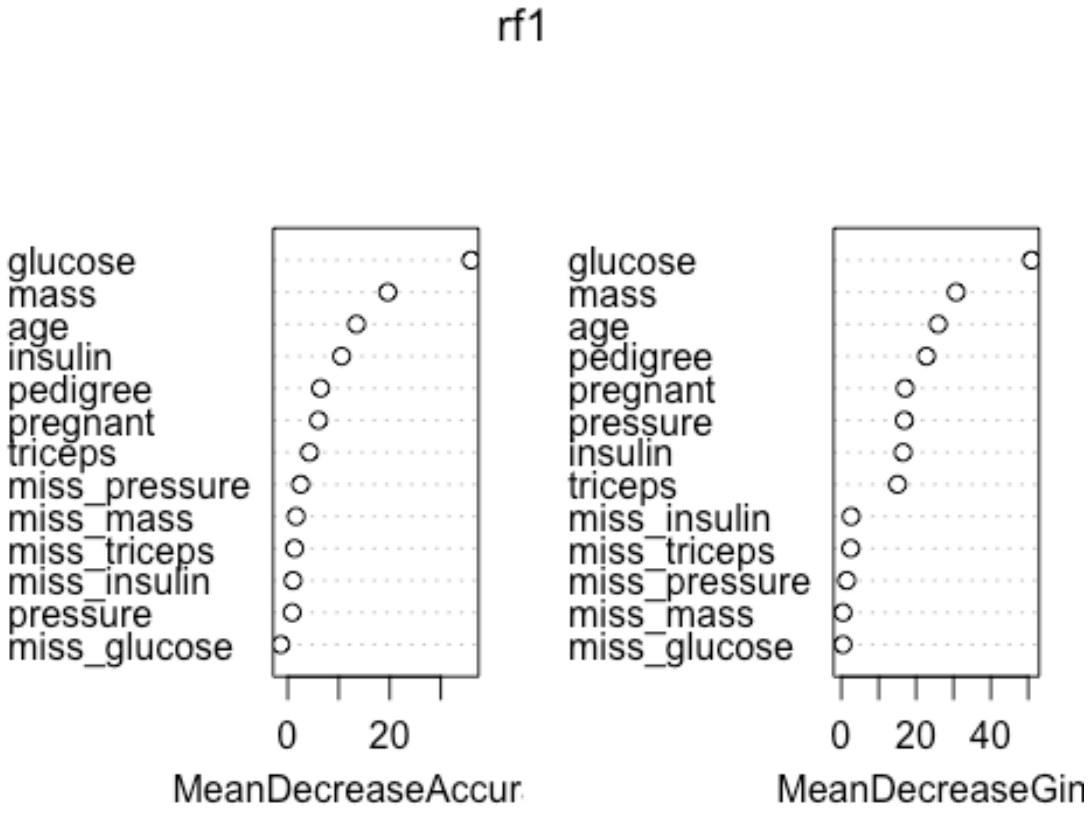
```
library(randomForest)

## randomForest 4.6-14

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

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin

set.seed(1)
rf1 = randomForest(as.factor(train_label) ~ .,
                   data = train_X,
                   # Number of trees
                   ntree = 500,
                   # Number of variables randomly sampled as candidates at e
ach split.
                   mtry = 2,
                   # We want the importance of predictors to be assessed.
                   importance = TRUE)

rf1

##
## Call:
##  randomForest(formula = as.factor(train_label) ~ ., data = train_X,      n
tree = 500, mtry = 2, importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 21.93%
## Confusion matrix:
##     0   1 class.error
```

```
## 0 307  45   0.1278409
## 1  73 113   0.3924731
```

The "OOB estimate of error rate" shows us how accurate our model is. $accuracy = 1 - errorrate$. OOB stands for "out of bag" - and bag is short for "bootstrap aggregation". So OOB estimates performance by comparing the predicted outcome value to the actual value across all trees using only the observations that were not part of the training data for that tree.

We can examine the relative variable importance in table and graph form. Random Forest estimates variable importance by separately examining each variable and estimating how much the model's accuracy drops when that variable's values are randomly shuffled (permuted). The shuffling temporarily removes any relationship between that covariate's value and the outcome. If a variable is important then the model's accuracy will suffer a large drop when it is randomly shuffled. But if the model's accuracy doesn't change it means the variable is not important to the model - e.g. maybe it was never even chosen as a split in any of the decision trees.

```
# As the function name suggests, this creates a variable importance plot
varImpPlot(rf1)
```




```
# Raw data
rf1$importance
```

```
##                              0            1 MeanDecreaseAccuracy
## pregnant       5.303587e-03 8.922439e-03         0.0065005595
## glucose        5.107832e-02 9.299729e-02         0.0653274569
## pressure       1.019476e-03 3.542268e-04         0.0007641140
## triceps        1.380128e-03 8.859413e-03         0.0039468980
## insulin        3.631436e-03 2.120840e-02         0.0095764460
## mass           1.556196e-02 4.204759e-02         0.0246714947
## pedigree       4.130405e-03 1.189099e-02         0.0067148458
## age            1.567527e-02 2.076999e-02         0.0174466247
## miss_glucose  -2.255136e-04 5.615425e-05        -0.0001328407
## miss_pressure  4.797760e-04 1.292979e-03         0.0007739085
## miss_triceps  -9.780655e-04 4.468804e-03         0.0008756241
## miss_insulin  -6.193218e-05 1.955534e-03         0.0006893311
## miss_mass      2.293553e-04 4.047019e-05         0.0001726647
##               MeanDecreaseGini
## pregnant             17.0687488
## glucose              50.8877107
## pressure             16.8652485
## triceps              15.0395991
## insulin              16.5157893
## mass                 30.7293787
## pedigree             22.7770302
## age                  25.8945064
## miss_glucose          0.4351143
## miss_pressure         1.4583062
## miss_triceps          2.5256183
## miss_insulin          2.6330872
## miss_mass             0.4391558
```

You can read up on the gini coefficient if interested. It's basically a measure of diversity or dispersion - a higher gini means the model is classifying better. The gini version does not randomly shuffle the variable cells.

