Crash course introduction to prediction computation

Paul Yousefi, PhD, MPH

June 2019

Before we start

This lab will performed in R and will use the following packages:

- `pROC`
- `caret`
- `glmnet`
- `ranger`
- `kernlab`

We'll be using data from Tsaprouni et al. 2014 that is publicly available on the NCBI Gene Expression Ombibus (GEO) website (accession number: GSE50660) at https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE50660

All course material, including the data and code used for this lab practical, is available for you to download here:

url for where to download data

Goals

- Partitioning data into training and testing sets
- Evaluating performance of risk scores
- Fitting models in training data
- Predicting outputs from those models in the testing data
- Quantifying model prediction performance

Getting started

To start, I'll load our data into active memory and have a look at what's available:

```
load("dataset.rda")
```

```
ls()
```

[1] "meth" "samples"

So we have two data objects:

- `meth` with DNA methylation data
- `samples` with other phenotype information on the participants of this study

Let's get a better sense of the variables available in `samples`:

```
str(samples)
```

```
> 'data.frame': 464 obs. of  6 variables:
>  $ gsm       : chr  "GSM1225377" "GSM1225378" "GSM1225379" "GSM1225380" ...
>  $ gse       : chr  "GSE50660" "GSE50660" "GSE50660" "GSE50660" ...
>  $ age       : num  50 56 49 64 51 50 47 46 50 56 ...
>  $ sex       : chr  "male" "male" "female" "male" ...
>  $ smoking   : chr  "former" "never" "former" "former" ...
>  $ ever.smoke: num  1 0 1 1 1 1 1 1 1 0 ...
```

```
summary(samples)
```

```
>      gsm                gse                 age            sex
>  Length:464         Length:464         Min.   :38.00   Length:464
>  Class :character   Class :character   1st Qu.:50.00   Class :character
>  Mode  :character   Mode  :character   Median :56.00   Mode  :character
>                                        Mean   :55.39
>                                        3rd Qu.:61.00
>                                        Max.   :67.00
>    smoking            ever.smoke
>  Length:464         Min.   :0.0000
>  Class :character   1st Qu.:0.0000
>  Mode  :character   Median :1.0000
>                     Mean   :0.6142
>                     3rd Qu.:1.0000
>                     Max.   :1.0000
```

```
table(samples$smoking)
```

```
>
> current  former   never
>      22     263     179
```

```
table(samples$ever.smoke)
```

```
>
>   0   1
> 179 285
```

The `smoking` variable has 3 categories, but it's easiest to begin with a binary outcome so let's focus on the `ever.smoke` variable that collapses the **current** and **former** subjects into a single category

- When I talk about predicting smoking going from now on I'll be referring to this `ever.smoke`variable

Applying risk scores

Single variable scores

The simplest type of risk score we can use for prediction is just a single individual variable. The site `cg05575921` in the *AHRR* gene has consistently been the CpG with methylation showing the strongest association with smoking in several studies looking broadly across the genome.

Perhaps the methylation levels of this site would be sufficient to predict whether someone has been a smoker. To see, let's begin by adding this CpG site as a variable to our phenotype data object `samples`:

```
samples$ahrr <- meth["cg05575921", ]
```

We can use a package called `pROC` to see how well different values of our `ahrr` variable explain smoking status:

