Predictive Modeling

Steve Pittard

2020-02-26

Contents

Chapter 1

Introduction

Predictive Modeling is a type of Machine Learning which itself is a sub branch of Artificial Intelligence. The following graphic provides us with some history of these domains. This is helpful if you are trying to orient yourself in the world of analytics and machine learning. Note that AI has been around for quite some time. The Wikipedia definition of AI is:

> The study of "intelligent agents": any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals

1.1 Machine Learning

Machine Learning relies upon "patterns and inference"" to "perform a specific task without using explicit instructions". It is a form of Applied AI that attempts to automatically learn from experience without being explicitly programmed. Think of Predictive Modeling as a subset of this which falls into two categories:

Supervised

Algorithms that build a model on a set of data containing both the inputs and the desired outputs ("labels" or known numeric values). When you want to map input to known output labels. Build a model that, when applied to "new" data, will hopefully predict the correct label.

Some common techniques for Supervised learning include: Generalized Linear Models (GLM), Logistic Regression, Random Forests, Decision Trees, Neural Networks, Multivariate Adaptive Regression Splines (MARS), and K Nearest Neighbors.

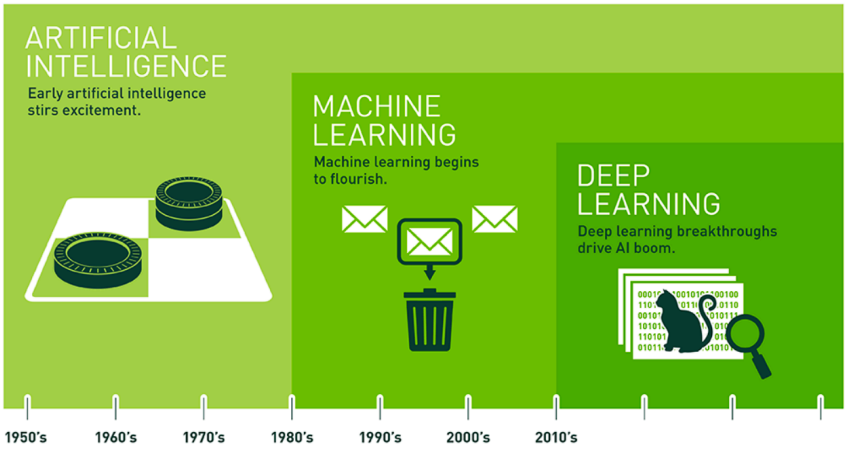


Figure 1.1:

Figure 1.2:

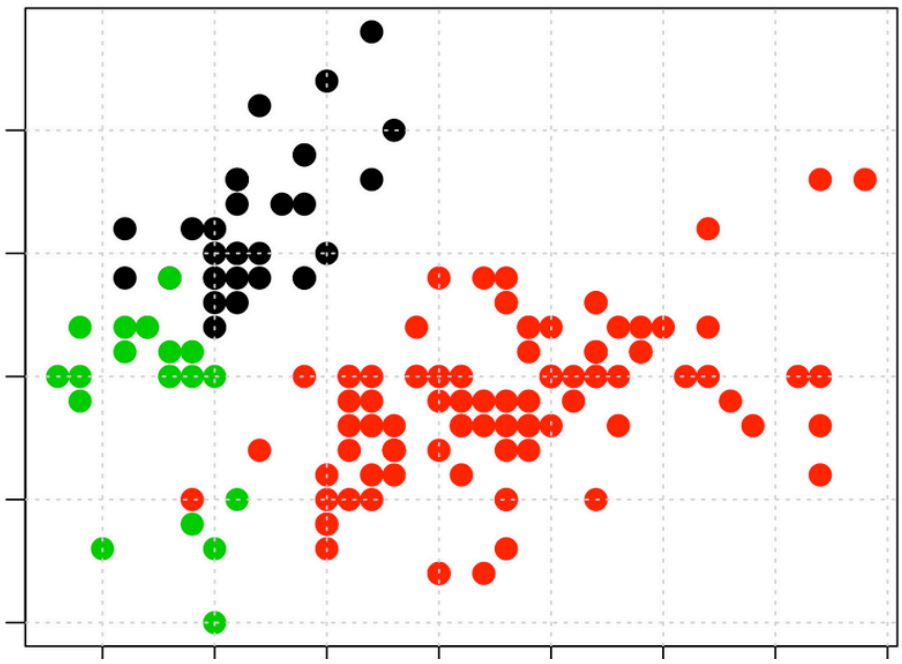


Figure 1.3:

Unsupervised

Algorithms that take a set of data that contains only inputs, and find structure in the data (e.g. clustering of data points)

Some common techniques for unsupervised learning include: hierarchical clustering, k-means clustering, mixture models, DBSCAN, Association Rules, Neural Networks

1.2 Predictive Modeling

This lecture is concerned primarily with Predictive Modeling. Some examples of Predictive Modeling include:

- Predict current CD4 cell count of an HIV-positive patient using genome sequences
- Predict Success of Grant Applications
- Use attributes of chemical compounds to predict likelihood of hepatic injury

- How many copies of a new book will sell ?
- Will a customer change Internet Service Providers ?

In this domain there are generally two types of predictive models.

1) Classification for Predicting Qualitative Outcomes:

This relates to situations such as whether someone has a disease ("positive") or not ("negative"). The problem could also be multi classification such as assigning an organism to one of a number of possible species.

2) Regression for Quantitative Out Comes

This is when we wish to predict a numeric / continuous outcome such as a final sales price for a house or car. It might also be a prediction of tomorrow's stock or Bit Coin price.

1.3 In-Sample vs Out-Of-Sample Data

The goal of predictive model is to generate models that can generalize to new data. It would be good if any model we generate could provide a good estimate of out of sample error. It's easy to generate a model on an entire data set (in sample data) and then turn around and use that data for prediction. But how will it perform on new data ? Haven't we just over trained our model ?

1.4 Performance Metrics

For either case (regression vs classification) we need some type of metric or measure to let us know how well a given model will work on new or unseen data - also known as "out of sample" data. for Classification problems we look at things like "sensitivity", "specificity", "accuracy", and "Area Under Curve". For Quantitative outcomes, we look at things like Root Mean Square Error (RMSE) or Mean Absolute Error (MAE). The selection of metric will frequently depend on your domain of interest. We'll use a couple of different methods.

1.5 Black Box

The good news is that you can treat building predictive models as a "Black Box". The bad news is that you can treat building predictive models as a "Black Box".

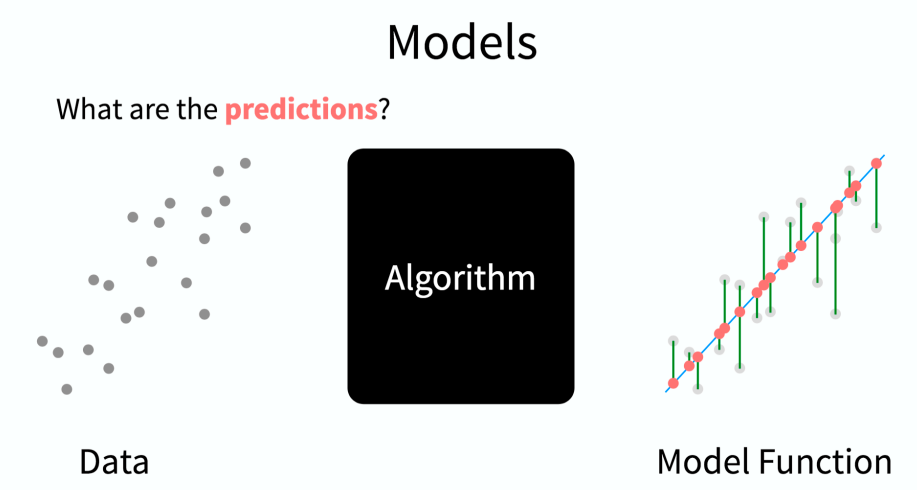

Models
What are the predictions?
Algorithm
Data
Model Function

Figure 1.4:

Chapter 2

Getting Hands On

```
library(tidyverse)
library(mlbench)
library(ROCR)
library(DataExplorer)
library(caret)
```

Let's consider the Pima Indians Data that is part of the **mlbench** package. You can install this package via the **Tools -> Install Package** menu item within R Studio or type the following at the R console prompt:

```
install.packages("mlbench")
```

Once you have it installed then load it into the work space as follows:

```
data("PimaIndiansDiabetes")

# Get a shorter handle. I hate typing.
pm <- PimaIndiansDiabetes
```

The description of the data set is as follows:

So we now have some data on which we can build a model. Specifically, there is a variable in the data called "diabetes" which indicates the disease / diabetes status ("pos" or "neg") of the person. It would be good to come up with a model that we could use with incoming data to determine if someone has diabetes.

2.1 Important Terminology

In predictive modeling there are some common terms to consider:

Format

pregnant Number of times pregnant
glucose Plasma glucose concentration (glucose tolerance test)
pressure Diastolic blood pressure (mm Hg)
triceps Triceps skin fold thickness (mm)
insulin 2-Hour serum insulin (mu U/ml)
mass Body mass index (weight in kg/(height in m)\^2)
pedigree Diabetes pedigree function
age Age (years)
diabetes Class variable (test for diabetes)

Figure 2.1:

ID	PREGNANT	GLUCOSE	INSULIN	...	DIABETES
001	6	148	178	...	POS
002	3	133	88	...	NEG
...	...	...	...	...	....
767	1	137	192	...	POS
768	0	116	173	...	NEG

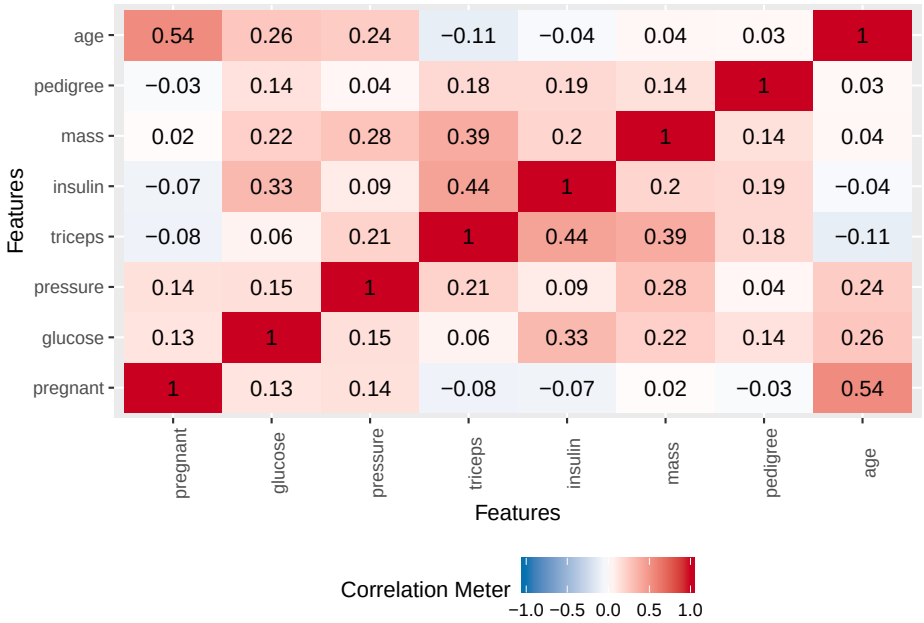
Each row is an "Observation"
Output - Labelled Data / Dependent Variable
Input Columns Are - Features / Variables / Attributes / Predictors

