Caret introudction

Koji Mizumura

September 6 - October 22, 2018

Table of Contents

Reference:  
[Bookdown](http://topepo.github.io/caret/index.html)  
[video](https://www.youtube.com/watch?v=z8PRU46I3NY)  
[Jap](https://logics-of-blue.com/r%E3%81%AB%E3%82%88%E3%82%8B%E6%A9%9F%E6%A2%B0%E5%AD%A6%E7%BF%92%EF%BC%9Acaret%E3%83%91%E3%83%83%E3%82%B1%E3%83%BC%E3%82%B8%E3%81%AE%E4%BD%BF%E3%81%84%E6%96%B9/)

# Set-up

# Visualizations

The featurePlot function is a wrapper for different lattice plots to visualize the data. For example, the following figures show the default plot fo continuous outcomes generated using the featureplot function.

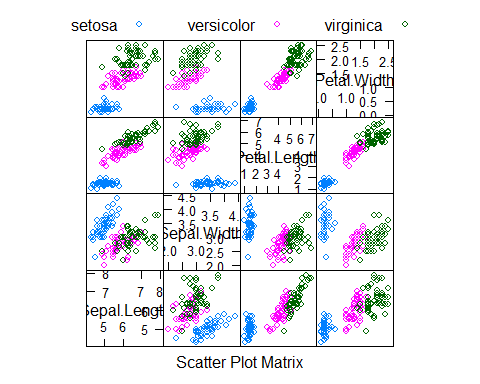
For classification data sets, the iris data are used for illusion.

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 ...

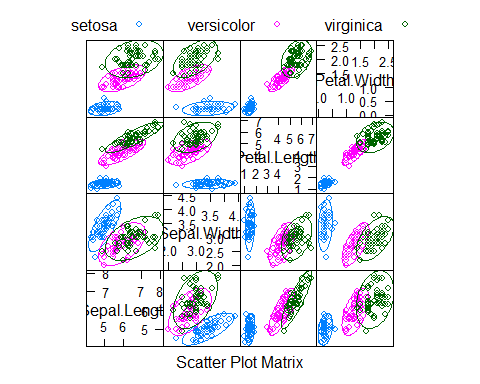
## Scatterplot matrix

featurePlot(x = iris[, 1:4],   
 y = iris$Species,   
 plot = "pairs",  
 ## Add a key at the top  
 auto.key = list(columns = 3))



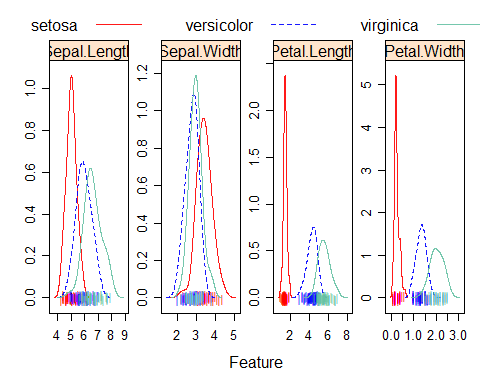
## Scatterplot matrix with ellipses

featurePlot(x=iris[,1:4],  
 y=iris$Species,  
 plot="ellipse",  
 ## Add a key at the topc  
 auto.key=list(columns=3))



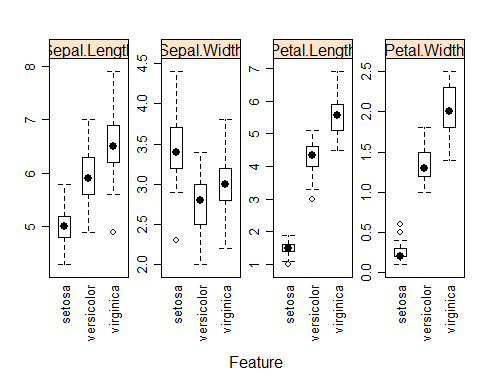
## Overlayed density plots

transparentTheme(trans = .9)  
featurePlot(x = iris[, 1:4],   
 y = iris$Species,  
 plot = "density",   
 ## Pass in options to xyplot() to   
 ## make it prettier  
 scales = list(x = list(relation="free"),   
 y = list(relation="free")),   
 adjust = 1.5,  
 pch = "|",  
 layout = c(4, 1),  
 auto.key = list(columns = 3))



## Box Plots

featurePlot(x=iris[,1:4],  
 y=iris$Species,  
 plot="box",  
 ## Pass in options to bwplot()  
 scales=list(y=list(relation="free"),  
 x=list(rot=90)),  
 layout=c(4,1),  
 auto.key=list(columns=2))