Now, the goal is to see how the model performs on the test dataset:

```
rf_predicted = predict(rf1, newdata = test_X)
table(rf_predicted, test_label)
```

```
##             test_label
## rf_predicted   0   1
##            0 124  40
##            1  24  42
```

Check the accuracy of the test set:

```
mean(rf_predicted == test_label) # ~0.73
```

```
## [1] 0.7217391
```

```
# devtools::install_github("ck37/ck37r")
summary(ck37r::rf_count_terminal_nodes(rf1))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   37.00   72.00   82.00   80.28   89.00  115.00
```

How did it do? Are the accuracies for the training and test sets similar?

Big question 4: Why is the random forest algorithm preferred to a single decision tree or bagged trees?

Challenge 4

1. Try a few other values of mtry - can you find one that has improved performance?
2. Maxnodes is another tuning parameter for randomForest - does changing it improve your performance?
3. **(time permitting)** Use the iris dataset to perform classification on the "Species" variable. What are you noticing about model fits between the pidd and iris datasets?

5. Boosting

"Boosting is a general method for improving the accuracy of any given learning algorithm" and evolved from AdaBoost and PAC learning (p. 1-2). Gradient boosted machines are ensembles decision tree methods of "weak" trees that are just slightly more accurate than random guessing. These are then "boosted" into "strong" learners. That is, the models don't have to be accurate over the entire feature space.

The model first tries to predict each value in a dataset - the cases that can be predicted easily are *downweighted* so that the algorithm does not try as hard to predict them.

However, the cases that the model has difficulty predicting are *upweighted* so that the model more assertively tries to predict them. This continues for multiple "boosting iterations", with a training-based performance measure produced at each iteration. This method can drive down generalization error (p. 5). (from Freund Y, Schapire RE. 1999. A short introduction to boosting. Journal of Japanese Society for Artificial Intelligence 14:771-780.

Rather than testing only a single model at a time, it is useful to compare tuning parameters within that single model. Bootstrap is the default, but we want cross-validation. We also want to specify different tuning parameters.

First create two objects - `gbm_control` and `gbm_grid`. `gbm_control` will allow us to tune the cross-validated performance metric, while `gbm_grid` lets us evaluate the model with different characteristics:

```
# Choose 10-fold repeated measure cross-validation as our performance metric
# (instead of the default "bootstrap")
gbm_control = trainControl(method = "repeatedcv",
    repeats = 10,
    # Calculate class probabilities
    classProbs = TRUE,
    # Indicate that our response varaible is binary
```

```
    summaryFunction = twoClassSummary)


gbm_grid = expand.grid(
  # Number of trees to fit, aka boosting iterations
  n.trees = seq(1, 1500, by = 100),
  # Depth of the decision tree (how many splits)
    interaction.depth = c(1, 3, 5),
  # Learning rate: lower means the ensemble will adapt more slowly
    shrinkage = c(0.01, 0.05, 0.1),
  # Stop splitting a tree if we only have this many obs in a tree node
    n.minobsinnode = 10)
```

Fit the model. Note that we will now use area under the ROC curve (called "AUC") as our performance metric, which relates the number of true positives (sensitivity) to the number of true negatives (specificity).

NOTE: This will take a few minutes to complete! See the .HTML or .PDF file for the output.

```
library(caret)
library(pROC)

## Type 'citation("pROC")' for a citation.

##
## Attaching package: 'pROC'

## The following object is masked from 'package:gmodels':
##
##     ci

## The following objects are masked from 'package:stats':
##
##     cov, smooth, var

set.seed(1)
trainlab_factor = factor(ifelse(train_label == 1, "pos", "neg"))
testlab_factor = factor(ifelse(test_label == 1, "pos", "neg"))
table(trainlab_factor, train_label)

##                 train_label
## trainlab_factor   0   1
##             neg 352   0
##             pos   0 186

table(testlab_factor, test_label)

##                test_label
## testlab_factor   0   1
##            neg 148   0
##            pos   0  82
```

```r
# cbind: caret expects the Y response and X predictors to be part of the same
dataframe
gbm1 = train(trainlab_factor ~ ., data = cbind(trainlab_factor, train_X),
             # We want the method gradient boosted machine ("gbm")
             method = "gbm",
             # Use "AUC" as our performance metric, which caret incorrectly c
alls "ROC"
             metric = "ROC",
             # Specify our cross-validated performance metric settings
             trControl = gbm_control,
             # Define our gbm model tunings
             tuneGrid = gbm_grid,
             # Keep output hidden (setting to TRUE will print this output)
             verbose = F)

# See how long this algorithm took to complete
gbm1$times
```

```
## $everything
##    user  system elapsed
## 394.531   0.813 396.284
##
## $final
##    user  system elapsed
##   0.106   0.000   0.108
##
## $prediction
## [1] NA NA NA
```

```r
# Review model summary table
gbm1
```