Figure 2.2:

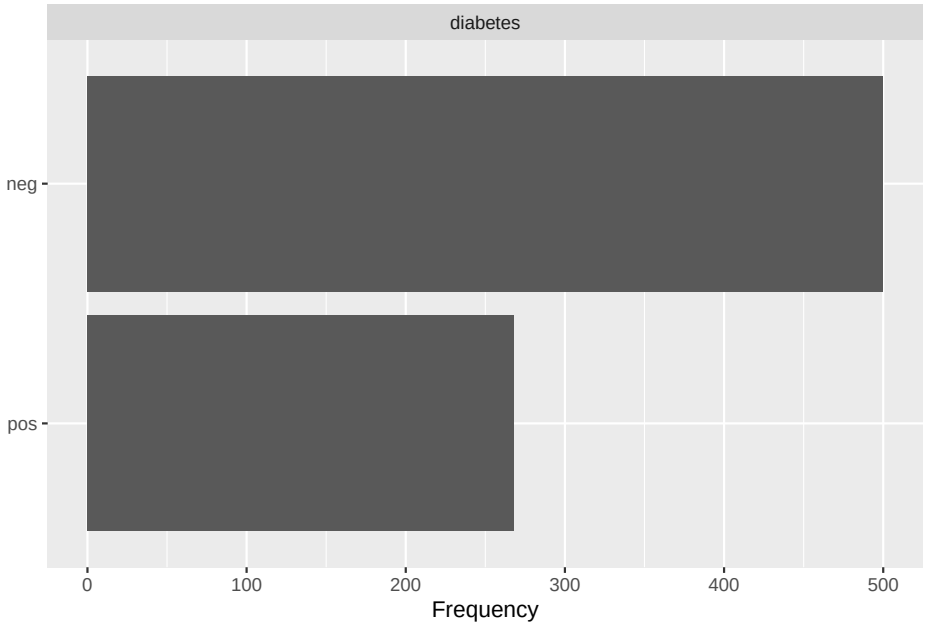
2.2 Exploratory Plots

We'll look use some stock plots from the **DataExplorer** package to get a feel for the data. Look at correlations between the variables to see if any are strongly correlated with the variable we wish to predict or any other variables.

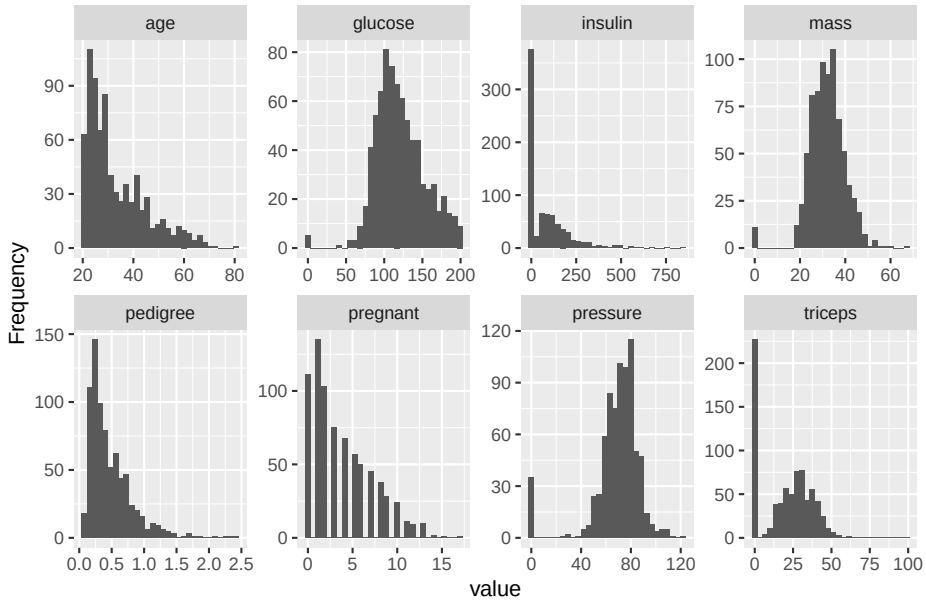
```
plot_correlation(pm, type="continuous")
```



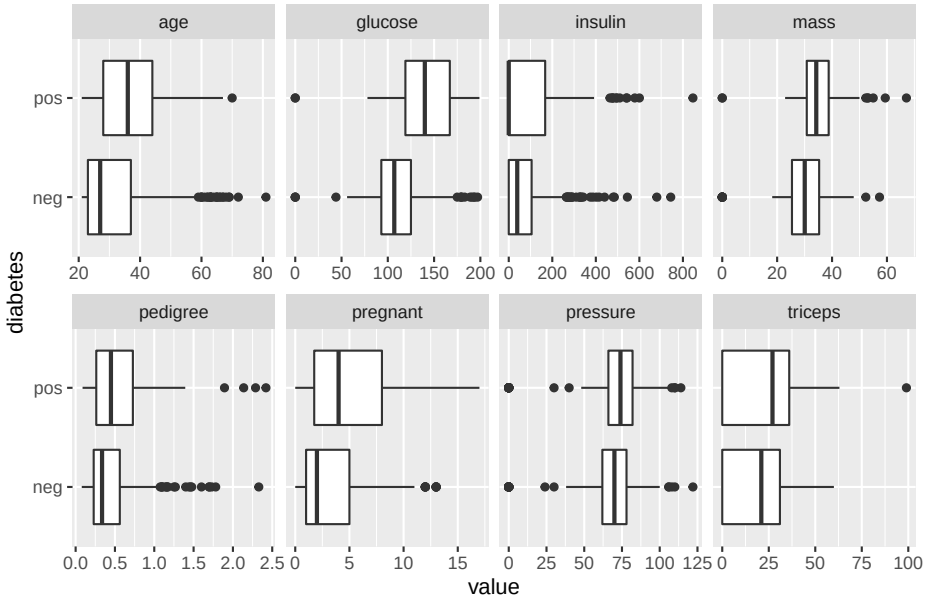

```
plot_bar(pm)
```




```
plot_histogram(pm)
```




```
plot_boxplot(pm,by="diabetes")
```


age
glucose
insulin
mass
pedigree
pregnant
pressure
triceps
pos
neg
diabetes
value
20 40 60 80
0 50 100 150 200
0 200 400 600 800
0 20 40 60
0.0 0.5 1.0 1.5 2.0 2.5
0 5 10 15
0 25 50 75 100 125
0 25 50 75 100

Chapter 3

A Common Modeling Workflow

The following graphic depicts the steps common to the Modeling Process. This is not the only way to proceed but it provides a very helpful schematic by which to plan your work.

3.1 Splitting The Data

A fundamental approach used in ML is to segment data into a "training" set which is some percentage of the original data - say 80%. The remaining 20% would be assigned to a "test" data set. Then we build a model on our training data set after which we use that model to predict outcomes for the test data set. This looks like the following.

Note that some scenarios will split the data into three data sets: 1) training, 2) validation, and 3) test. This scenario is used when tuning so called hyper parameters for methods that have "tuning" parameters that could influence the resulting model. We'll stick with the basic "train / test" approach for now.

Splitting the data is not particularly challenging. We can use the built in **sample** function in R to do this. We aren't sampling with replacement here which guarantees that no record can exist in both sets. That is, if a record from the data set is assigned to the training set, it will not be in the test data set.

```
# Make this example reproducible
set.seed(123)
percent <- .80

# Get the indices for a training set.
```

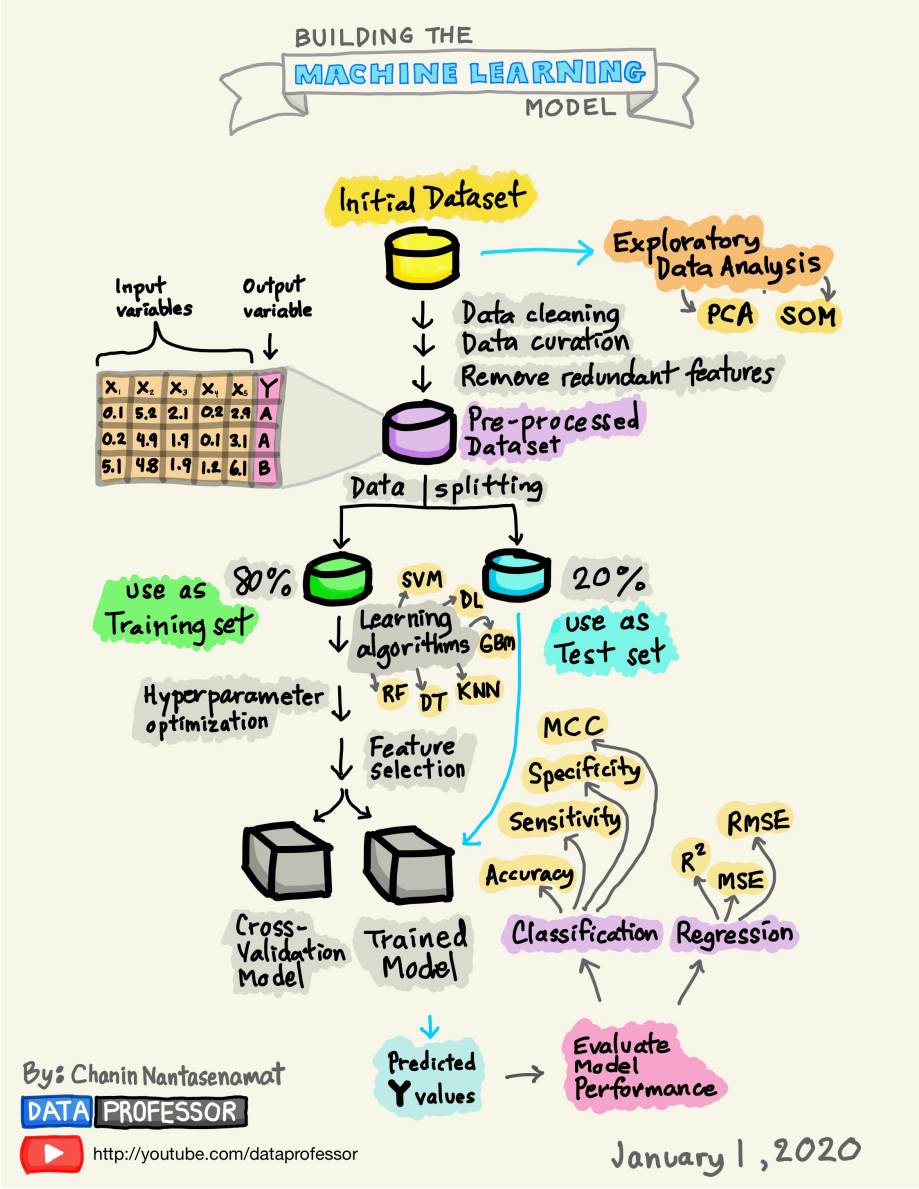

BUILDING THE
MACHINE LEARNING
MODEL
Initial Dataset
Exploratory Data Analysis
PCA
SOM
Input variables
Output variable
Data cleaning
Data curation
Remove redundant features
Pre-processed Data set
Data splitting
80%
20%
Use as Training set
Use as Test set
SVM
DL
Learning algorithms
GBM
RF
DT
KNN
Hyperparameter optimization
Feature selection
MCC
Specificity
Sensitivity
Accuracy
RMSE
R²
MSE
Classification
Regression
Cross-Validation Model
Trained Model
Predicted Y values
Evaluate Model Performance
By: Chanin Nantasenamat
DATA PROFESSOR
http://youtube.com/dataprofessor
January 1, 2020