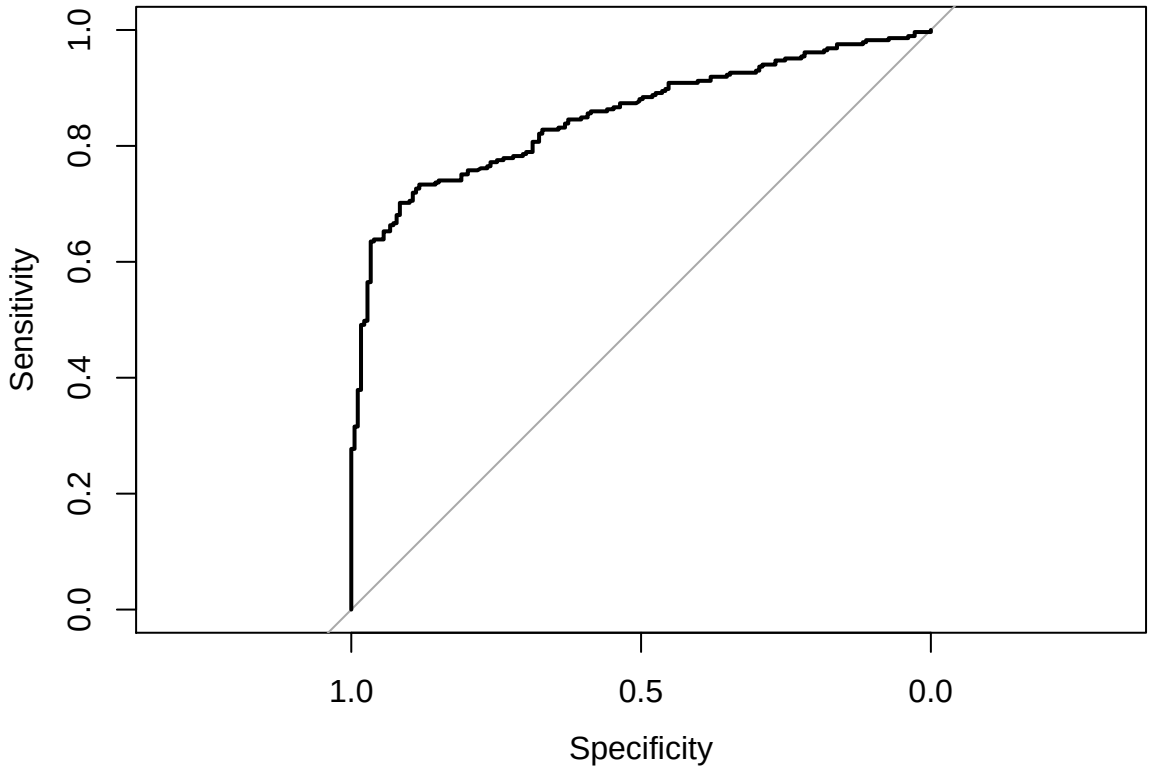
```
## load the pROC package
library("pROC")

## use the formula-based syntax of the package
roc(ever.smoke ~ ahrr, data = samples)
```

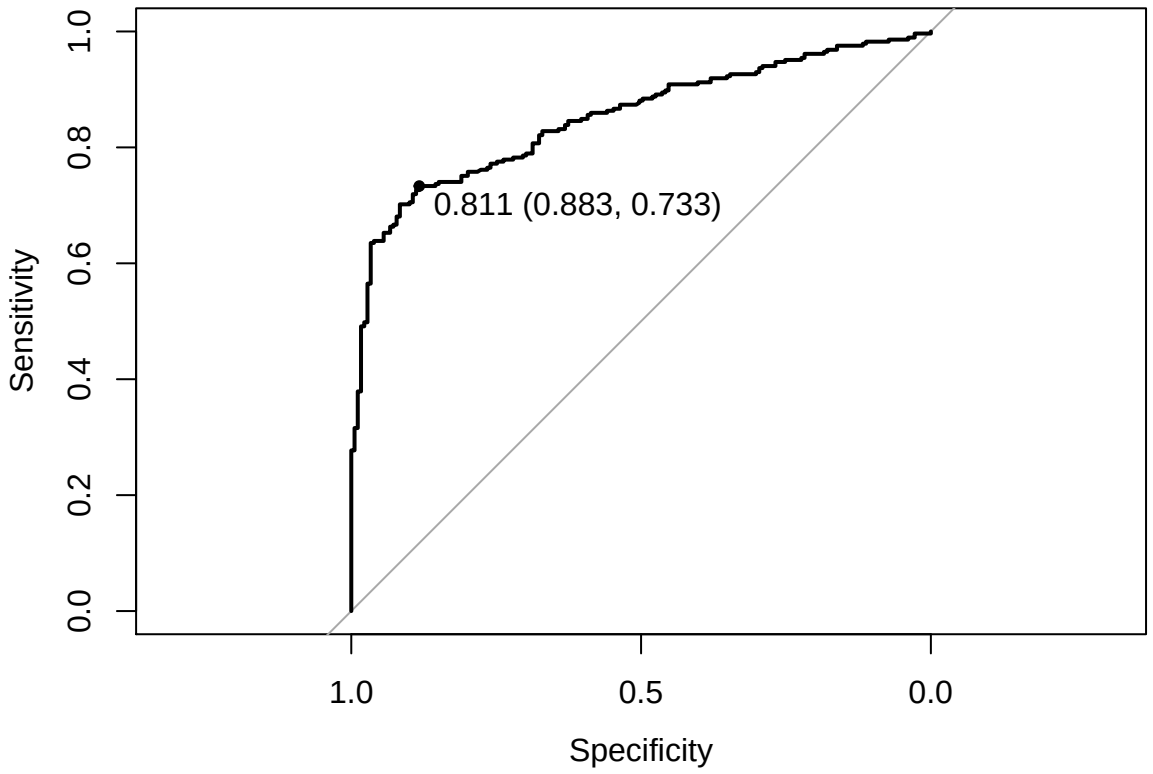
```
>
> Call:
> roc.formula(formula = ever.smoke ~ ahrr, data = samples)
>
> Data: ahrr in 179 controls (ever.smoke 0) > 285 cases (ever.smoke 1).
> Area under the curve: 0.851
```