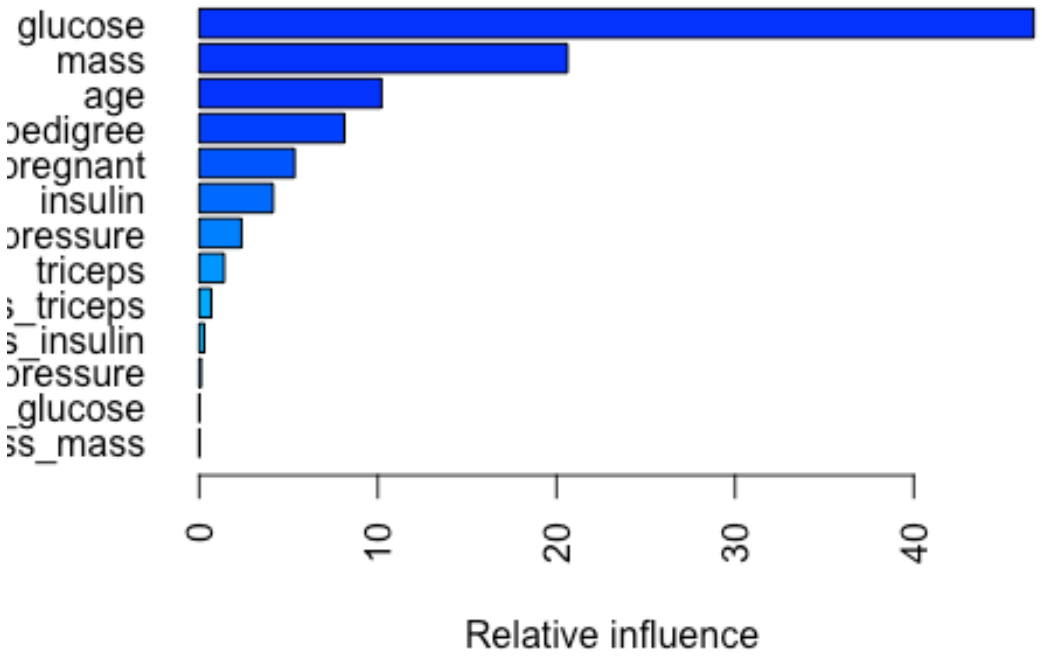
```
## Stochastic Gradient Boosting
##
## 538 samples
##  13 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 485, 484, 484, 483, 485, 484, ...
## Resampling results across tuning parameters:
##
##   shrinkage  interaction.depth  n.trees  ROC        Sens       Spec
##   0.01       1                    1      0.6910234  1.0000000  0.0000000
##   0.01       1                  101      0.8141540  0.9275873  0.4126316
##   0.01       1                  201      0.8265751  0.9113730  0.5001462
##   0.01       1                  301      0.8344289  0.9053730  0.5333626
##   0.01       1                  401      0.8392021  0.9020079  0.5591520
##   0.01       1                  501      0.8420353  0.8943333  0.5714620
```

```
##   0.01       1                  601     0.8438251  0.8903413  0.5796491
##   0.01       1                  701     0.8446268  0.8860714  0.5833626
##   0.01       1                  801     0.8454926  0.8838016  0.5807310
##   0.01       1                  901     0.8450706  0.8801190  0.5822515
##   0.01       1                 1001     0.8455533  0.8770000  0.5800877
##   0.01       1                 1101     0.8455015  0.8764603  0.5790058
##   0.01       1                 1201     0.8452869  0.8750635  0.5806140
##   0.01       1                 1301     0.8454775  0.8736349  0.5811696
##   0.01       1                 1401     0.8446595  0.8739127  0.5801462
##   0.01       3                    1     0.7658240  1.0000000  0.0000000
##   0.01       3                  101     0.8427068  0.9182381  0.4823684
##   0.01       3                  201     0.8473891  0.8886587  0.5640351
##   0.01       3                  301     0.8489809  0.8778571  0.5773392
##   0.01       3                  401     0.8487455  0.8673730  0.5918421
##   0.01       3                  501     0.8484154  0.8639603  0.6020760
##   0.01       3                  601     0.8461491  0.8608333  0.6011111
##   0.01       3                  701     0.8446087  0.8571667  0.6001170
##   0.01       3                  801     0.8431556  0.8529127  0.6044737
##   0.01       3                  901     0.8412398  0.8506270  0.6060526
##   0.01       3                 1001     0.8397626  0.8477937  0.6088012
##   0.01       3                 1101     0.8374436  0.8466746  0.6103216
##   0.01       3                 1201     0.8354813  0.8435397  0.6071930
##   0.01       3                 1301     0.8340519  0.8406984  0.6076316
##   0.01       3                 1401     0.8328112  0.8409841  0.6043275
##   0.01       5                    1     0.7570015  1.0000000  0.0000000
##   0.01       5                  101     0.8478162  0.9086190  0.5193860
##   0.01       5                  201     0.8497367  0.8793095  0.5860234
##   0.01       5                  301     0.8494096  0.8653968  0.6097368
##   0.01       5                  401     0.8471307  0.8580238  0.6177485
##   0.01       5                  501     0.8454829  0.8526190  0.6215497
##   0.01       5                  601     0.8420162  0.8460556  0.6151754
##   0.01       5                  701     0.8392534  0.8438095  0.6214912
##   0.01       5                  801     0.8360712  0.8395476  0.6161696
##   0.01       5                  901     0.8339937  0.8349921  0.6177193
##   0.01       5                 1001     0.8311194  0.8349683  0.6149415
##   0.01       5                 1101     0.8283181  0.8312619  0.6138596
##   0.01       5                 1201     0.8261425  0.8306984  0.6155263
##   0.01       5                 1301     0.8244431  0.8304048  0.6106725
##   0.01       5                 1401     0.8223289  0.8272619  0.6128655
##   0.05       1                    1     0.6971552  1.0000000  0.0000000
##   0.05       1                  101     0.8415592  0.8928889  0.5684211
##   0.05       1                  201     0.8445282  0.8784048  0.5769591
##   0.05       1                  301     0.8420119  0.8710397  0.5907602
##   0.05       1                  401     0.8415165  0.8684841  0.5913743
##   0.05       1                  501     0.8379691  0.8628095  0.5983918
##   0.05       1                  601     0.8351149  0.8602619  0.5962865
##   0.05       1                  701     0.8321879  0.8579603  0.5955848
##   0.05       1                  801     0.8299858  0.8559683  0.5893567
##   0.05       1                  901     0.8281597  0.8540000  0.5912573
##   0.05       1                 1001     0.8255876  0.8497302  0.5907018
```

```
##   0.05       1                  1101     0.8241446  0.8491349  0.5897368
##   0.05       1                  1201     0.8233256  0.8466032  0.5946491
##   0.05       1                  1301     0.8206682  0.8443175  0.5886550
##   0.05       1                  1401     0.8190378  0.8415317  0.5892690
##   0.05       3                     1     0.7605950  1.0000000  0.0000000
##   0.05       3                   101     0.8461536  0.8617143  0.5947368
##   0.05       3                   201     0.8366650  0.8509603  0.6059357
##   0.05       3                   301     0.8265584  0.8355635  0.6064620
##   0.05       3                   401     0.8210224  0.8281984  0.6055848
##   0.05       3                   501     0.8154838  0.8241746  0.6050000
##   0.05       3                   601     0.8112575  0.8204762  0.6049123
##   