Figure 3.1:

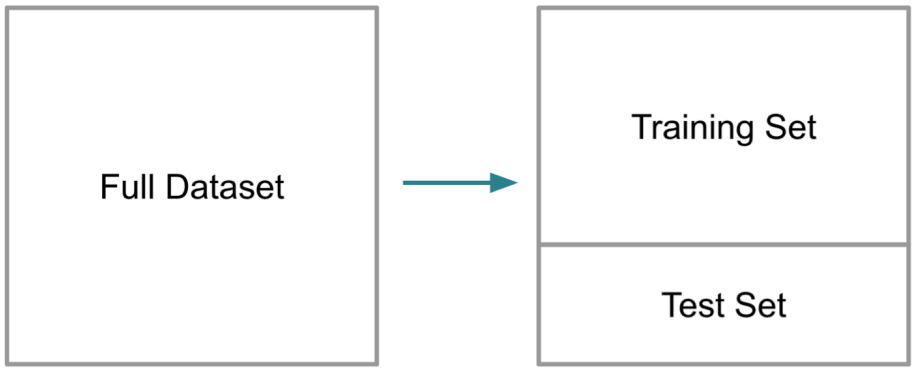


Figure 3.2:

```
idx <- sample(1:nrow(pm),round(.8*nrow(pm)),F)

# Use bracket notation to create the train / test pair
train <- pm[idx,]
test  <- pm[-idx,]

# The following should have 80 percent of the original
# data

round(nrow(train)/nrow(pm)*100)
```

```
## [1] 80
```

3.2 First Model

Now let's build a Generalized Linear Model to do the prediction. We will employ logistic regression.

```
myglm <- glm(diabetes ~ .,
             data = train,
             family = "binomial")

summary(myglm)
```

```
##
## Call:
## glm(formula = diabetes ~ ., family = "binomial", data = train)
##
```

```
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.3941  -0.7235  -0.4285   0.7476   3.0031
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -8.2308564  0.7816436 -10.530  < 2e-16 ***
## pregnant     0.1138202  0.0366475   3.106  0.00190 **
## glucose      0.0366854  0.0041947   8.746  < 2e-16 ***
## pressure    -0.0131360  0.0059415  -2.211  0.02704 *
## triceps     -0.0006303  0.0075466  -0.084  0.93343
## insulin     -0.0017394  0.0009826  -1.770  0.07667 .
## mass         0.0847273  0.0161080   5.260 1.44e-07 ***
## pedigree     0.9057850  0.3329203   2.721  0.00651 **
## age          0.0120925  0.0107367   1.126  0.26005
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 790.13  on 613  degrees of freedom
## Residual deviance: 581.40  on 605  degrees of freedom
## AIC: 599.4
##
## Number of Fisher Scoring iterations: 5
```

In looking at the output we see some problems such as a number of predictors aren't significant so maybe we should eliminate them from the model. For now, we'll keep going because we are trying to outline the larger process / workflow.

3.3 First Prediction

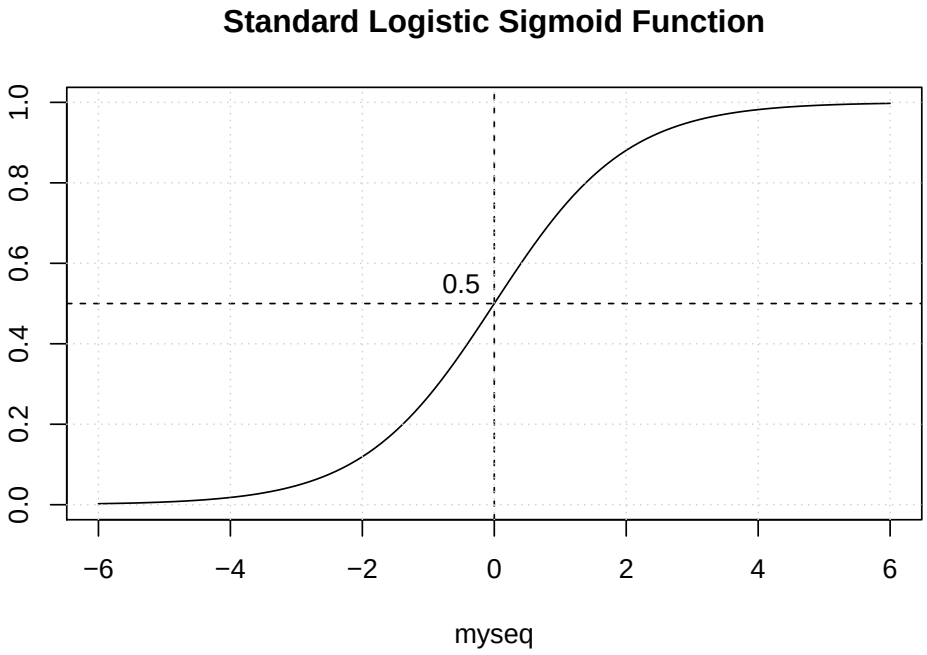
We could now use this new model to predict outcomes using the test data set. Remember that we are attempting to predict a binary outcome - in this case whether the person is positive for diabetes or negative.

What we get back from the prediction object are probabilities for which we have to determine a threshold above which we would say the observation is "positive" for diabetes and, below the threshold, "negative".

```
probs <- predict(myglm,
                 newdata = test,
                 type = "response")

probs[1:10]
```

```
##         2         3         9        12        13        17        18
## 0.0503311 0.8208652 0.6680994 0.9016430 0.7766679 0.3361188 0.2029466
##        23        25        31
## 0.9453408 0.6693923 0.4026717
```

With logistic regression we are dealing with a curve like the one below which is a sigmoid function. The idea is to take our probabilities, which range between 0 and 1, and then pick a threshold over which we would classify that person as being positive for diabetes.

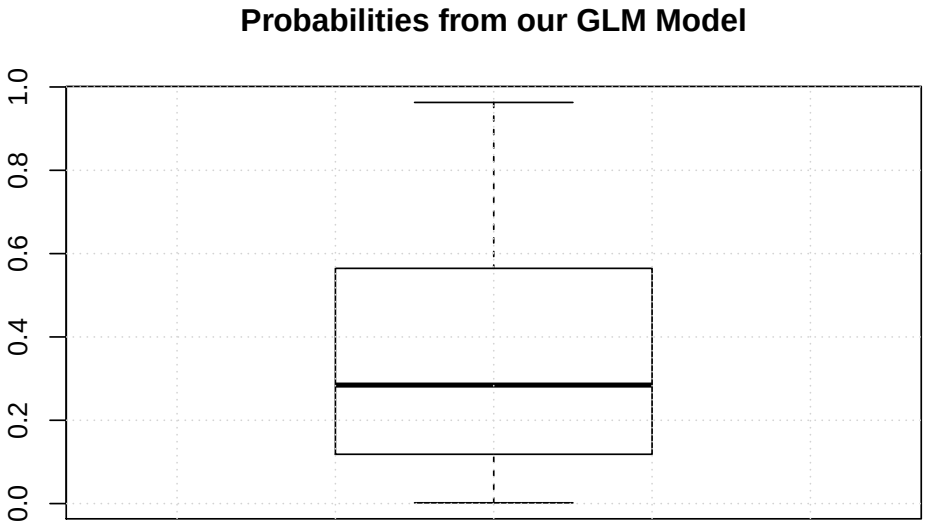


3.3.1 Selecting The Correct Alpha

The temptation is to select 0.5 as the threshold such that if a returned probability exceeds 0.5 then we classify the associated subject as being "positive" for the disease. But then this assumes that the probabilities are distributed accordingly. This is frequently not the case though it doesn't stop people from using 0.5.

We might first wish to look at the distribution of the returned probabilities before making a decision about where to set the threshold. We can see clearly that selecting 0.5 in this case would not be appropriate.

```
boxplot(probs,
        main="Probabilities from our GLM Model")
grid()
```



The median is somewhere around .25 so we could use that for now although we are just guessing.

```
mypreds <- ifelse(probs > 0.25,"pos","neg")
mypreds <- factor(mypreds, levels = levels(test[["diabetes"]]))
mypreds[1:10]
```

```
##   2   3   9  12  13  17  18  23  25  31
## neg pos pos pos pos pos neg pos pos pos
## Levels: neg pos
```

3.3.2 Confusion Matrices

Next, we would compare our predictions against the known outcomes which are stored in the test data frame:

```
# How does this compare to the truth ?
table(predicted = mypreds,
      actual = test$diabetes)
```

```
##          actual
## predicted neg pos
##       neg  60   7
##       pos  37  50
```

What we are doing is building a "Confusion Matrix" which can help us determine how effective our model is. From such a matrix table we can compute a number of "performance measures", such as accuracy, precision, sensitivity, specificity and others, to help assess the quality of the model. In predictive modeling we are always interested in how well any given model will perform on "new" data.