## Scatter Plots

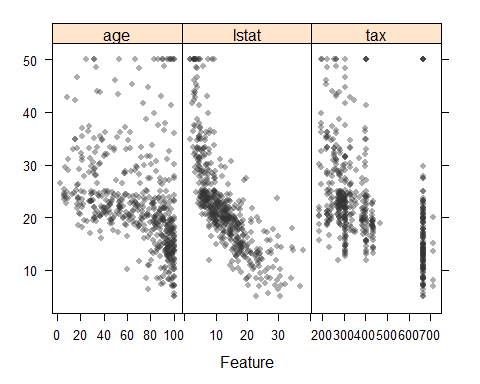
For regresson, the Boston Housing data is used:

# install.packages("mlbench")  
  
library(mlbench)  
data(BostonHousing)  
  
regVar <- c("age","lstat","tax")  
str(BostonHousing[,regVar])

## 'data.frame': 506 obs. of 3 variables:  
## $ age : num 65.2 78.9 61.1 45.8 54.2 58.7 66.6 96.1 100 85.9 ...  
## $ lstat: num 4.98 9.14 4.03 2.94 5.33 ...  
## $ tax : num 296 242 242 222 222 222 311 311 311 311 ...

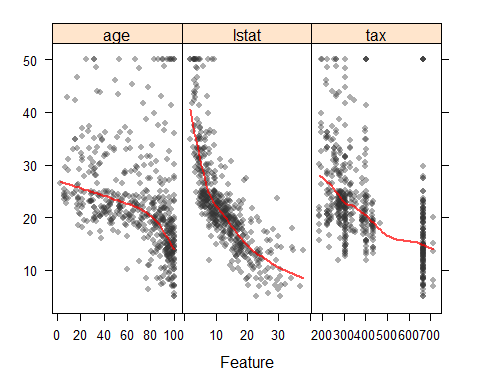
When the predictors are continuous, featurePlot can be used to create scatter plots of each of the predictors with the outcome. For example:

theme1 <- trellis.par.get()  
theme1$plot.symbol$col=rgb(.2,.2,.2,.4)  
theme1$plot.symbol$pch = 16  
theme1$plot.line$col = rgb(1, 0, 0, .7)  
theme1$plot.line$lwd <- 2  
trellis.par.set(theme1)  
  
featurePlot(x=BostonHousing[,regVar],  
 y=BostonHousing$medv,  
 plot="scatter",  
 layout=c(3,1))



Note that the x-axis scales are different. The function automatically uses scales = list(y = list(relation = "free")) so you don’t have to add it. We can also pass in options to the \*\*lattice\*\* function xyplot. For example, we can add a scatter plot smoother by passing in new options:

trellis.par.set(theme1)  
  
  
featurePlot(x = BostonHousing[, regVar],   
 y = BostonHousing$medv,   
 plot = "scatter",  
 type = c("p", "smooth"),  
 span = .5,  
 layout = c(3, 1))



The options degree and span control the smoothness of the smoother.

# Chapter 3. Pre-processing

There are multiple techniques for data pre-processing.

*Creating Dummy Variables* Zero- and Near Zero-Variance Predictors *Identifying Correlated Predictors* Linear Dependencies *The preProcess Function* Centering and Scaling *Imputation* Transforming Predictors *Putting It All Together* Class Distance Calculations

caret includes several functions to pre-process the predictor data. It assumes that all of the data are numeric(i.e., factors have been converted to dummy variables via model.matrix, dummyVars or other means).

note that the later chapter on using [recipes](https://tidymodels.github.io/recipes/) with train shows how that approach canoffer a more diverse and customizable interface to pre-process in the package.

## Creating dummy variables

The function dummyVars can be used to generate a compete (less than full rank parameterized) set of dummy variables from one or more factors. The function takes a fomula and a dataset and ouputs an object that can be used to create the dummy variables using theb predict method.