0.05       3                   701     0.8078383  0.8181984  0.6006140
##   0.05       3                   801     0.8064064  0.8159444  0.5940643
##   0.05       3                   901     0.8024680  0.8133651  0.6004678
##   0.05       3                  1001     0.8014744  0.8119444  0.5956725
##   0.05       3                  1101     0.7998821  0.8133492  0.5902632
##   0.05       3                  1201     0.7982461  0.8136508  0.5915205
##   0.05       3                  1301     0.7971761  0.8148016  0.5929825
##   0.05       3                  1401     0.7964035  0.8119603  0.5850000
##   0.05       5                     1     0.7644819  1.0000000  0.0000000
##   0.05       5                   101     0.8427054  0.8424127  0.6204678
##   0.05       5                   201     0.8279460  0.8324841  0.6090643
##   0.05       5                   301     0.8184642  0.8264444  0.6051754
##   0.05       5                   401     0.8121584  0.8213492  0.5966374
##   0.05       5                   501     0.8079199  0.8151032  0.5948538
##   0.05       5                   601     0.8046587  0.8150873  0.5907018
##   0.05       5                   701     0.8010289  0.8148175  0.5878947
##   0.05       5                   801     0.7990237  0.8119841  0.5873684
##   0.05       5                   901     0.7968377  0.8133889  0.5852339
##   0.05       5                  1001     0.7951769  0.8116984  0.5786842
##   0.05       5                  1101     0.7936225  0.8113968  0.5809649
##   0.05       5                  1201     0.7921254  0.8088333  0.5804971
##   0.05       5                  1301     0.7912529  0.8054286  0.5756140
##   0.05       5                  1401     0.7909409  0.8045635  0.5756433
##   0.10       1                     1     0.6829330  1.0000000  0.0000000
##   0.10       1                   101     0.8440118  0.8733175  0.5842982
##   0.10       1                   201     0.8387858  0.8653651  0.6016374
##   0.10       1                   301     0.8354025  0.8582540  0.6053509
##   0.10       1                   401     0.8288565  0.8551270  0.6046491
##   0.10       1                   501     0.8245268  0.8486111  0.6016374
##   0.10       1                   601     0.8213448  0.8448730  0.5928070
##   0.10       1                   701     0.8180249  0.8440873  0.5940058
##   0.10       1                   801     0.8162646  0.8406587  0.5928655
##   0.10       1                   901     0.8142852  0.8380794  0.5929240
##   0.10       1                  1001     0.8115095  0.8332460  0.5955556
##   0.10       1                  1101     0.8089632  0.8304206  0.5972515
##   0.10       1                  1201     0.8060828  0.8301111  0.5938596
##   0.10       1                  1301     0.8049358  0.8295476  0.5871637
##   0.10       1                  1401     0.8024305  0.8275317  0.5900877
##   0.10       3                     1     0.7593493  1.0000000  0.0000000
```

```
##   0.10       3                   101     0.8330548  0.8472381  0.6166082
##   0.10       3                   201     0.8189339  0.8350238  0.6037719
##   0.10       3                   301     0.8094620  0.8241825  0.5982164
##   0.10       3                   401     0.8021852  0.8159286  0.5970468
##   0.10       3                   501     0.7987521  0.8161984  0.5885088
##   0.10       3                   601     0.7957885  0.8102302  0.5869006
##   0.10       3                   701     0.7936706  0.8085397  0.5824269
##   0.10       3                   801     0.7902502  0.8056984  0.5792982
##   0.10       3                   901     0.7888260  0.8051667  0.5814912
##   0.10       3                  1001     0.7873603  0.8028810  0.5794152
##   0.10       3                  1101     0.7863389  0.8025873  0.5756433
##   0.10       3                  1201     0.7855761  0.8011667  0.5799123
##   0.10       3                  1301     0.7844486  0.8000317  0.5792690
##   0.10       3                  1401     0.7842370  0.8011825  0.5754678
##   0.10       5                     1     0.7704317  1.0000000  0.0000000
##   0.10       5                   101     0.8288583  0.8281825  0.6216082
##   0.10       5                   201     0.8104716  0.8202460  0.6057310
##   0.10       5                   301     0.8046545  0.8162302  0.5896199
##   0.10       5                   401     0.7993697  0.8128095  0.5869006
##   0.10       5                   501     0.7954564  0.8108571  0.5857895
##   0.10       5                   601     0.7930708  0.8094286  0.5836842
##   0.10       5                   701     0.7917251  0.8094286  0.5805263
##   0.10       5                   801     0.7896246  0.8080079  0.5868129
##   0.10       5                   901     0.7886445  0.8054127  0.5825731
##   0.10       5                  1001     0.7876360  0.8051270  0.5815497
##   0.10       5                  1101     0.7862813  0.8034444  0.5810819
##   0.10       5                  1201     0.7854635  0.8028730  0.5794444
##   0.10       5                  1301     0.7847371  0.7994444  0.5788304
##   0.10       5                  1401     0.7853493  0.8020000  0.5777485
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were n.trees = 201,
##  interaction.depth = 5, shrinkage = 0.01 and n.minobsinnode = 10.
```

```r
# Plot variable importance
summary(gbm1, las = 2)
```