We can also visualize our results by using the `pROC` package's `plot.roc()` function on the saved output

```
roc.out <- roc(ever.smoke ~ ahrr, data = samples)
plot(roc.out)
```




```
plot(roc.out,
    print.thres="best",
    print.thres.best.method="closest.topleft")
```



Weighted risk scores from published coefficients

Another common approach in omic prediction is to apply a risk score using information from multiple loci or omic measures weighted by previously reported magnitudes of association observed between those features and the outcome of interest. This has perhaps been most often performed using genetic data to compose 'polygenic risk scores', but can easily be extended to other types of data input.

For example, in the context of DNA methylation data we can define and subsequently apply a smoking score derived from the published coefficients of the largest blood epigenome wide association study (EWAS) meta-analysis to date in Joehanes et al. 2016:

> Joehanes R, Just AC, Marioni RE, Pilling LC, Reynolds LM, Mandaviya PR et al. Epigenetic Signatures of Cigarette Smoking. Circ Cardiovasc Genet 2016;

- The coefficients from their models were distributed in their Supplemental Table 2 that I previously saved and load into R below:

```
load("joehanes2016_st2_bonf.rda")
```

The `joehanes` object has summary information on the 2617 CpGs that were significant at a Bonferroni p-value threshold in the original meta-analysis and that were available in our methylation dataset

```
str(joehanes)
```

```
> 'data.frame': 2617 obs. of  14 variables:
>  $ probe.id            : chr  "cg16145216" "cg19406367" "cg05603985" "cg14099685" ...
>  $ infinium.design.type: chr  "I" "II" "II" "II" ...
>  $ chromosome          : chr  "1" "1" "1" "11" ...
>  $ location..hg19.     : chr  "42,385,662" "66,999,929" "2,161,049" "47,546,068" ...
>  $ strand              : chr  "R" "R" "F" "R" ...
>  $ gene.symbol         : chr  "HIVEP3" "SGIP1" "SKI" "CUGBP1" ...
```

```
>  $ effect              : num  0.0298 0.0175 -0.0122 -0.0124 -0.0262 -0.0182 -0.0166 -0.0163 -0.0134 -0
>  $ std..error          : num  0.002 0.0013 0.0009 0.0009 0.002 0.0014 0.0013 0.0013 0.0011 0.0067 ...
>  $ z.value             : num  14.5 13.9 -13.8 -13.7 -13.4 ...
>  $ p.value             : num  6.7e-48 7.0e-44 1.8e-43 1.5e-42 6.1e-41 ...
>  $ fdr                 : num  3.3e-42 1.7e-38 2.8e-38 1.8e-37 5.9e-36 ...
>  $ previously.seen     : int  1 1 1 1 1 0 1 1 1 1 ...
>  $ small.molecules     : int  0 0 0 0 0 0 0 0 0 0 ...
>  $ sign                : chr  "++++++++++-++++" "+++++++++++++++" "----------+--" "----------
```

Let's restrict our big methylation data object, `meth`, to just the CpGs that are in the `joehanes` list. This keeps the CpGs that we expect to be most related to smoking behavior, in the correct format, while reducing the size of the data were working with:

```
X <- meth[joehanes$probe.id, ]
## transpose to make columns = methylation site variables,
##                  rows = subjects/observations
X <- t(X)
```

To compute an individual's risk score using these coefficients, we need to take the sum of each coefficient multiplied by the participant's value at the corresponding variable, for example:

$$\hat{Y}_{JoehanesScore} = \sum_{i}^{2617CpGs} \hat{\beta}_i X_i$$

- Where $\hat{\beta}_i$ are the individual previously estimated Joehanes et al. coefficients and X_i are their corresponding variables (CpG site measurements in this case).