For example, the etitanic dataset in the earth package includes two factors: pclass(passenger class with levels 1st,2nd and 3rd) and sex (with levels female, male). The base R function model.matrix would generate th following variables:

data(etitanic)  
head(etitanic)

## pclass survived sex age sibsp parch  
## 1 1st 1 female 29.0000 0 0  
## 2 1st 1 male 0.9167 1 2  
## 3 1st 0 female 2.0000 1 2  
## 4 1st 0 male 30.0000 1 2  
## 5 1st 0 female 25.0000 1 2  
## 6 1st 1 male 48.0000 0 0

model.matrix(survived~.,data=etitanic) %>%   
 head()

## (Intercept) pclass2nd pclass3rd sexmale age sibsp parch  
## 1 1 0 0 0 29.0000 0 0  
## 2 1 0 0 1 0.9167 1 2  
## 3 1 0 0 0 2.0000 1 2  
## 4 1 0 0 1 30.0000 1 2  
## 5 1 0 0 0 25.0000 1 2  
## 6 1 0 0 1 48.0000 0 0

Using dummyVars:

dummies <- dummyVars(survived~.,data=etitanic)  
predict(dummies,newdata=etitanic) %>%   
 head()

## pclass.1st pclass.2nd pclass.3rd sex.female sex.male age sibsp parch  
## 1 1 0 0 1 0 29.0000 0 0  
## 2 1 0 0 0 1 0.9167 1 2  
## 3 1 0 0 1 0 2.0000 1 2  
## 4 1 0 0 0 1 30.0000 1 2  
## 5 1 0 0 1 0 25.0000 1 2  
## 6 1 0 0 0 1 48.0000 0 0

Now there is no intercept and each factor has a dummy variable for each level, so this parameterization may not be useful for some model functions, such as lm.

## Zero- and Near Zero-Variance Predictors

In some situations, the data generating mechanism can create predictors that only have a single unique value (ie. a **“zero-variance-predictor”**). For many models (excluding tree-based models), this may cause the model to crash or the fit to be unstable.

Similarly, predictors might have only a handful of unique values that occur with very low frequencies. For example, in the drug resistance data, the nR11 descritor (number of 11-membered rings) data have a few unique numeric values that are highly unbalanced:

data(mdrr)  
  
# Multidrug Resistance Reversal (MDRR) Agent Data  
  
mdrrDescr$nR11 %>%   
 table() %>%   
 data.frame()

## . Freq  
## 1 0 501  
## 2 1 4  
## 3 2 23

The concern here that these predictors may become zero-variance predictors when the data are split into cross-validation/bootstrap sub-samples or that a few samples may have an undue influence on the model. These “near-zero-variance” predictors may need to be identified and eliminated prior to modeling.

To identify these types of predictors, the following two metrics can be calculated: \* the frequency of the most prevalent value over the second most frequent value (called the “frequency ratio’’), which would be near one for well-behaved predictors and very large for highly-unbalanced data and \* the “percent of unique values’’ is the number of unique values divided by the total number of samples (times 100) that approaches zero as the granularity of the data increases

If the frequency ratio is greater than a pre-specified threshold and the unique value percentage is less than a threshold, we might consider a predictor to be near zero-variance.

We would not want to falsely identify data that have low **granularity** but are evenly distributed, such as data from a discrete uniform distribution. Using both criteria should not falsely detect such predictors.

Looking at the MDRR data, the nearZeroVar function can be used to identify near zero-variance variables (the saveMetrics argument can be used to show the details and usually defaults to FALSE):

nzv <- nearZeroVar(mdrrDescr,saveMetrics = T)  
nzv[nzv$nzv,][1:10,]

## freqRatio percentUnique zeroVar nzv  
## nTB 23.00000 0.3787879 FALSE TRUE  
## nBR 131.00000 0.3787879 FALSE TRUE  
## nI 527.00000 0.3787879 FALSE TRUE  
## nR03 527.00000 0.3787879 FALSE TRUE  
## nR08 527.00000 0.3787879 FALSE TRUE  
## nR11 21.78261 0.5681818 FALSE TRUE  
## nR12 57.66667 0.3787879 FALSE TRUE  
## D.Dr03 527.00000 0.3787879 FALSE TRUE  
## D.Dr07 123.50000 5.8712121 FALSE TRUE  
## D.Dr08 527.00000 0.3787879 FALSE TRUE

dim(mdrrDescr)

## [1] 528 342

nzv <- nearZeroVar(mdrrDescr)  
filteredDescr <- mdrrDescr[, -nzv]  
dim(filteredDescr)

## [1] 528 297

## Identifying correlated predictors

While there are some models that thriveon correlated predictors (such as pls), other models may benefit from reducing the level of correlation between predictors.