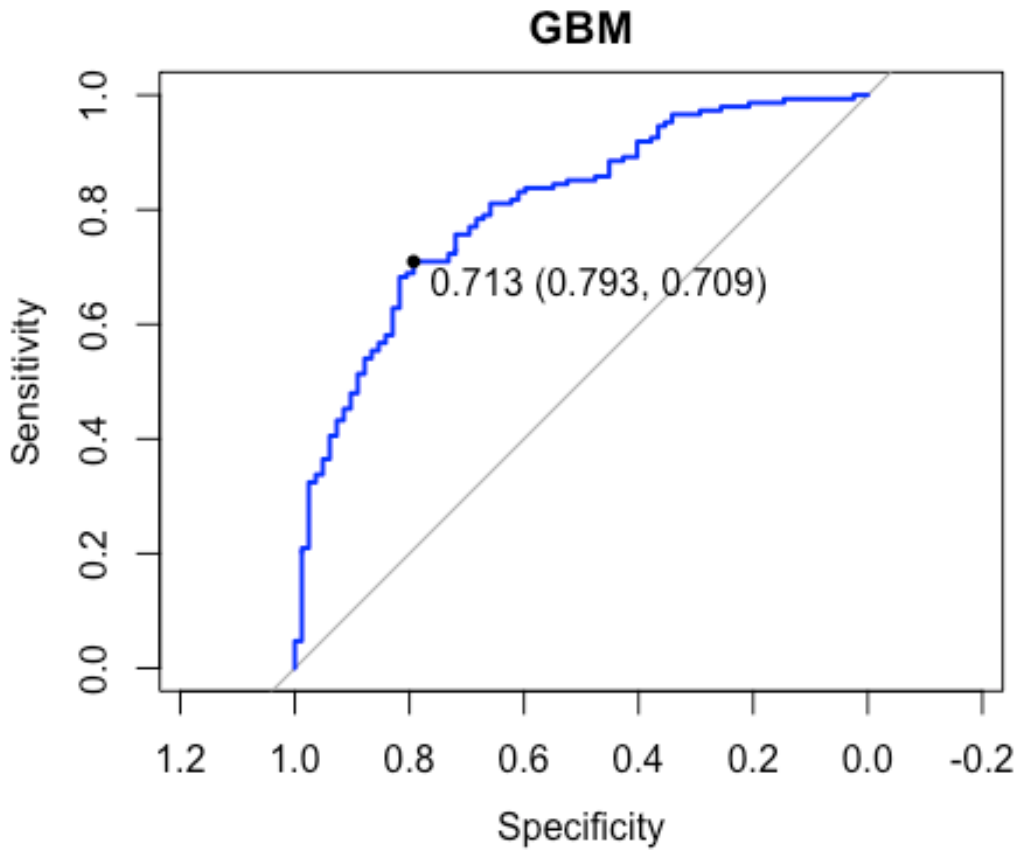



```
##                          var    rel.inf
## glucose              glucose 46.7202788
## mass                    mass 20.6182606
## age                      age 10.2067669
## pedigree           pedigree  8.1298600
## pregnant           pregnant  5.3413151
## insulin             insulin  4.1410929
## pressure           pressure  2.3692215
## triceps             triceps  1.4021933
## miss_triceps   miss_triceps  0.6789003
## miss_insulin   miss_insulin  0.2873914
## miss_pressure miss_pressure  0.1047192
## miss_glucose   miss_glucose  0.0000000
## miss_mass         miss_mass  0.0000000

# Generate predicted values
gbm_predicted = predict(gbm1, test_X)

# Generate class probabilities
gbm_probs = predict(gbm1, test_X, type = "prob")

# View final model
```