Equivalently, we can compute the exact same quantity, $\hat{Y}_{JoehanesScore}$, more simply using matrix multiplication:

$$\hat{Y}_{JoehanesScore} = X\hat{\beta}$$

- Where X is an NxP matrix of all P variables being used and $\hat{\beta}$ is a corresponding vector of all Joehanes et al. coefficients.

To implement this, lets start by making a named vector of the `joehanes` coefficients:

```
coefs <- joehanes$effect
names(coefs) <- joehanes$probe.id
```

We can then use matrix multiplication against our observed DNA methylation values to get our $\hat{Y}_{JoehanesScore}$ values:

```
y.hat <-   X %*% coefs

summary(y.hat)
```

```
>        V1
>  Min.   :-5.621
>  1st Qu.:-5.145
>  Median :-4.924
>  Mean   :-4.844
>  3rd Qu.:-4.603
>  Max.   :-3.486
```

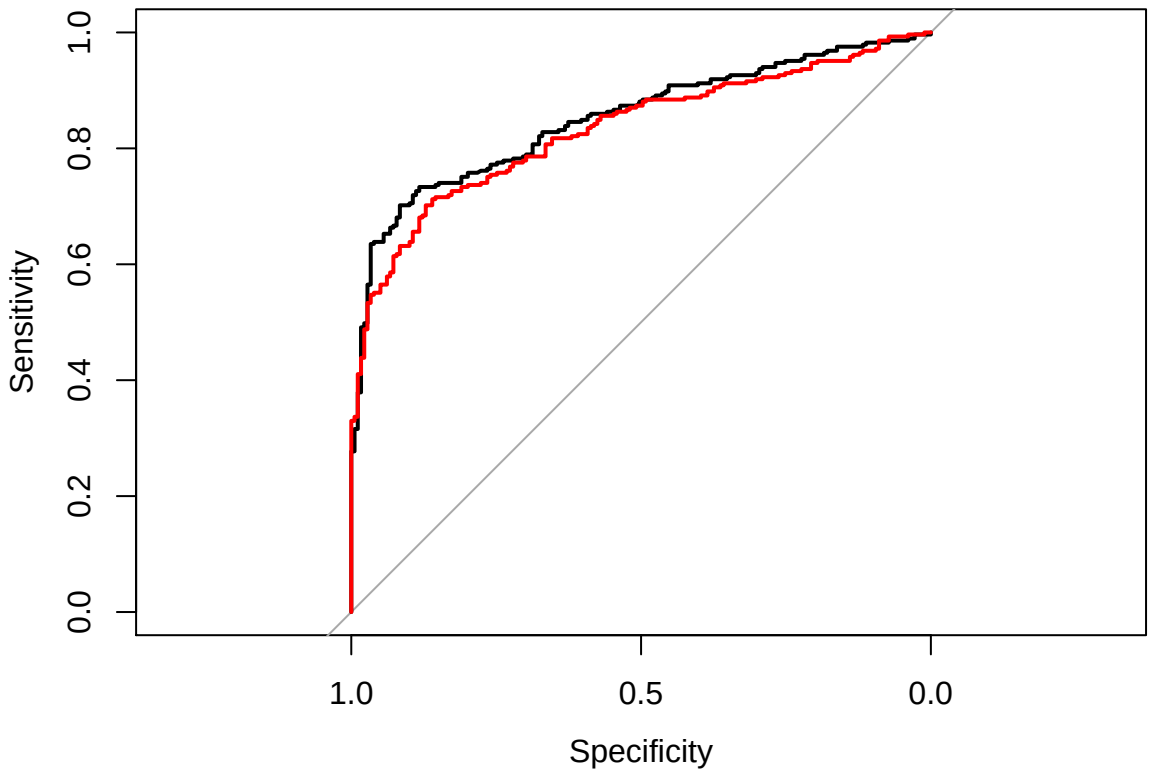
By adding this output as a variable to our `samples` data, we can again use the `pROC` package to evaluate and visualize the prediction performance of this score:

```
samples$y.hat <- as.vector(y.hat)
```

```
roc.out.again <- roc(ever.smoke ~ y.hat, data = samples)
roc.out.again
```

```
>
> Call:
> roc.formula(formula = ever.smoke ~ y.hat, data = samples)
>
> Data: y.hat in 179 controls (ever.smoke 0) < 285 cases (ever.smoke 1).
> Area under the curve: 0.832
```

```
plot(roc.out)
lines.roc(roc.out.again, col="red")
```




```
coords(roc.out.again, "best",
                      best.method="closest.topleft",
                      ret=c("threshold", "accuracy"))
```

```
>  threshold   accuracy
> -4.9185894  0.7693966
```