Given a correlation matrix, the findCorrelation function uses the following algorithm to flag predictors for removal:

descCor <- cor(filteredDescr)  
highCorr <- sum(abs(descCor[upper.tri(descCor)])>.999)  
  
highCorr

## [1] 65

For the previous MDRR data, there are 65 descriptors that are almost perfectly correlated (|correlation| > 0.999), such as the total information index of atomic composition (IAC) and the total information content index (neighborhood symmetry of 0-order) (TIC0) (correlation = 1). The code chunk below shows the effect of removing descriptors with absolute correlations above 0.75.

descCor <- cor(filteredDescr)  
summary(descCor[upper.tri(descCor)])

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## -0.99607 -0.05373 0.25006 0.26078 0.65527 1.00000

m2 <- matrix(1:20, 4, 5)  
lower.tri(m2)

## [,1] [,2] [,3] [,4] [,5]  
## [1,] FALSE FALSE FALSE FALSE FALSE  
## [2,] TRUE FALSE FALSE FALSE FALSE  
## [3,] TRUE TRUE FALSE FALSE FALSE  
## [4,] TRUE TRUE TRUE FALSE FALSE

m2[lower.tri(m2)] <- NA  
m2

## [,1] [,2] [,3] [,4] [,5]  
## [1,] 1 5 9 13 17  
## [2,] NA 6 10 14 18  
## [3,] NA NA 11 15 19  
## [4,] NA NA NA 16 20

## Linear dependencies

The function findLinearCombos uses the QR decomposition of a matrix to enumerate sets of linear combinations (if they exist). For example, consider the following matrix that could have produced by a less-than-full-rank parameterizations of a two-way experimental layout:

ltfrDesign <- matrix(0, nrow=6, ncol=6)  
ltfrDesign[,1] <- c(1, 1, 1, 1, 1, 1)  
ltfrDesign[,2] <- c(1, 1, 1, 0, 0, 0)  
ltfrDesign[,3] <- c(0, 0, 0, 1, 1, 1)  
ltfrDesign[,4] <- c(1, 0, 0, 1, 0, 0)  
ltfrDesign[,5] <- c(0, 1, 0, 0, 1, 0)  
ltfrDesign[,6] <- c(0, 0, 1, 0, 0, 1)  
  
ltfrDesign

## [,1] [,2] [,3] [,4] [,5] [,6]  
## [1,] 1 1 0 1 0 0  
## [2,] 1 1 0 0 1 0  
## [3,] 1 1 0 0 0 1  
## [4,] 1 0 1 1 0 0  
## [5,] 1 0 1 0 1 0  
## [6,] 1 0 1 0 0 1

Note that columns two and three add up to the first column. Similarly, columns four, five and six add up the first column. findLinearCombos will return a list that enumerates these dependencies. For each linear combination, it will incrementally remove columns from the matrix and test to see if the dependencies have been resolved. findLinearCombos will also return a vector of column positions can be removed to eliminate the linear dependencies:

comboInfo <- findLinearCombos(ltfrDesign)  
comboInfo

## $linearCombos  
## $linearCombos[[1]]  
## [1] 3 1 2  
##   
## $linearCombos[[2]]  
## [1] 6 1 4 5  
##   
##   
## $remove  
## [1] 3 6

ltfrDesign[,-comboInfo$remove]

## [,1] [,2] [,3] [,4]  
## [1,] 1 1 1 0  
## [2,] 1 1 0 1  
## [3,] 1 1 0 0  
## [4,] 1 0 1 0  
## [5,] 1 0 0 1  
## [6,] 1 0 0 0

These types of dependencies can arise when large numbers of binary chemial fingerprints are used to describe the structure of a molecule.

## 3.5 The preProcess Function

The preProcess class can be used for many operations on predictors, including *centering* and *scaling*. The function preProcess estimates the required parameters for each operation and predict.preProcess is used to apply them to specific data sets. This function can also be interfaces when calling the train function.

Several types of techniques are described in the next few sections and then another example is used to demonstrate how multiple methods can be used. Note that, in all cases, the preProcess function estimates whatever it requires from a specific data set (e.g. the training set) and then applies these transformations to any data set without recomputing the values

## Centering and scaling

In the example below, the half of the MDRR data are used to estimate the location and scale of the predictors. The function preProcess doesn’t actually pre-process the data. predict.preProcess is used to pre-process this and other data sets.

set.seed(96)  
inTrain <- sample(seq(along=mdrrClass),length(mdrrClass)/2)  
  
training <- filteredDescr[inTrain,]  
test <- filteredDescr[-inTrain,]  
trainMDRR <- mdrrClass[inTrain]  
testMDRR <- mdrrClass[-inTrain]  
  
preProcValues <- preProcess(training,method=c("center","scale"))  
  
trainTrainsformed <- predict(preProcValues,training)  
testTransformed <- predict(preProcValues,test)

The preProcess option "range scales the data to the interval between zero and one.