There are some functions that can help us compute a confusion matrix. Because the variable we are trying to predict, (diabetes), is a two level factor, ("neg" or "pos") we'll need to turn our predictions into a comparable factor. Right now, it's just a character string.

```
# test$diabetes <- ordered(test$diabetes,c("pos","neg"))

mypreds <- factor(mypreds,
                  levels=levels(test$diabetes))

caret::confusionMatrix(mypreds,test$diabetes,positive="pos")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg  60   7
##        pos  37  50
##
##                Accuracy : 0.7143
##                  95% CI : (0.636, 0.7841)
##     No Information Rate : 0.6299
##     P-Value [Acc > NIR] : 0.01718
##
##                   Kappa : 0.4472
##
##  Mcnemar's Test P-Value : 1.232e-05
##
##             Sensitivity : 0.8772
##             Specificity : 0.6186
##          Pos Pred Value : 0.5747
##          Neg Pred Value : 0.8955
##              Prevalence : 0.3701
##          Detection Rate : 0.3247
##    Detection Prevalence : 0.5649
##       Balanced Accuracy : 0.7479
##
##        'Positive' Class : pos
##
```

3.4 Performance Measures Revisited

This is helpful stuff although there are a number of measures to select as a primary performance metric. Ideally, we would already know which performance metric we would select to effectively "judge" the quality of our model. In medical

tests, "sensitivity" and "specificity" are commonly used. Some applications use "Accuracy" (which isn't good when there is large group imbalance). Anyway, if, for example, we pick "sensitivity" as a judge of model quality we see that is somewhere around .87. (A much deeper discussion about selecting the best performance measure is in order but we'll keep moving for now)

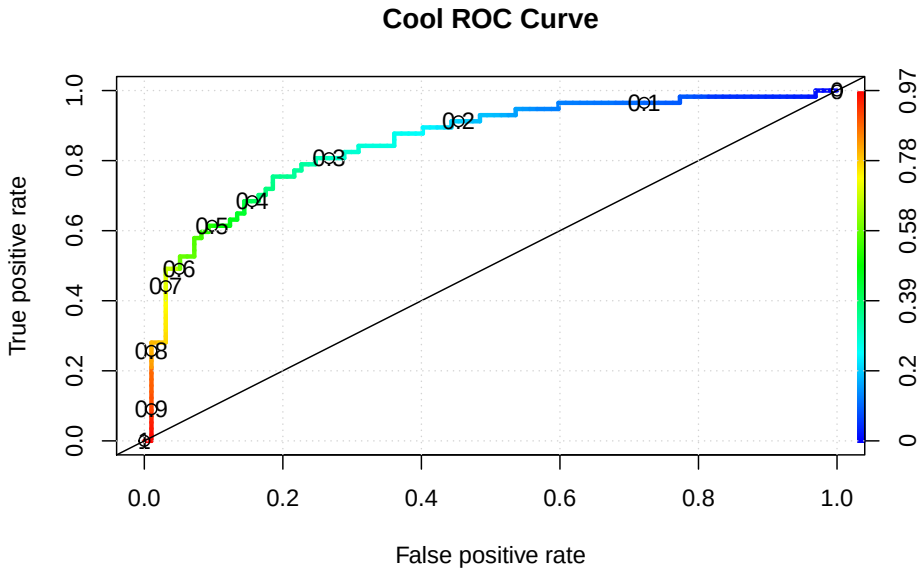
The problem here is that all we have done is looked at the confusion matrix corresponding to one specific (and arbitrary) threshold value when what we need is to look at a number of confusion matrices corresponding to many different thresholds. For example, we might get a better sensitivity level had we selected the mean of the returned probabilities. This process could go on and on and on… So we would benefit from a rigorous approach to find the "best" threshold.

3.5 The ROC curve

One way to do this is to use something known as the ROC curve. Luckily, R has functions to do this. This isn't surprising as it is a standard tool that has been in use for decades long before the hype of AI and ML was around. The ROC curve gives us a "one stop shop" for estimating a value of alpha that results in maximal area under a curve.

In fact, maximizing the area under a given ROC curve winds up being an effective way to judge the differences between one method and another. So, if we wanted to compare the glm model against a Support Vector Machine model, we could use the respective AUC (Area Under Curve) metric to help us. This isn't the only way to do this but it's reasonable for now.

```
pred <- ROCR::prediction(predictions = probs,
                         labels = test$diabetes)

perf <- performance(pred,
                    "tpr",
                    "fpr")
plot(perf,colorize=T,
     print.cutoffs.at=seq(0,1,by=0.1),
     lwd=3,las=1,main="Cool ROC Curve")
abline(a = 0, b = 1)

grid()
```




```
myroc <- performance(pred,measure="auc")
myroc@y.values[[1]]
```

```
## [1] 0.8507868
```

So what value of alpha corresponds to the stated max AUC of .80 ? We'll have to dig into the performance object to get that but it looks to be between 0.30 and 0.40. Note that this is somewhat academic since knowing the max AUC alone helps us decide if our model is any "good". For completeness we could use another R function to nail this down:

```
library(pROC)
proc <- roc(test$diabetes,probs)
round(coords(proc, "b", ret="t", transpose = FALSE),2)
```

```
## [1] 0.35
```

Chapter 4

Other Methods ?

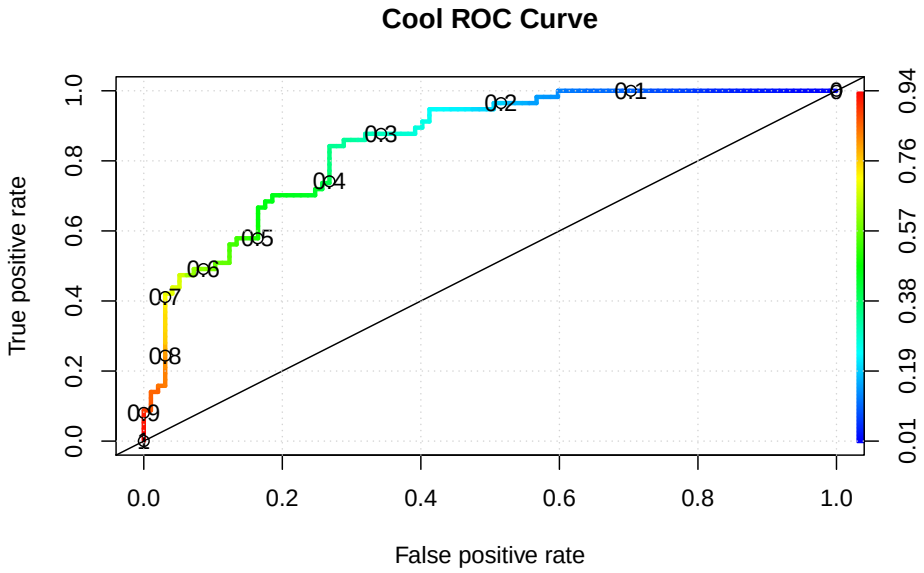
Well, we could use another method to see if it yields better performance as determined by the AUC ? Let's use the **ranger** function which is a fast implementation of random forests. One thing you will notice is that we need to include the "probability" argument in the call to ranger to get the necessary probabilities for computing the AUC. This is one of the aggravations with using different functions. They all have their own peculiar way of doing things.

```
library(ranger)
ranger_mod <- ranger(diabetes ~ .,
                     data = train,
                     probability = TRUE,mtry=4)

# Returns probabilities
ranger_pred <- predict(ranger_mod,data=test)

myroc <- roc(test$diabetes,
             ranger_pred$predictions[,2])

myroc$auc
```

```
## Area under the curve: 0.8502
```

```
pred <- ROCR::prediction(ranger_pred$predictions[,2],
                         test$diabetes)
perf <- performance(pred,
                    "tpr",
                    "fpr")
plot(perf,colorize=T,
        print.cutoffs.at=seq(0,1,by=0.1),
     lwd=3,las=1,main="Cool ROC Curve")
```

```
abline(a = 0, b = 1)

grid()
```




```
rroc <- performance(pred,measure="auc")
rroc@y.values[[1]]
```

```
## [1] 0.8502442
```

It turns out that this didn't appear to improve things - at least with one invocation of the method.

4.1 Improving The Model(s)

We haven't accomplished very much here because we need to look at multiple versions of the data in case we sampled a number of outliers in the creation of our training data. Or, maybe we have excluded a large number of outliers in the training set so they wound up in the test data set which means that the predictive power of our model isn't as robust as it should be.

Our next steps should involve creating multiple versions of the training and test pairs (say 3 times), compute the optimal AUC, and then look at how those values vary for each of those individual versions. If the AUCs vary widely then maybe our model is over training. If it's not varying widely, it could be that that the model has high bias.

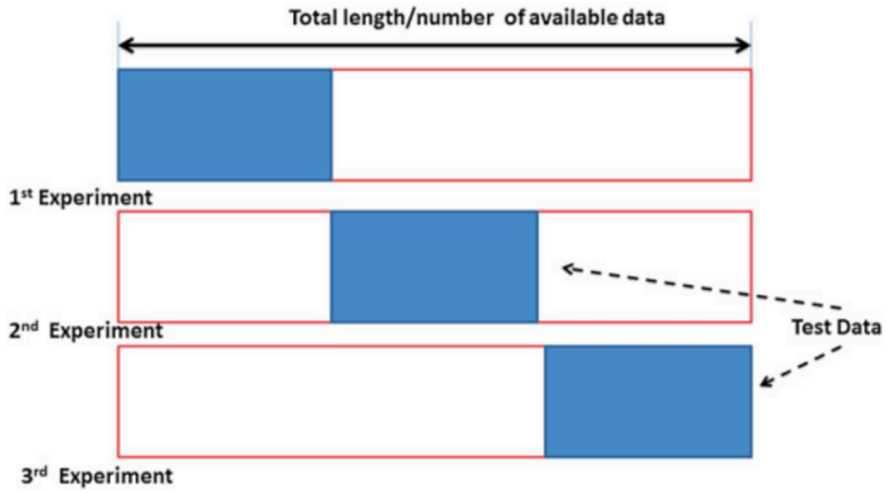


Figure 4.1:

4.2 Cross Fold Validation

This is a method that gives us multiple estimates of out-of-sample error, rather than a single estimate. In particular, we'll use an approach called "K-Fold Cross Validation" where we will partition our data into 3 individual "folds"" which are basically equal in size. Then we'll create a loop that does the following:

- Combines 2 of the folds into a training data set
- Builds a model on the combined 2-folds data
- Applies the model to holdout fold
- Computes the AUC value and stores it

Each fold is simply a portion of the data. We'll generate a list called "folds" that contains 3 elements each of which are 256 index elements corresponding to rows in pm. The way we did the sample insures that each row shows up only in one fold.

To drive this home, and in case the graphic didn't help, consider the following simple data frame:

```
##   id   m1   m2   m3
## 1 a1 0.89 0.36 0.10
## 2 b2 0.75 0.27 0.22
## 3 c3 0.98 0.85 0.95
## 4 d4 0.04 0.36 0.75
## 5 e5 0.90 0.30 0.82
## 6 f6 0.87 0.76 0.42
## 7 g7 0.78 0.84 0.59
```

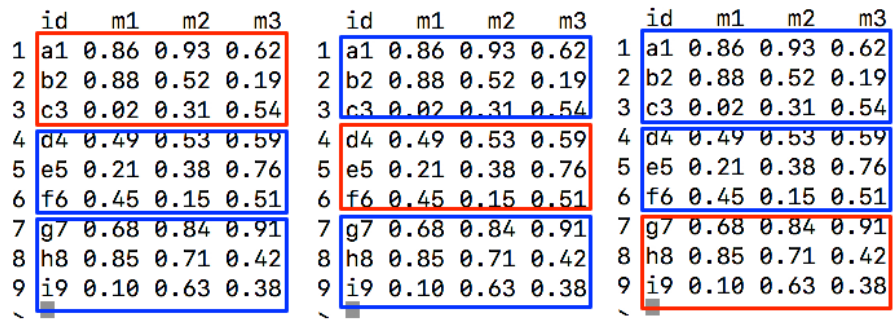


Figure 4.2:

```
## 8 h8 0.38 0.46 0.80
## 9 i9 0.04 0.73 0.89
```

If we created three folds out of this data frame it would look like the following:

Here is our function to implement the K-Fold validation. It's pretty straightforward to define in terms of coding though it winds up being somewhat specific to the particular method we are using.

```
cross_fold <- function(numofolds = 3) {

  # Function to Do Cross fold validation

  # Split the data into K folds (numofolds)

  folds <- split(sample(1:nrow(pm)),1:numofolds)

  # We setup some blank lists to stash results
  folddf    <- list()  # Contains folds
  modl      <- list()  # Hold each of the K models
  predl     <- list()  # Hold rach of the K predictions
  auc       <- list()  # Hold the auc for a given model

  # Create a formula that can be used across multiple
  # iterations through the loop.

  myform <- "diabetes ~ ."

  for (ii in 1:length(folds)) {

    # This list holds the actual model we create for each of the
    # 10 folds
```

```
    modl[[ii]] <- glm(formula = myform,
                      data = pm[-folds[[ii]],],
                      family = "binomial"
    )

    # This list will contain / hold the models build on the fold

    predl[[ii]]  <- predict(modl[[ii]],
                            newdata=pm[folds[[ii]],],
                            type="response")

    # This list will hold the results of the AUC per iteration

    pred <- ROCR::prediction(predl[[ii]],
                             pm[folds[[ii]],]$diabetes)

    roc  <- performance(pred,measure="auc")
    auc[[ii]] <- roc@y.values[[1]]
  }
  return(unlist(auc))
}
```

Running this is now quite simple. By default, this function will loop three times corresponding to the number of folds. During each iteration it will:

- use glm to build a model on the training folds
- create a prediction object using the training fold
- compute the underlying AUC associated with the prediction
- store the AUC in a vector

At the end of the function, the vector containing the computed AUCs will be returned.

```
set.seed(123)
cross_fold()
```

```
## [1] 0.8381928 0.8136312 0.8284142
```

```
# Use more folds
set.seed(123)
cross_fold(8)
```

```
## [1] 0.8676471 0.8290598 0.8335738 0.7822762 0.8505859 0.8852290 0.7816377
## [8] 0.8199725
```

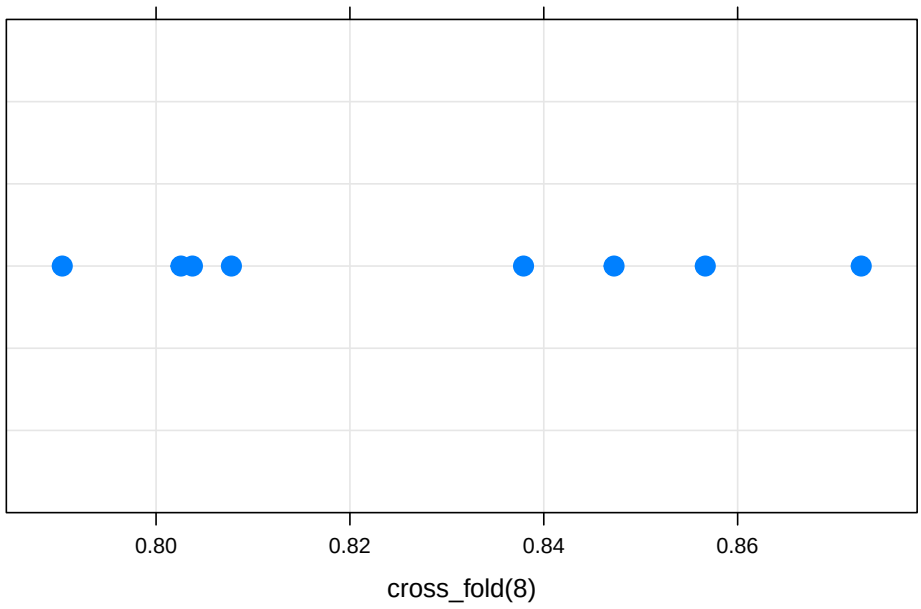
We could take the average of the AUCs to get a sense of how well this method would apply to unseen data.

```
set.seed(123)
mean(cross_fold(8))
```

```
## [1] 0.8312478
```

Let's plot the individual values:

```
stripplot(cross_fold(8),
          main="AUC values for K-Fold Validation",
          type=c("g","p"),pch=19,cex=1.5)
```



Chapter 5

Is There a Better Way ?

In R, as well as with Python, there are a growing number of packages available to help simplify repetitive processes. Building predictive models is no exception especially given that so many sub processes are involved such as splitting data, building a model, making a prediction, comparing it to the labelled data, and so on. The **caret** package provides an easy point of entry into the world of predictive modeling. It provides the following features:

```
- Streamlined and consistent syntax for more than
  200 different models
- Can implement any of the 238 different methods using a single function
- Easy data splitting to simplify the creation of train / test pairs
- Realistic model estimates through built-in resampling
- Convenient feature importance determination
- Easy selection of different performance metrics (e.g. "ROC","Accuracy", "Sensitivity")
- Automated and semi-automated parameter tuning
- Simplifed comparison of different models
```

The caret package was designed specifically for predictive modeling and, in particular, to provide an intuitive approach to creating, managing, and comparing different models emerging from various methods. Let's work through our previous examples using functions from **caret**.

5.1 Data Splitting Using Caret

Let's split the data into a training / test pair. The caret package provides some useful functions for this one of which is the **CreateDataPartition** function.

```
idx <- createDataPartition(pm$diabetes,
                           p=.80,
```

```
                              list=FALSE)

Train <- pm[idx,]
Test  <- pm[-idx,]

nrow(Train)
```

```
## [1] 615
```

Now we can use this to create a training object with caret. We'll create a GLM model similar to the one we've already created. The primary, and most frequently used, function in **caret** is the **train** function. In this example we'll use it build a model using the **glm** method. We will also specify that "Accuracy" will be the preferred performance measure.

```
myglm_caret <- train(diabetes ~ .,
                     data = Train,
                     method = "glm",
                     metric = "Accuracy")
myglm_caret
```

```
## Generalized Linear Model
##
## 615 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 615, 615, 615, 615, 615, 615, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.7711624  0.4728834
```

5.2 Specifying Control Options

We can even request cross fold validation without having to write our own function to do this. To do this requires the specification of a "control" object which contains information that we would like for the **train** function to consider as it does its work.

Not every invocation of **train** requires an associated **trainControl** object although as you become more experienced building models, you will frequently use this approach.

```
control <- trainControl(method = "cv", number = 5)

myglm_caret <- train(diabetes ~ .,
                     data = Train,
                     method = "glm",
                     metric = "Accuracy",
                     trControl = control)
myglm_caret
```

```
## Generalized Linear Model
##
## 615 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 492, 492, 492, 492, 492
## Resampling results:
##
##   Accuracy   Kappa
##   0.7772358  0.4789439
```

5.3 Inspecting The Model

The object returned by caret has a great deal of information packed into it much of which is there to support reproducibility. Some key aspects of the object include, in this case, the Accuracy computation for each fold.

```
myglm_caret$resample
```

```
##    Accuracy     Kappa Resample
## 1 0.7642276 0.4612596    Fold1
## 2 0.7967480 0.5194562    Fold2
## 3 0.7886179 0.5142770    Fold3
## 4 0.7235772 0.3270035    Fold4
## 5 0.8130081 0.5727232    Fold5
```

```
myglm_caret$results
```

```
##   parameter  Accuracy     Kappa AccuracySD    KappaSD
## 1      none 0.7772358 0.4789439 0.03477927 0.09365205
```

```
# Note that the final reported Accuracy metric is simply the average of
# the reported Accuracy values for each fold
```

```
myglm_caret$results[2] == mean(myglm_caret$resample$Accuracy)
```

```
##   Accuracy
## 1     TRUE
```

Of course, you can always look at the model itself to get summary information just as you could if you were not using the **train** function. That is, the **caret** package does not try to conceal or replace what could be done using standard approaches. It overlays the model with information in a way that is transparent.

```
summary(myglm_caret)
```

```
##
## Call:
## NULL
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.8341  -0.7359  -0.4129   0.7035   3.0465
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -8.683585   0.822689 -10.555  < 2e-16 ***
## pregnant     0.124630   0.036130   3.449 0.000562 ***
## glucose      0.037852   0.004344   8.714  < 2e-16 ***
## pressure    -0.014394   0.005861  -2.456 0.014046 *
## triceps      0.002132   0.007774   0.274 0.783853
## insulin     -0.001500   0.001041  -1.440 0.149737
## mass         0.092785   0.016824   5.515 3.49e-08 ***
## pedigree     1.357094   0.348000   3.900 9.63e-05 ***
## age          0.006461   0.010406   0.621 0.534661
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 796.05  on 614  degrees of freedom
## Residual deviance: 572.82  on 606  degrees of freedom
## AIC: 590.82
##
## Number of Fisher Scoring iterations: 5
```

5.4 How Well Did It Perform ?

Remember that one of the features of using caret is that it can help us estimate the out of sample error we will experience when applying our model to new or unseen data. The above model provides an estimate of out of band accuracy as .77

```
myglm_caret
```

```
## Generalized Linear Model
##
## 615 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 492, 492, 492, 492, 492
## Resampling results:
##
##   Accuracy   Kappa
##   0.7772358  0.4789439
```

Let's create a prediction object using the Test data to see how close we came to this estimate. It's a little worse than caret's estimate which is to be expected. Also, this is just an estimate using a single threshold when using the predict function. There are more sophisticated ways to estimate the accuracy.

```
mypreds_glm <- predict(myglm_caret, Test)

# Create a Table of known outcomes vs the predicted outcomes
outcome <- table(preds=mypreds_glm,actual=Test$diabetes)

(acc_outcome <- sum(diag(outcome))/sum(outcome))
```

```
## [1] 0.7581699
```

5.5 Comparing Performance Across Other Methods

The advantage of the **train** function is that we can use the same control objects across a number of modeling techniques which then makes it easier to compare performance across various methods.

As an example, instead of using the "glm" method we could pick another one such as Decision Tree. All we need to know is the name of the method we want.

A complete list of supported models listed by category can be found here

```
control <- trainControl(method = "cv", number = 5)

myrpart_caret <- train(diabetes ~ .,
                       data = Train,
                       method = "rpart",
                       metric = "Accuracy",
                       trControl = control)
myrpart_caret
```

```
## CART
##
## 615 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 492, 492, 492, 492, 492
## Resampling results across tuning parameters:
##
##   cp          Accuracy   Kappa
##   0.01860465  0.7430894  0.4023315
##   0.11162791  0.7333333  0.4178842
##   0.22790698  0.6666667  0.1355614
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.01860465.
```

This method employs a Decision Tree approach which also involves use of "hyper parameters". However, at this point we don't really need to know much about those (although we should) when selecting the method. The larger point is that all we need to know is the name of the alternative method and we can reuse the previous **control** object.

5.6 Different Performance Measures

Not only can we easily select different methods we can also select different performance measures. It does require changes to the control object and arguments to the **train** function though we do not need to read the underlying help pages for a given method to do this. This is a true convenience and time saver that makes reproducing these experiments much easier.

In this example we want to use the "Area Under Curve" (AUC) metric that comes from an associated ROC curve. To do this will require the model to gen-

erate class probabilities from which to build the ROC curve so this information needs to be specified in the control object.

```
control <- trainControl(classProbs = TRUE,
                        summaryFunction = twoClassSummary,
                        method = "cv",
                        number = 8)

myglm_caret_roc <- train(diabetes ~ .,
                         data = Train,
                         method = "glm",
                         metric = "ROC",
                         trControl = control)

myglm_caret_roc
```

```
## Generalized Linear Model
##
## 615 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (8 fold)
## Summary of sample sizes: 538, 538, 538, 538, 538, 538, ...
## Resampling results:
##
##   ROC        Sens   Spec
##   0.8345335  0.895  0.5712251
```

This is a true convenience and we don't have to use a separate R package to compute the AUC. It becomes a by product of the modeling process.