Does the joehanes score predict never/ever smoking better than just cg05575921 alone?

```
roc.out$auc
```

```
> Area under the curve: 0.851
```

```
roc.out.again$auc
```

```
> Area under the curve: 0.832
```

No, it doesn't appear to! Can you think of any possible explanations for why this might be the case??

Training a novel predictor

Up until now, we've been evaluating the performance of predictors that other independent studies had previously developed. To develop our own prediction models, we first need to split our dataset into two parts:

1. a training set: the major subset of data where we can *fit* our prediction models and estimate their relevant parameter values with access to the outcome variable observations
2. testing set: a smaller sub-set of observations withheld from training where we can *apply* the models with trained and get an independent assessment of the predictor performance

This data splitting approach is essentially replicating within our own dataset the design we used implicitly to evaluate the performance of Joehanes et al. 2016 smoking score:

- There, the observations in the original Joehanes et al. 2016 paper served as the training set and the Tibshirani et al. data we've been using was the testing set
- Here, without access to two independent datasets we'll need to artificially generate a partition within our single dataset: reserving a subset of observations for training, and the remaining subset for testing

The `caret` package has functions to help with splitting the observations of your dataset along with a suite of other tools commonly used to train and test prediction models.

One of these is the `createDataPartition()` function which creates an index of the rows that will be allocated for training vs. testing at a user-selected percentage split of the observations. Below we dedicate 75% percent of our data for training and reserve 25% for testing:

```
library(caret)

set.seed(138) # makes random processes reproducible
Y <- samples$ever.smoke
in.train <- createDataPartition(
  y = samples$ever.smoke,
  ## the outcome data are needed
  p = .75,
  ## The percentage of data in the
  ## training set
  list = FALSE
)
```

The index returned by `createDataPartition()` can then be easily used to split our `samples` data object into two separate objects:

```
training <- samples[ in.train,]
testing  <- samples[-in.train,]

nrow(training)
```

```
> [1] 348
```

```
nrow(testing)
```

```
> [1] 116
```

Manual model training

Once these two independent set of observations have been generated, we can easily fit any model we can think up in the training subset and assess how well it performs in the reserved testing subset.

For example, if we wanted to see how well a linear subset of the 2617 CpGs that were bonferroni significant in Joehanes et al. 2016 performed at predicting our `ever.smoke` variable we could fit a penalized regression model like a lasso (least absolute shrinkage and selection operator) in our training dataset:

```
library(glmnet)
set.seed(20)
fit.lasso <- cv.glmnet(y = training$ever.smoke,
                       x = X[in.train,],
                       family='binomial',
                       alpha=1)
```

And then predict from that model into the reserved testing observations:

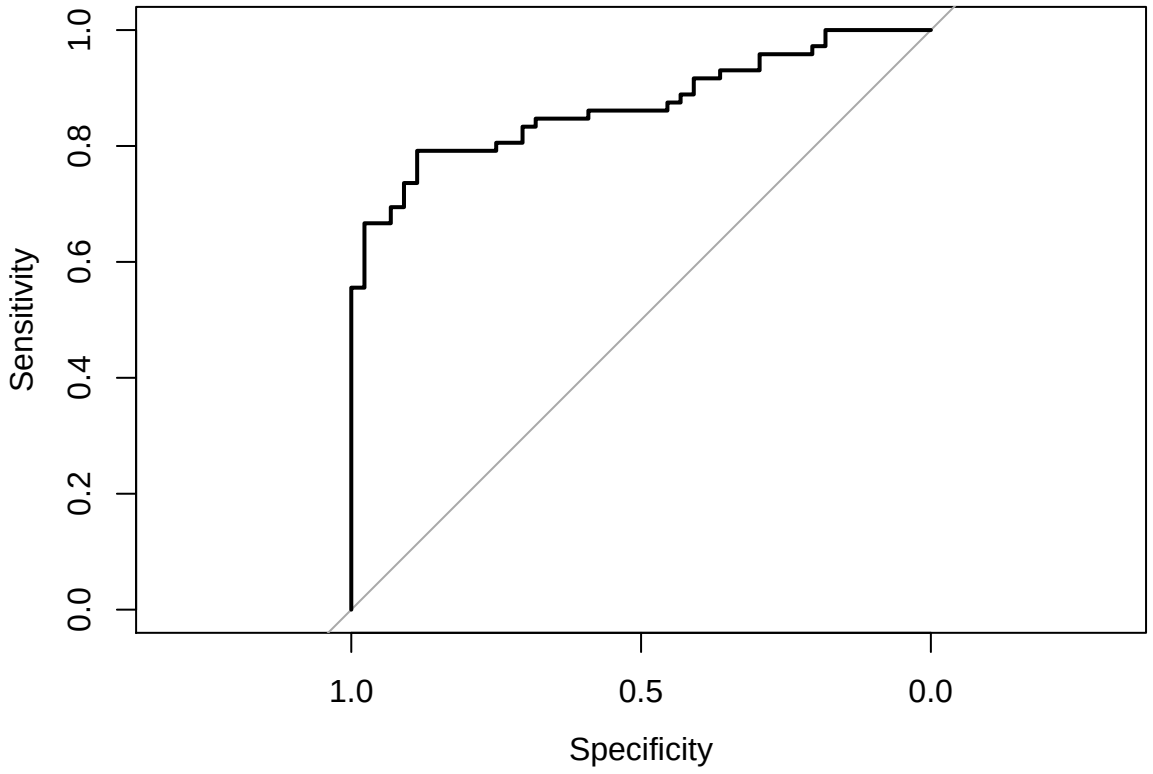
```
pred.lasso <- predict(fit.lasso, newx = X[-in.train,],
                      s = "lambda.min",
                      type = "response")
```

All of the tools we'd employed previously from the `pROC` package can still be used to asses how well our new predictions performed:

```
roc.lasso <- roc(testing$ever.smoke, as.vector(pred.lasso))
roc.lasso
```

```
>
> Call:
> roc.default(response = testing$ever.smoke, predictor = as.vector(pred.lasso))
>
> Data: as.vector(pred.lasso) in 44 controls (testing$ever.smoke 0) < 72 cases (testing$ever.smoke 1).
> Area under the curve: 0.8696
```

```
plot(roc.lasso)
```



Up until now, the output we'd been returning from the `predict()` function had all been on 0 to 1 probability scale (i.e. the probability the model predict of an observation being a 'ever smoker') because we'd used the `type = "response"` argument.

However, `predict()` can alternately return the class that the model predicts is most probable by changing the `type` argument to be `type = "class"`.

```
pred.lasso <- predict(fit.lasso, newx = X[-in.train,],
                      s = "lambda.min",
                      type = "class")
```

This is useful to be able to use an additional function called `confusionMatrix()` in the `caret` package that allows us to more carefully compare how our predictions correspond to the truth and what errors we're experiencing

```
caret::confusionMatrix(as.factor(pred.lasso), as.factor(testing$ever.smoke))
```

```
> Confusion Matrix and Statistics
>
>           Reference
> Prediction  0  1
>          0 31 13
>          1 13 59
>
>                Accuracy : 0.7759
>                  95% CI : (0.6891, 0.8481)
>     No Information Rate : 0.6207
>     P-Value [Acc > NIR] : 0.0002663
>
>                   Kappa : 0.524
>  Mcnemar's Test P-Value : 1.0000000
>
>             Sensitivity : 0.7045
>             Specificity : 0.8194
>          Pos Pred Value : 0.7045
>          Neg Pred Value : 0.8194
>              Prevalence : 0.3793
>          Detection Rate : 0.2672
>    Detection Prevalence : 0.3793
>       Balanced Accuracy : 0.7620
>
>        'Positive' Class : 0
>
```

Caret model training

While any model can conceivably be trained/tested in our partitioned datasets, we need to know the R-package that distributes that model and the potentially idiosyncratic syntax for fitting that particular model.

However, the `caret` package's function `train()` is a wrapper the conveniently facilitates a common syntax for implementing many common machine learning/prediction models. This makes it easier to quickly deploy a diverse set of candidate models with less hassle required to develop model/package-specific code.