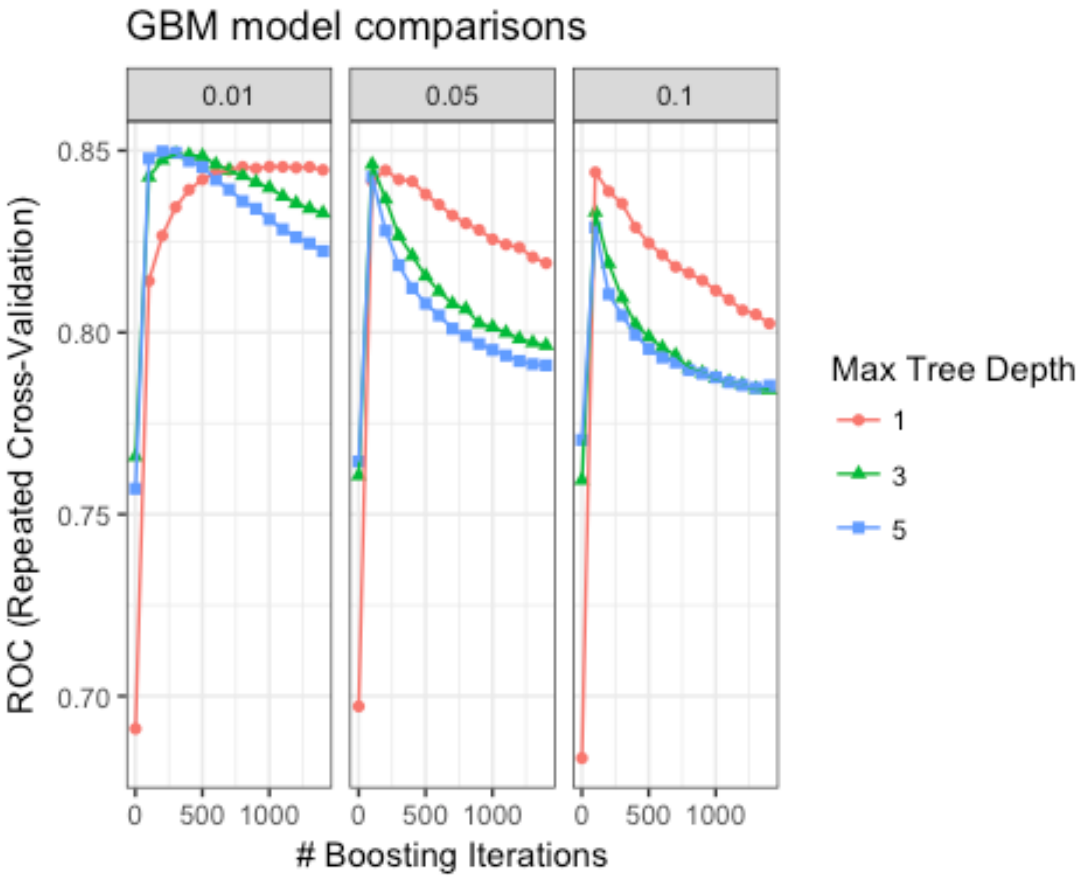
```r
gbm_cm = confusionMatrix(gbm_predicted, testlab_factor)
gbm_cm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg 125  39
##        pos  23  43
##
##                Accuracy : 0.7304
##                  95% CI : (0.6682, 0.7866)
##     No Information Rate : 0.6435
##     P-Value [Acc > NIR] : 0.003125
##
##                   Kappa : 0.3858
##  Mcnemar's Test P-Value : 0.056780
##
##             Sensitivity : 0.8446
##             Specificity : 0.5244
##          Pos Pred Value : 0.7622
##          Neg Pred Value : 0.6515
##              Prevalence : 0.6435
##          Detection Rate : 0.5435
##    Detection Prevalence : 0.7130
##       Balanced Accuracy : 0.6845
##
##        'Positive' Class : neg
##
```

```r
# Define ROC characteristics
rocCurve = roc(response = testlab_factor,
    predictor = gbm_probs[, "neg"],
    levels = rev(levels(testlab_factor)),
    auc=TRUE, ci=TRUE)

# Plot AUC
plot(rocCurve, print.thres = "best", main = "GBM", col = "blue")
```




```
# ggsave("gbm AUC.png")

# Plot the cross-validated AUC of the different configurations
ggplot(gbm1) + theme_bw() + ggtitle("GBM model comparisons")
```




```
# ggsave("gbm tuning comparison.png")
```