## Imputation

preProcess can be used to impute data sets based only on information in the training set. One method of doing this is with K-nearest neighbors. For an arbitrary sample, the K closest neighbors are found in the training set and the value for the predictor is imputed using these values (e.g. using the mean). Using this approach will automatically trigger preProcess to center and scale the data, regardless of what is in the method argument. Alternatively, bagged trees can also be used to impute. For each predictor in the data, a bagged tree is created using all of the other predictors in the training set. When a new sample has a missing predictor value, the bagged model is used to predict the value. While, in theory, this is a more powerful method of imputing, the computational costs are much higher than the nearest neighbor technique.

## Transforming predictors

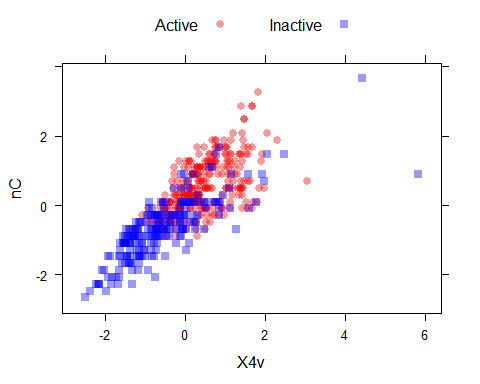
In some cases, there is a need to use **principal component analysis (PCA)** to transform the data to a smaller sub–space where the new variable are uncorrelated with one another. The preProcess class can apply this transformation by including "pca" in the method argument. Doing this will also force scaling of the predictors. Note that when PCA is requested, predict.preProcess changes the column names to PC1, PC2 and so on.

Similarly, **independent component analysis (ICA)** can also be used to find new variables that are linear combinations of the original set such that the components are independent (as opposed to uncorrelated in PCA). The new variables will be labeled as IC1, IC2 and so on.

The “spatial sign” transformation [Serneels et al, 2006](https://pubs.acs.org/doi/abs/10.1021/ci050498u) projects the data for a preidctor to the unit circle in p dimensions, where p is the number of predictors. Essentially, a vector of data is divided by its norm.

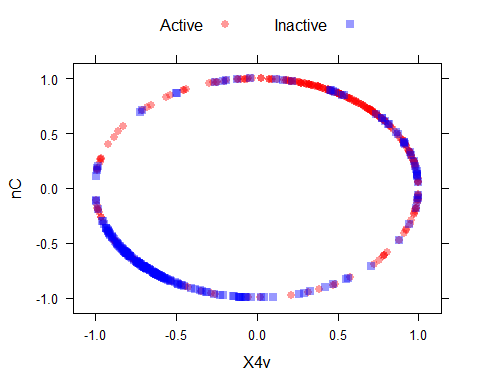
The two figures below show two centered and scaled discritors from the MDRR data before and after the spatial sign transformation. The predictors should be centered and scaled before applyin this transformation.

library(AppliedPredictiveModeling)  
transparentTheme(trans=.4)  
  
plotSubset <- data.frame(scale(mdrrDescr[, c("nC", "X4v")]))   
xyplot(nC ~ X4v,  
 data = plotSubset,  
 groups = mdrrClass,   
 auto.key = list(columns = 2))



After the spatial sign:

transformed <- spatialSign(plotSubset)  
transformed <- as.data.frame(transformed)  
  
xyplot(nC~X4v,  
 data=transformed,  
 groups=mdrrClass,  
 auto.key=list(columns=2))



Another option, "BoxCox" will estimate a **Box–Cox transformation** on the predictors if the data are greater than zero.

preProcValues2 <- preProcess(training,method="BoxCox")  
trainBC <- predict(preProcValues2,training)  
testBC <- predict(preProcValues2,test)  
preProcValues2

## Created from 264 samples and 258 variables  
##   
## Pre-processing:  
## - Box-Cox transformation (258)  
## - ignored (0)  
##   
## Lambda estimates for Box-Cox transformation:  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## -2.000 0.200 0.500 0.495 0.900 2.000

The NA values correspond to the predictors that could not be transformed. This transformation requires the data to be greater than zero. Two similar transformations, the Yeo-Johnson and exponential transformation of Manly (1976) can also be used in preProcess.