For example, we can perform a rather basic call to `train()` in order to fit a RandomForest model from the `ranger` package:

```r
set.seed(825)
fit.rf <- caret::train(y = as.factor(training$ever.smoke),
                x = X[in.train,],
                method = "ranger") # 2.2 minutes
```

Fitting our training model with this approach still allows us to predict and evaluate model fits in the testing dataset in the ways we've learned so far:

```r
pred.rf <- predict(fit.rf, newdata = X[-in.train,])
confusionMatrix(pred.rf, as.factor(testing$ever.smoke))
```

```
> Confusion Matrix and Statistics
>
>           Reference
> Prediction  0  1
>          0 35 13
>          1  9 59
>
>                Accuracy : 0.8103
>                  95% CI : (0.7271, 0.8772)
>     No Information Rate : 0.6207
>     P-Value [Acc > NIR] : 8.136e-06
>
>                   Kappa : 0.6042
>  Mcnemar's Test P-Value : 0.5224
>
>             Sensitivity : 0.7955
>             Specificity : 0.8194
>          Pos Pred Value : 0.7292
>          Neg Pred Value : 0.8676
>              Prevalence : 0.3793
>          Detection Rate : 0.3017
>    Detection Prevalence : 0.4138
>       Balanced Accuracy : 0.8074
>
>        'Positive' Class : 0
>
```

Conveniently however, we can train a host of different models by only modifying the `method` argument in `train()`. Thus, deciding we would additionally like to fit a Support Vector Machines model from the `kernlab` package only requires updating our method to `method = "svmRadial"`

```r
set.seed(637)
fit.svm <- caret::train(y = as.factor(training$ever.smoke),
                x = X[in.train,],
                method = "svmRadial") ## 1.6 minutes
```

```r
pred.svm <- predict(fit.svm, newdata = X[-in.train,])
confusionMatrix(pred.svm, as.factor(testing$ever.smoke))
```

```
> Confusion Matrix and Statistics
>
>           Reference
> Prediction  0  1
>          0 32 15
>          1 12 57
```

```
>
>                Accuracy : 0.7672
>                  95% CI : (0.6797, 0.8407)
>     No Information Rate : 0.6207
>     P-Value [Acc > NIR] : 0.0005646
>
>                   Kappa : 0.5121
>  Mcnemar's Test P-Value : 0.7003114
>
>             Sensitivity : 0.7273
>             Specificity : 0.7917
>          Pos Pred Value : 0.6809
>          Neg Pred Value : 0.8261
>              Prevalence : 0.3793
>          Detection Rate : 0.2759
>    Detection Prevalence : 0.4052
>       Balanced Accuracy : 0.7595
>
>        'Positive' Class : 0
>
```