Also check out the "xgboost" R package for a more powerful way to boost your trees.

Challenge 5

Big question 5: What are some defining characteristics of the algorithms we have covered in these five exercises?

6. Ensemble methods

You have learned how to fit several single algorithms and have explored a little about how you can define their different (hyper)parameters. However, the "SuperLearner" R package is a method that simplifies ensemble learning by allowing you to simultaneously evaluate the cross-validated performance of multiple algorithms and/or a single algorithm with differently tuned hyperparameters.

Let's see how the four classification algorithms you learned in this workshop (KNN, decision tree, random forest, and gradient boosted machines) compare to each other and also to binary logistic regression (`glm`) and to the mean of Y as a benchmark algorithm, in terms of their cross-validated error!

A “wrapper” is a short function that adapts an algorithm for the SuperLearner package. Check out the different algorithm wrappers offered by SuperLearner:

```
library(SuperLearner)

## Loading required package: nnls

## Super Learner

## Version: 2.0-23

## Package created on 2018-03-09

listWrappers()

## All prediction algorithm wrappers in SuperLearner:

##  [1] "SL.bartMachine"      "SL.bayesglm"         "SL.biglasso"
##  [4] "SL.caret"            "SL.caret.rpart"      "SL.cforest"
##  [7] "SL.dbarts"           "SL.earth"            "SL.extraTrees"
## [10] "SL.gam"              "SL.gbm"              "SL.glm"
## [13] "SL.glm.interaction"  "SL.glmnet"           "SL.ipredbagg"
## [16] "SL.kernelKnn"        "SL.knn"              "SL.ksvm"
## [19] "SL.lda"              "SL.leekasso"         "SL.lm"
## [22] "SL.loess"            "SL.logreg"           "SL.mean"
## [25] "SL.nnet"             "SL.nnls"             "SL.polymars"
## [28] "SL.qda"              "SL.randomForest"     "SL.ranger"
## [31] "SL.ridge"            "SL.rpart"            "SL.rpartPrune"
## [34] "SL.speedglm"         "SL.speedlm"          "SL.step"
## [37] "SL.step.forward"     "SL.step.interaction" "SL.stepAIC"
## [40] "SL.svm"              "SL.template"         "SL.xgboost"

##
## All screening algorithm wrappers in SuperLearner:

## [1] "All"
## [1] "screen.corP"           "screen.corRank"        "screen.glmnet"
## [4] "screen.randomForest"   "screen.SIS"            "screen.template"
## [7] "screen.ttest"          "write.screen.template"
```

Instead of splitting the data like before, since we are using cross-validation we actually want to use the entire pidd dataset - cross-validation will perform as many training and test splits as necessary (this is called the number of “folds”) for us!

```
str(pidd)

## 'data.frame':    768 obs. of  14 variables:
##  $ pregnant     : num  6 1 8 1 0 5 3 10 2 8 ...
##  $ glucose      : num  148 85 183 89 137 116 78 115 197 125 ...
##  $ pressure     : num  72 66 64 66 40 74 50 72 70 96 ...
##  $ triceps      : num  35 29 29 23 35 29 32 29 45 29 ...
##  $ insulin      : num  125 125 125 94 168 125 88 125 543 125 ...
```

```
##  $ mass         : num  33.6 26.6 23.3 28.1 43.1 25.6 31 35.3 30.5 32.3 ...
##  $ pedigree     : num  0.627 0.351 0.672 0.167 2.288 ...
##  $ age          : num  50 31 32 21 33 30 26 29 53 54 ...
##  $ diabetes     : Factor w/ 2 levels "neg","pos": 2 1 2 1 2 1 2 1 2 2 ...
##  $ miss_glucose : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ miss_pressure: num  0 0 0 0 0 0 0 1 0 0 ...
##  $ miss_triceps : num  0 0 1 0 0 1 0 1 0 1 ...
##  $ miss_insulin : num  1 1 1 0 0 1 0 1 0 1 ...
##  $ miss_mass    : num  0 0 0 0 0 0 0 0 0 1 ...

# convert our Y variable to numeric type
Y_sl = ifelse(pidd$diabetes == "pos", 1, 0)

# Now, tell SuperLearner which algorithms to incorporate
SL_library = c("SL.mean", "SL.knn", "SL.glm", "SL.rpart", "SL.randomForest",
"SL.gbm")
```

Fit the ensemble:

```
library(SuperLearner)
# This is a seed that is compatible with multicore parallel processing.
# See ?set.seed for more information.
set.seed(1, "L'Ecuyer-CMRG")

# This may take a minute to execute.
cv_sl = CV.SuperLearner(Y = Y_sl, X = subset(pidd, select = -diabetes),
                        SL.library = SL_library,
                        family = binomial(),
                        cvControl = list(V = 5),
                        # Set to T to show details
                        verbose = F)

## Loading required package: gbm

## Loading required package: survival

##
## Attaching package: 'survival'

## The following object is masked from 'package:caret':
##
##     cluster

## Loading required package: splines

## Loading required package: parallel

## Loaded gbm 2.1.3

cv_sl
```

```
##
## Call:
## CV.SuperLearner(Y = Y_sl, X = subset(pidd, select = -diabetes), family = b
inomial(),
##     SL.library = SL_library, verbose = F, cvControl = list(V = 5))
##
##
## Cross-validated predictions from the SuperLearner:  SL.predict
##
## Cross-validated predictions from the discrete super learner (cross-validat
ion selector):  discreteSL.predict
##
## Which library algorithm was the discrete super learner:  whichDiscreteSL
##
## Cross-validated prediction for all algorithms in the library:  library.pre
dict
```