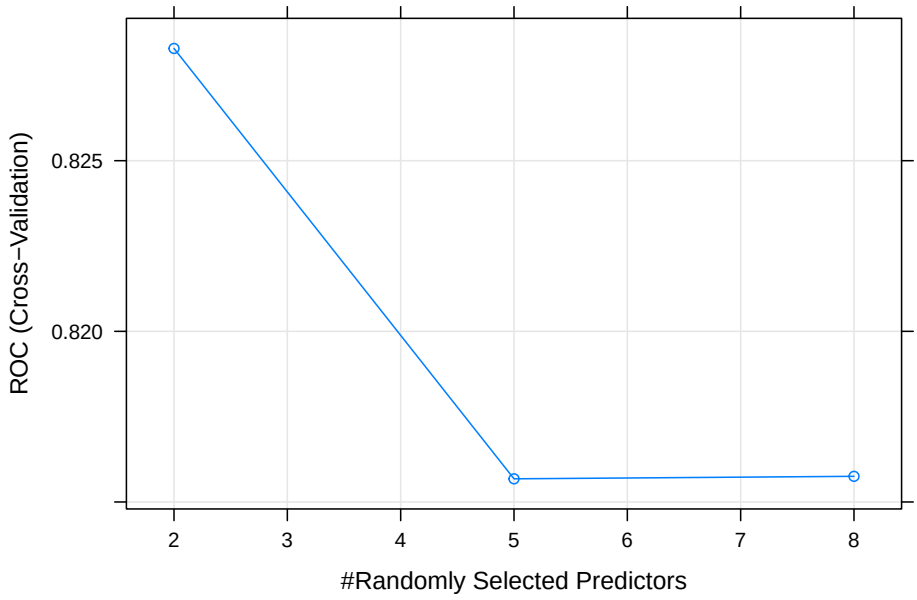
Specifically, the control object can remain the same across different methods assuming that we wish to continue with classification. Here, we'll use Random Forests which are a generalization beyond a single Decision Tree.

```
# The following is the same control object from before
control <- trainControl(classProbs = TRUE,
                        summaryFunction = twoClassSummary,
                        method = "cv",
                        number = 8)
# We'll
myrf_caret <- train(diabetes ~ .,
                data = Train,
 #               method = "svmLinear",
                method = "rf",
                metric = "ROC",
```

```
                trControl = control)

myrf_caret
```

```
## Random Forest
##
## 615 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (8 fold)
## Summary of sample sizes: 539, 538, 538, 538, 538, 538, ...
## Resampling results across tuning parameters:
##
##   mtry  ROC        Sens    Spec
##   2     0.8282888  0.8650  0.5947293
##   5     0.8156784  0.8475  0.5945513
##   8     0.8157514  0.8625  0.5947293
##
## ROC was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 2.
```

```
plot(myrf_caret)
```



Chapter 6

Feature Importance

Another advantage of using the caret **train** function is that it provides a method to determine variable importance. This is useful when considering what features to include or not when building a model. If we summarize a given model, our myglm_caret model, we'll see that some of our predictors are not significant.

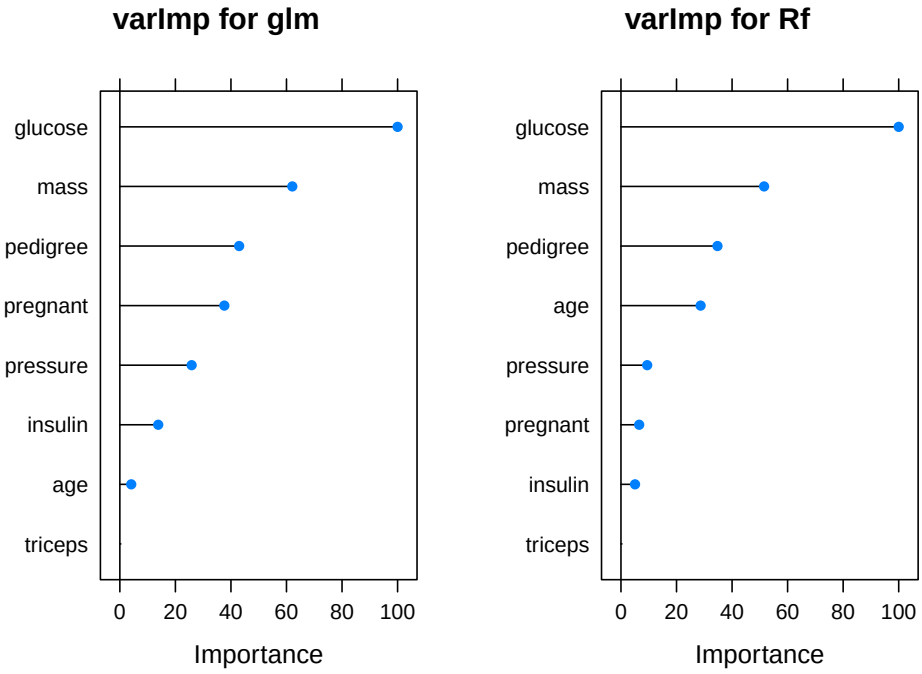
We could use the **varImp** to use statistics generated by the specific modeling process itself. For more complex modeling techniques this winds up being very useful since digging into the model diagnostics can be daunting - although quite useful.

```
varImp(myglm_caret)
```

```
## glm variable importance
##
##          Overall
## glucose  100.000
## mass      62.095
## pedigree  42.957
## pregnant  37.622
## pressure  25.851
## insulin   13.818
## age        4.107
## triceps    0.000
```

If you wanted to see how the different models rates the significance of predictor variables then you can easily plot them.

```
library(gridExtra)
p1 <- plot(varImp(myglm_caret),main="varImp for glm")
p2 <- plot(varImp(myrf_caret),main="varImp for Rf")
grid.arrange(p1,p2,ncol=2)
```

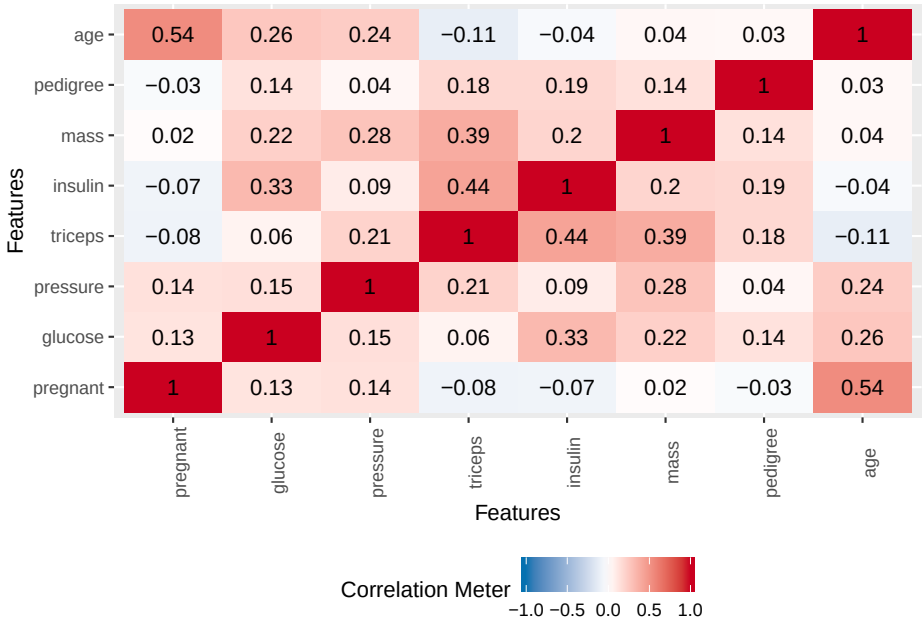


6.0.1 Feature Elimination

The caret package also supports "recursive feature elimination" which automates the selection of optimal features. This can be controversial since such a process could work at the expense of important statistical considerations. However, it remains a tool in the Machine Learning toolbox.

Let's work though an example of this using caret functions. First, we'll remove highly correlated predictor variables from consideration. We don't really have a lot of highly correlated variables. It turns out that "age" is correlated with "pregnant" at a level of 0.54.

```
plot_correlation(pm[,1:8], type="continuous")
```




```
# find attributes that exceed some spcified threshold
highlyCorrelated <- findCorrelation(cor(pm[,1:8]),
                                    cutoff=0.5,
                                    names = TRUE)

# print indexes of highly correlated attributes
print(highlyCorrelated)
```

```
## [1] "age"
```

6.0.2 The rfe Function

Let's apply the RFE method on the Pima Indians Diabetes data set. The algorithm is configured to explore all possible subsets of the attributes. All 8 attributes are selected in this example, although in the plot showing the accuracy of the different attribute subset sizes, we can see that just 4 attributes gives almost comparable results

```
rfFuncs$summary <- twoClassSummary
control <- rfeControl(functions=rfFuncs,
                      method="cv",
                      number=4)

# run the RFE algorithm
results <- rfe(pm[,1:8],
```

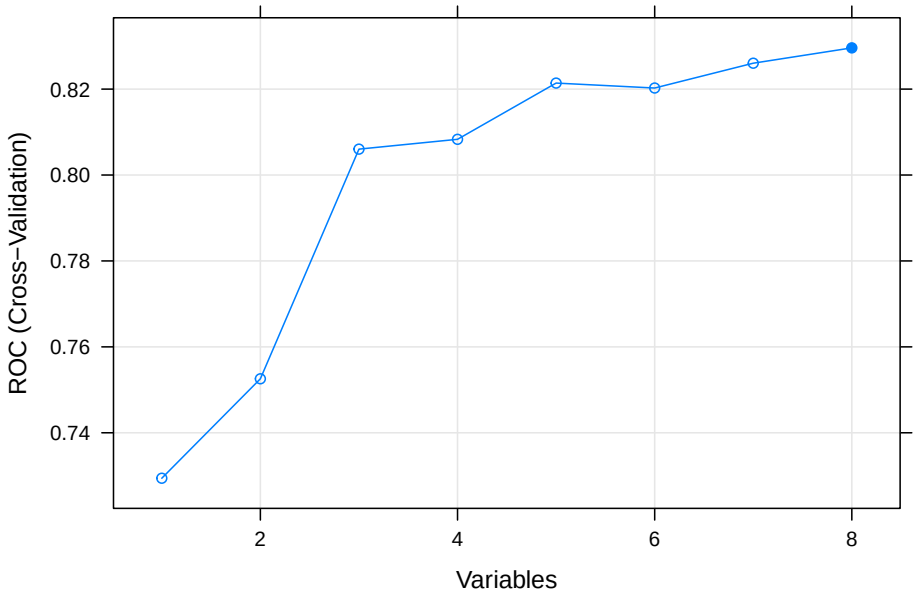
```
               pm[,9],
               sizes=c(1:8),
               rfeControl=control,
               metric="ROC")

# summarize the results
print(results)
```

```
##
## Recursive feature selection
##
## Outer resampling method: Cross-Validated (4 fold)
##
## Resampling performance over subset size:
##
##  Variables    ROC  Sens   Spec   ROCSD  SensSD  SpecSD Selected
##          1 0.7294 0.852 0.4142 0.02253 0.03946 0.02239
##          2 0.7526 0.792 0.5709 0.01260 0.03637 0.04620
##          3 0.8060 0.828 0.5597 0.02291 0.03233 0.07755
##          4 0.8083 0.836 0.5858 0.03138 0.06075 0.06826
##          5 0.8214 0.852 0.5709 0.02983 0.05327 0.09069
##          6 0.8203 0.858 0.5522 0.03067 0.02000 0.09975
##          7 0.8260 0.842 0.5560 0.02251 0.03020 0.10857
##          8 0.8296 0.856 0.5784 0.02589 0.03637 0.09701        *
##
## The top 5 variables (out of 8):
##    glucose, mass, age, pregnant, insulin
```

```
# list the chosen features
predictors(results)
```

```
## [1] "glucose"  "mass"     "age"      "pregnant" "insulin"  "pedigree"
## [7] "triceps"  "pressure"
```

```
# plot the results
plot(results, type=c("g", "o"))
```