An extremely diverse selection of models is available through `train()` and details on the implementation of each is listed on the `caret` website http://topepo.github.io/caret/train-models-by-tag.html

```
## list names of all caret models
names(getModelInfo())
```

```
>   [1] "ada"                "AdaBag"             "AdaBoost.M1"
>   [4] "adaboost"           "amdai"              "ANFIS"
>   [7] "avNNet"             "awnb"               "awtan"
>  [10] "bag"                "bagEarth"           "bagEarthGCV"
>  [13] "bagFDA"             "bagFDAGCV"          "bam"
>  [16] "bartMachine"        "bayesglm"           "binda"
>  [19] "blackboost"         "blasso"             "blassoAveraged"
>  [22] "bridge"             "brnn"               "BstLm"
>  [25] "bstSm"              "bstTree"            "C5.0"
>  [28] "C5.0Cost"           "C5.0Rules"          "C5.0Tree"
>  [31] "cforest"            "chaid"              "CSimca"
>  [34] "ctree"              "ctree2"             "cubist"
>  [37] "dda"                "deepboost"          "DENFIS"
>  [40] "dnn"                "dwdLinear"          "dwdPoly"
>  [43] "dwdRadial"          "earth"              "elm"
>  [46] "enet"               "evtree"             "extraTrees"
>  [49] "fda"                "FH.GBML"            "FIR.DM"
>  [52] "foba"               "FRBCS.CHI"          "FRBCS.W"
>  [55] "FS.HGD"             "gam"                "gamboost"
>  [58] "gamLoess"           "gamSpline"          "gaussprLinear"
>  [61] "gaussprPoly"        "gaussprRadial"      "gbm_h2o"
>  [64] "gbm"                "gcvEarth"           "GFS.FR.MOGUL"
>  [67] "GFS.LT.RS"          "GFS.THRIFT"         "glm.nb"
>  [70] "glm"                "glmboost"           "glmnet_h2o"
>  [73] "glmnet"             "glmStepAIC"         "gpls"
>  [76] "hda"                "hdda"               "hdrda"
>  [79] "HYFIS"              "icr"                "J48"
>  [82] "JRip"               "kernelpls"          "kknn"
```

```
>  [85] "knn"                  "krlsPoly"             "krlsRadial"
>  [88] "lars"                 "lars2"                "lasso"
>  [91] "lda"                  "lda2"                 "leapBackward"
>  [94] "leapForward"          "leapSeq"              "Linda"
>  [97] "lm"                   "lmStepAIC"            "LMT"
> [100] "loclda"               "logicBag"             "LogitBoost"
> [103] "logreg"               "lssvmLinear"          "lssvmPoly"
> [106] "lssvmRadial"          "lvq"                  "M5"
> [109] "M5Rules"              "manb"                 "mda"
> [112] "Mlda"                 "mlp"                  "mlpKerasDecay"
> [115] "mlpKerasDecayCost"    "mlpKerasDropout"      "mlpKerasDropoutCost"
> [118] "mlpML"                "mlpSGD"               "mlpWeightDecay"
> [121] "mlpWeightDecayML"     "monmlp"               "msaenet"
> [124] "multinom"             "mxnet"                "mxnetAdam"
> [127] "naive_bayes"          "nb"                   "nbDiscrete"
> [130] "nbSearch"             "neuralnet"            "nnet"
> [133] "nnls"                 "nodeHarvest"          "null"
> [136] "OneR"                 "ordinalNet"           "ORFlog"
> [139] "ORFpls"               "ORFridge"             "ORFsvm"
> [142] "ownn"                 "pam"                  "parRF"
> [145] "PART"                 "partDSA"              "pcaNNet"
> [148] "pcr"                  "pda"                  "pda2"
> [151] "penalized"            "PenalizedLDA"         "plr"
> [154] "pls"                  "plsRglm"              "polr"
> [157] "ppr"                  "PRIM"                 "protoclass"
> [160] "pythonKnnReg"         "qda"                  "QdaCov"
> [163] "qrf"                  "qrnn"                 "randomGLM"
> [166] "ranger"               "rbf"                  "rbfDDA"
> [169] "Rborist"              "rda"                  "regLogistic"
> [172] "relaxo"               "rf"                   "rFerns"
> [175] "RFlda"                "rfRules"              "ridge"
> [178] "rlda"                 "rlm"                  "rmda"
> [181] "rocc"                 "rotationForest"       "rotationForestCp"
> [184] "rpart"                "rpart1SE"             "rpart2"
> [187] "rpartCost"            "rpartScore"           "rqlasso"
> [190] "rqnc"                 "RRF"                  "RRFglobal"
> [193] "rrlda"                "RSimca"               "rvmLinear"
> [196] "rvmPoly"              "rvmRadial"            "SBC"
> [199] "sda"                  "sdwd"                 "simpls"
> [202] "SLAVE"                "slda"                 "smda"
> [205] "snn"                  "sparseLDA"            "spikeslab"
> [208] "spls"                 "stepLDA"              "stepQDA"
> [211] "superpc"              "svmBoundrangeString"  "svmExpoString"
> [214] "svmLinear"            "svmLinear2"           "svmLinear3"
> [217] "svmLinearWeights"     "svmLinearWeights2"    "svmPoly"
> [220] "svmRadial"            "svmRadialCost"        "svmRadialSigma"
> [223] "svmRadialWeights"     "svmSpectrumString"    "tan"
> [226] "tanSearch"            "treebag"              "vbmpRadial"
> [229] "vglmAdjCat"           "vglmContRatio"        "vglmCumulative"
> [232] "widekernelpls"        "WM"                   "wsrf"
> [235] "xgbDART"              "xgbLinear"            "xgbTree"
> [238] "xyf"
```

It should additionally be noted that the `train()` function has powerful functionality beyond simply providing

convenient code for calling a large variety of different models. However, these will be explored in great depth later in the workshop.

Be patient young padawan!