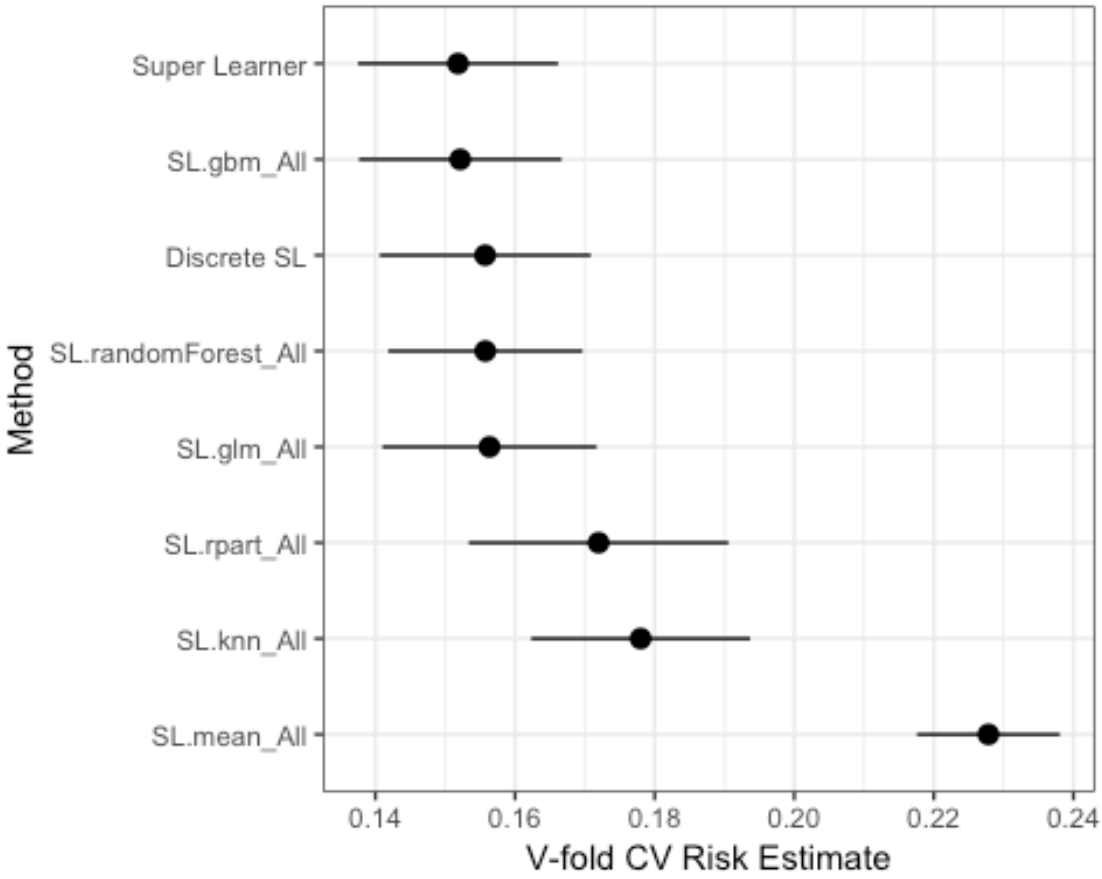
NOTE: Again, this will take a few minutes to complete! See the .HTML or .PDF file for the output!

Risk is a performance estimate - it's the average loss, and loss is how far off the prediction was for an individual observation. The lower the risk, the fewer errors the model makes in its prediction. SuperLearner's default loss metric is squared error $(y_{actual} - y_{predicted})^2$, so the risk is the mean-squared error (just like in ordinary least *squares* regression). View and plot results:

```
summary(cv_sl)
```

```
##
## Call:
## CV.SuperLearner(Y = Y_sl, X = subset(pidd, select = -diabetes), family = b
inomial(),
##     SL.library = SL_library, verbose = F, cvControl = list(V = 5))
##
## Risk is based on: Mean Squared Error
##
## All risk estimates are based on V =  5
##
##            Algorithm     Ave        se     Min     Max
##        Super Learner 0.15183 0.0073169 0.13004 0.17158
##          Discrete SL 0.15572 0.0077175 0.13114 0.17342
##          SL.mean_All 0.22782 0.0052203 0.21390 0.24040
##           SL.knn_All 0.17799 0.0080058 0.16320 0.19656
##           SL.glm_All 0.15634 0.0078346 0.13946 0.17435
##         SL.rpart_All 0.17197 0.0094943 0.13712 0.18561
##  SL.randomForest_All 0.15574 0.0070992 0.13689 0.17744
##           SL.gbm_All 0.15214 0.0074113 0.13114 0.17342
```

```
plot(cv_sl) + theme_bw()
```




```
# ggsave("SuperLearner.pdf")
```

"Discrete SL" is when the SuperLearner chooses the single algorithm with the lowest risk. "SuperLearner" is a weighted average of multiple algorithms, or an "ensemble". In theory the weighted-average should have a little better performance, although they often tie. In this case we only have a few algorithms so the difference is minor.

Big question 6: Why do you want to consider ensemble methods for your machine learning projects instead of a single algorithm?

Challenge 6

1. What are the elements of the CV_SL object? Take a look at 1 or 2 of them. Hint: use the `names()` function to list the elements of an object, then `$` to access them (just like a dataframe).

A longer tutorial on SuperLearner is available here:
https://github.com/ck37/superlearner-guide