ROC (Cross−Validation)
0.74
0.76
0.78
0.80
0.82
2
4
6
8
Variables

Chapter 7

Comparing Models

One of the more frequent activities in Machine Learning relates to setting up "shoot outs" between different models to see which one will perform the best. This is something we could do without **caret** but the package does help accomplish this using a standard interface. We'll keep using the Pima Indians Data and (re)build a few models. We'll use a common **control** object as well as a seed to maintain reproducibility.

```
control <- trainControl(method="cv",
                        number=5,
                        summaryFunction = twoClassSummary,
                        classProbs = TRUE)

# Train the glm model
set.seed(7)
model_glm <- train(diabetes ~ .,
                   data=pm,
                   method="glm",
                   metric="ROC",
                   trControl=control)

# Train the Decision Tree <odel
set.seed(7)
model_rpart <- train(diabetes~.,
                   data=pm,
                   method="rpart",
                   metric="ROC",
                   trControl=control)

# Train the Random Forest model
set.seed(7)
```

```
model_rf <- train(diabetes~.,
                  data=pm,
                  method="rf",
                  metric="ROC",
                  trControl=control)

# Train the knn model
set.seed(7)
model_knn <- train(diabetes~.,
                  data=pm,
                  method="knn",
                  metric="ROC",
                  trControl=control)

# Use the resamples function to prep for comparisons
results <- resamples(list(GLM   = model_glm,
                          RPART = model_rpart,
                          RF    = model_rf,
                          KNN   = model_knn))
```

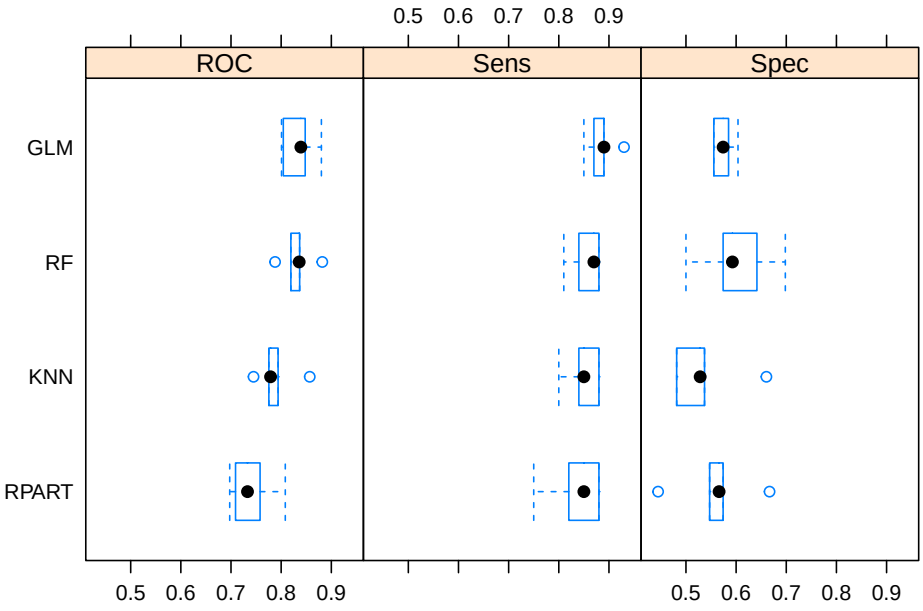
Now we can easily look at how well the different models compare:

```
# summarize the distributions
summary(results)
```

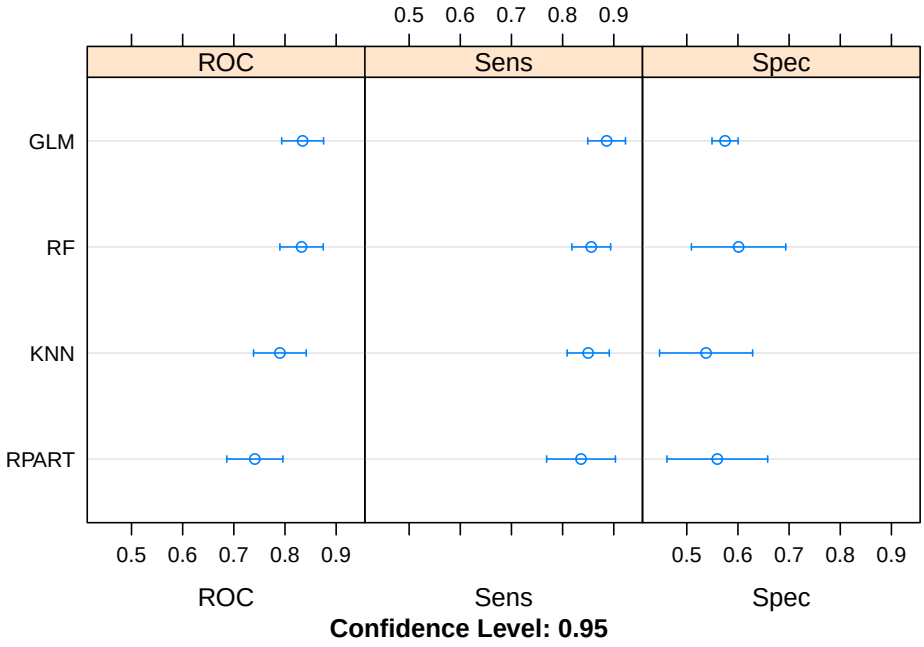
```
##
## Call:
## summary.resamples(object = results)
##
## Models: GLM, RPART, RF, KNN
## Number of resamples: 5
##
## ROC
##            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## GLM   0.8007407 0.8042593 0.8392593 0.8344745 0.8479245 0.8801887    0
## RPART 0.6974074 0.7090566 0.7329245 0.7411184 0.7579630 0.8082407    0
## RF    0.7878704 0.8193519 0.8359434 0.8323997 0.8368519 0.8819811    0
## KNN   0.7449074 0.7754630 0.7788889 0.7900217 0.7938679 0.8569811    0
##
## Sens
##       Min. 1st Qu. Median  Mean 3rd Qu. Max. NA's
## GLM   0.85    0.87   0.89 0.886    0.89 0.93    0
## RPART 0.75    0.82   0.85 0.836    0.88 0.88    0
## RF    0.81    0.84   0.87 0.856    0.88 0.88    0
## KNN   0.80    0.84   0.85 0.850    0.88 0.88    0
##
```

```
## Spec
##            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## GLM   0.5555556 0.5555556 0.5740741 0.5747729 0.5849057 0.6037736    0
## RPART 0.4444444 0.5471698 0.5660377 0.5596785 0.5740741 0.6666667    0
## RF    0.5000000 0.5740741 0.5925926 0.6012579 0.6415094 0.6981132    0
## KNN   0.4814815 0.4814815 0.5283019 0.5377358 0.5370370 0.6603774    0
```

```
# boxplots of results
bwplot(results)
```




```
# dot plots of results
dotplot(results)
```


0.5 0.6 0.7 0.8 0.9
ROC
Sens
Spec
GLM
RF
KNN
RPART
0.5 0.6 0.7 0.8 0.9
0.5 0.6 0.7 0.8 0.9
ROC
Sens
Spec
Confidence Level: 0.95

Chapter 8

Using Exeternal ML Frameworks

There are a number of companies that provide easy access to Machine Learning services including Google, Amazon, Data Robot, and H2o. In particular, the company H20.ai provides frameworks for accessible Machine Learning by experts and non-experts. They promote the idea of "citizen data science" which seeks to lower barriers to participation in the world of AI. While they have a commercial product, they also provide an open source tool:

> H2O is a fully open source, distributed in-memory machine learning platform with linear scalability. H2O supports the most widely used statistical & machine learning algorithms including gradient boosted machines, generalized linear models, deep learning and more.

Moreover, H2O provides access to an "Auto ML" service that selects methods appropriate to a given data set. This is useful to help jump start ideas.

> H2O also has an industry leading AutoML functionality that automatically runs through all the algorithms and their hyperparameters to produce a leaderboard of the best models. The H2O platform is used by over 18,000 organizations globally and is extremely popular in both the R & Python communities.

Better yet, there is an R package called, somewhat unimaginatively, "h2o" "which provides:

> R interface for 'H2O', the scalable open source machine learning platform that offers parallelized implementations of many supervised and unsupervised machine learning algorithms such as Generalized Linear Models, Gradient Boosting Machines (including XGBoost), Random Forests, Deep Neural Networks (Deep Learning), Stacked

Ensembles, Naive Bayes, Cox Proportional Hazards, K-Means, PCA, Word2Vec, as well as a fully automatic machine learning algorithm (AutoML).

8.1 H2O In Action

The package must first be installed which can done using the install.packages function (or the menu in R Studio). Loading the library is done just as you would any other library.

```
library(h2o)
```

The goal of using this library is not to replace the methods available to you in R but, just like the caret package, seeks to provide a uniform interface for a variety of underlying methods. This includes common methods including an "Auto ML" service that picks methods for you. Let's apply h2o to our work. The underlying h2o architecture uses a "running instance" concept that can be initialized and accessed from R. You initialize it once per interactive session.

```
h2o.init()
```

```
##
## H2O is not running yet, starting it now...
##
## Note:  In case of errors look at the following log files:
##     /var/folders/wh/z0v5hqgx3dzdfgz47lnbr_3w0000gn/T//RtmpVqPBVd/h2o_esteban_started
##     /var/folders/wh/z0v5hqgx3dzdfgz47lnbr_3w0000gn/T//RtmpVqPBVd/h2o_esteban_started
##
##
## Starting H2O JVM and connecting: .. Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         2 seconds 966 milliseconds
##     H2O cluster timezone:       America/New_York
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.26.0.2
##     H2O cluster version age:    6 months and 30 days !!!
##     H2O cluster name:           H2O_started_from_R_esteban_bqw141
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   1.78 GB
##     H2O cluster total cores:    4
##     H2O cluster allowed cores:  4
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
```

```
##     H2O Internal Security:      FALSE
##     H2O API Extensions:         Amazon S3, XGBoost, Algos, AutoML, Core V3, Core V4
##     R Version:                  R version 3.5.3 (2019-03-11)
```

```
## Warning in h2o.clusterInfo():
## Your H2O cluster version is too old (6 months and 30 days)!
## Please download and install the latest version from http://h2o.ai/download/
```

Once the h2o environment has been initialized then work can begin. This will take the form of using R functions provided by the h2o package to read in data and prepare it for use with various methods. Let's repeat the regression on mtcars using h2o functions. Since mtcars is already available in the the R environment we can easily import it into h2o.

```
# Import mtcars
mtcars_h2o_df <- as.h2o(mtcars)
```

```
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
# Idenitfy the variable to be predicted
y <- "mpg"

# Put the predictor names into a vector
x <- setdiff(colnames(mtcars_h2o_df),y)
```

8.2 Create Some Models

Now let's create some training and test data sets. We could do this ourselves using conventional R commands or helper functions from the caret package. However, the h2o package provides its own set of helpers.

```
data(mtcars)
splits <- h2o.splitFrame(mtcars_h2o_df, ratios=0.8,seed=1)
train_h2o <- splits[[1]]
test_h2o  <- splits[[2]]

train_h2o
```

```
##    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## 2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## 3 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## 4 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

```
## 5 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
## 6 14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
##
## [29 rows x 11 columns]
```

Now let's create a model. We'll use the Generalized Linear Model function from h2o. It is important to note that this function is implemented from within h2o. That is, we are not in anyway using any existing R packages to do this nor are we using anything from the care package. Here we'll request a 4-Fold, Cross Validation step as part of the model assembly.

```
h2o_glm_model <- h2o.glm(y=y,x=x,train_h2o,nfolds=4)
```

```
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
summary(h2o_glm_model)
```

```
## Model Details:
## ==============
##
## H2ORegressionModel: glm
## Model Key:  GLM_model_R_1582752355388_1
## GLM Model: summary
##     family     link                               regularization
## 1 gaussian identity Elastic Net (alpha = 0.5, lambda = 1.0664 )
##   number_of_predictors_total number_of_active_predictors
## 1                         10                           9
##   number_of_iterations    training_frame
## 1                    1 RTMP_sid_b1e4_560
##
## H2ORegressionMetrics: glm
## ** Reported on training data. **
##
## MSE:  6.185253
## RMSE:  2.487017
## MAE:  1.940791
## RMSLE:  0.1135999
## Mean Residual Deviance :  6.185253
## R^2 :  0.8392098
## Null Deviance :1115.568
## Null D.o.F. :28
## Residual Deviance :179.3723
## Residual D.o.F. :19
```

```
## AIC :157.1413
##
##
##
## H2ORegressionMetrics: glm
## ** Reported on cross-validation data. **
## ** 4-fold cross-validation on training data (Metrics computed for combined holdout predictions
##
## MSE:  8.847285
## RMSE:  2.974439
## MAE:  2.315981
## RMSLE:  0.1361342
## Mean Residual Deviance :  8.847285
## R^2 :  0.7700083
## Null Deviance :1133.569
## Null D.o.F. :28
## Residual Deviance :256.5713
## Residual D.o.F. :19
## AIC :167.5216
##
##
## Cross-Validation Metrics Summary:
##                              mean          sd cv_1_valid cv_2_valid
## mae                     2.4396207  0.35686472  3.2835414  2.2728186
## mean_residual_deviance  10.026211   3.4173844  18.312454  7.5719833
## mse                     10.026211   3.4173844  18.312454  7.5719833
## null_deviance           283.39224    66.06932  332.70395  127.60623
## r2                     0.74077624 0.060155686  0.7743174  0.6026595
## residual_deviance       64.142815   12.455326   73.24982    45.4319
## rmse                    3.0881271  0.49481392  4.2793055  2.7517238
## rmsle                  0.14022091 0.020872122 0.18750846 0.13113023
##                        cv_3_valid cv_4_valid
## mae                     2.2584624  1.9436611
## mean_residual_deviance   8.042097   6.178309
## mse                      8.042097   6.178309
## null_deviance              372.35  300.90878
## r2                      0.7525647  0.8335634
## residual_deviance       88.463066   49.42647
## rmse                     2.835859  2.4856205
## rmsle                  0.13590272 0.10634225
##
## Scoring History:
##             timestamp   duration iterations negative_log_likelihood
## 1 2020-02-26 16:26:00  0.000 sec          0              1115.56759
##   objective
## 1  38.46785
```

```
##
## Variable Importances: (Extract with `h2o.varimp`)
## =================================================
##
##    variable relative_importance scaled_importance  percentage
## 1        wt          1.19294625        1.00000000 0.208632891
## 2       cyl          0.92951526        0.77917615 0.162561772
## 3      disp          0.78424629        0.65740287 0.137155861
## 4        hp          0.69294345        0.58086729 0.121188021
## 5      carb          0.62287613        0.52213261 0.108934035
## 6        am          0.55736672        0.46721864 0.097477175
## 7        vs          0.46246830        0.38766902 0.080880507
## 8      drat          0.45447201        0.38096604 0.079482047
## 9      gear          0.02108593        0.01767551 0.003687692
## 10     qsec          0.00000000        0.00000000 0.000000000
```

Now we can do a prediction on the object against the test set.

```
(h2o_glm_preds <- h2o.predict(h2o_glm_model,test_h2o))
```

```
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
##    predict
## 1 25.93186
## 2 20.67344
## 3 24.38237
##
## [3 rows x 1 column]
```

Now look at the performance diagnostics:

```
h2o.performance(h2o_glm_model,test_h2o)
```

```
## H2ORegressionMetrics: glm
##
## MSE:  6.762548
## RMSE:  2.60049
## MAE:  2.495891
## RMSLE:  0.1076563
## Mean Residual Deviance :  6.762548
## R^2 :  -2.052306
## Null Deviance :10.87611
## Null D.o.F. :2
## Residual Deviance :20.28765
```

```
## Residual D.o.F. :-7
## AIC :36.24783
```

8.3 Saving A Model

You can save the contents of any h2o generated model by using the h2o.saveModel() function. You could extract pieces of information from the S4 object but saving the model is easy to do - as is reading it back in.

```
model_path <- h2o.saveModel(h2o_glm_model,path=getwd(),force=TRUE)

# If you need to load a previously saved model

saved_model <- h2o.loadModel(model_path)
```

8.4 Using the Auto ML Feature

Are you curious as to what model might be the "best" for your data ? This is a very fertile field of research that keeps growing and some feel will one be the dominant technology in ML - where a model picks a model. Sounds odd but that is where it is heading. Check the current h2o Auto ML documentation for more details. For now, most of the Auto ML services use a set of heuristics to examine data and then find the most appropriate method to build a model. The currently supported method implementations in the opensource version include:

```
- three pre-specified XGBoost GBM (Gradient Boosting Machine) models
- a fixed grid of GLMs, a default Random Forest (DRF)
- five pre-specified H2O GBMs
- a near-default Deep Neural Net
- an Extremely Randomized Forest (XRT)
- a random grid of XGBoost GBMs
- a random grid of H2O GBMs
- a random grid of Deep Neural Nets.
```

8.5 Launching A Job

Of course, it all begins with the idea of specifying a performance metric such as RMSE or the area under a ROC curve. The idea here is that we specify some input, apply transformations, create a test/train pair, and then call the h2o auto function.

```
h2o_auto_mtcars <- h2o.automl(y = y, x = x,
                              training_frame = train_h2o,
                              leaderboard_frame = test_h2o,
                              max_runtime_secs = 60,
                              seed = 1,
                              sort_metric = "RMSE",
                              project_name = "mtcars")
```

```
##
  |
  |                                                                      |   0%
  |
  |=                                                                     |   2%
  |
  |==                                                                    |   3%
  |
  |===                                                                   |   5%
  |
  |====                                                                  |   7%
  |
  |======                                                                |   9%
  |
  |=========                                                             |  13%
  |
  |=========                                                             |  14%
  |
  |==========                                                            |  15%
  |
  |===========                                                           |  17%
  |
  |============                                                          |  19%
  |
  |==============                                                        |  21%
  |
  |===============                                                       |  24%
  |
  |===================                                                   |  29%
  |
  |=====================                                                 |  32%
  |
  |======================                                                |  34%
  |
  |========================                                              |  36%
  |
  |=========================                                             |  37%
```

```
|
|=========================                                  |  39%
|
|=========================                                  |  40%
|
|==========================                                 |  42%
|
|==========================                                 |  43%
|
|===========================                                |  45%
|
|===========================                                |  46%
|
|============================                               |  48%
|
|============================                               |  49%
|
|=============================                              |  51%
|
|=============================                              |  52%
|
|==============================                             |  54%
|
|==============================                             |  55%
|
|===============================                            |  57%
|
|===============================                            |  58%
|
|================================                           |  60%
|
|================================                           |  61%
|
|=================================                          |  61%
|
|=================================                          |  63%
|
|==================================                         |  65%
|
|==================================                         |  66%
|
|===================================                        |  68%
|
|===================================                        |  70%
|
|====================================                       |  71%
```

```
  |
  |=============================================                    |  73%
  |
  |==============================================                   |  75%
  |
  |==============================================                   |  76%
  |
  |===============================================                  |  78%
  |
  |================================================                 |  80%
  |
  |================================================                 |  81%
  |
  |=================================================                |  83%
  |
  |=================================================                |  85%
  |
  |==================================================               |  87%
  |
  |==================================================               |  88%
  |
  |===================================================              |  90%
  |
  |====================================================             |  92%
  |
  |====================================================             |  93%
  |
  |=====================================================            |  95%
  |
  |======================================================           |  97%
  |
  |======================================================           |  98%
  |
  |=======================================================================| 100%
```

Let's check out the object that is returned. It is an S4 object in R which means that it has "slots" which can be accessed via the "@" operator.

```
slotNames(h2o_auto_mtcars)
```

```
## [1] "project_name"  "leader"        "leaderboard"   "event_log"
## [5] "training_info"
```

```
h2o_auto_mtcars@leaderboard
```

```
##                                             model_id
## 1              DeepLearning_1_AutoML_20200226_162602
## 2         GBM_grid_1_AutoML_20200226_162602_model_53
```

```
## 3 DeepLearning_grid_1_AutoML_20200226_162602_model_4
## 4     XGBoost_grid_1_AutoML_20200226_162602_model_14
## 5      XGBoost_grid_1_AutoML_20200226_162602_model_5
## 6         GBM_grid_1_AutoML_20200226_162602_model_52
##   mean_residual_deviance      rmse       mse       mae      rmsle
## 1              0.1049364 0.3239390 0.1049364 0.3196167 0.01441855
## 2              0.2147231 0.4633822 0.2147231 0.4138704 0.02128094
## 3              0.4519215 0.6722511 0.4519215 0.6689904 0.03046289
## 4              0.4586288 0.6772213 0.4586288 0.6318582 0.03060372
## 5              0.4787276 0.6919014 0.4787276 0.5760670 0.03091116
## 6              0.7571985 0.8701716 0.7571985 0.7831136 0.03729489
##
## [85 rows x 6 columns]
```

Stop the H2O instance

```
h2o.shutdown(prompt=FALSE)
```

```
## [1] TRUE
```