## Putting it all together

In *Applied PRedictive Modeling* threre is a case study where the execution times of jobs in a high performance computing environment are being predicted. The data are:

library(AppliedPredictiveModeling)  
data(schedulingData)  
str(schedulingData)

## 'data.frame': 4331 obs. of 8 variables:  
## $ Protocol : Factor w/ 14 levels "A","C","D","E",..: 4 4 4 4 4 4 4 4 4 4 ...  
## $ Compounds : num 997 97 101 93 100 100 105 98 101 95 ...  
## $ InputFields: num 137 103 75 76 82 82 88 95 91 92 ...  
## $ Iterations : num 20 20 10 20 20 20 20 20 20 20 ...  
## $ NumPending : num 0 0 0 0 0 0 0 0 0 0 ...  
## $ Hour : num 14 13.8 13.8 10.1 10.4 ...  
## $ Day : Factor w/ 7 levels "Mon","Tue","Wed",..: 2 2 4 5 5 3 5 5 5 3 ...  
## $ Class : Factor w/ 4 levels "VF","F","M","L": 2 1 1 1 1 1 1 1 1 1 ...

The data are a mix of categorical and numeric predictors. Suppose we want to use the Yeo-Johnson transformation on the continuous predictors then center and scale them. Let’s also suppose that we will be running a tree-based models so we might want to keep the factors as factors (as opposed to creating dummy variables). We run the function on all the columns except the last, which is the outcome.

pp\_hpc <- preProcess(schedulingData[,-8],  
 method=c("center","scale","YeoJohnson"))  
pp\_hpc

## Created from 4331 samples and 7 variables  
##   
## Pre-processing:  
## - centered (5)  
## - ignored (2)  
## - scaled (5)  
## - Yeo-Johnson transformation (5)  
##   
## Lambda estimates for Yeo-Johnson transformation:  
## -0.08, -0.03, -1.05, -1.1, 1.44

transformed <- predict(pp\_hpc,newdata=schedulingData[,-8])  
head(transformed)

## Protocol Compounds InputFields Iterations NumPending Hour Day  
## 1 E 1.2289592 -0.6324580 -0.0615593 -0.554123 0.004586516 Tue  
## 2 E -0.6065826 -0.8120473 -0.0615593 -0.554123 -0.043733201 Tue  
## 3 E -0.5719534 -1.0131504 -2.7894869 -0.554123 -0.034967177 Thu  
## 4 E -0.6427737 -1.0047277 -0.0615593 -0.554123 -0.964170752 Fri  
## 5 E -0.5804713 -0.9564504 -0.0615593 -0.554123 -0.902085020 Fri  
## 6 E -0.5804713 -0.9564504 -0.0615593 -0.554123 0.698108782 Wed

The two predictors labeled as “ignored” in the output are the two factor predictors. These are not altered but the numeric predictors are transformed. However, the predictor for the number of pending jobs, has a very sparse and unbalanced distribution:

mean(schedulingData$NumPending==0)

## [1] 0.7561764

For some other models, this might be an issue (especially if we resample or down-sample the data). We can add a filter to check for zero- or near zero-variance predictors prior to running the processing calculations:

schedulingData %>%   
 dplyr::select(8) %>%   
 head()

## Class  
## 1 F  
## 2 VF  
## 3 VF  
## 4 VF  
## 5 VF  
## 6 VF

pp\_no\_nzv <- preProcess(schedulingData[,-8],  
 method=c("center","scale","YeoJohnson","nzv"))  
pp\_no\_nzv

## Created from 4331 samples and 7 variables  
##   
## Pre-processing:  
## - centered (4)  
## - ignored (2)  
## - removed (1)  
## - scaled (4)  
## - Yeo-Johnson transformation (4)  
##   
## Lambda estimates for Yeo-Johnson transformation:  
## -0.08, -0.03, -1.05, 1.44

schedulingData[1:6,-8]

## Protocol Compounds InputFields Iterations NumPending Hour Day  
## 1 E 997 137 20 0 14.00000 Tue  
## 2 E 97 103 20 0 13.81667 Tue  
## 3 E 101 75 10 0 13.85000 Thu  
## 4 E 93 76 20 0 10.10000 Fri  
## 5 E 100 82 20 0 10.36667 Fri  
## 6 E 100 82 20 0 16.53333 Wed

predict(pp\_no\_nzv,newdata = schedulingData[1:6,-8])

## Protocol Compounds InputFields Iterations Hour Day  
## 1 E 1.2289592 -0.6324580 -0.0615593 0.004586516 Tue  
## 2 E -0.6065826 -0.8120473 -0.0615593 -0.043733201 Tue  
## 3 E -0.5719534 -1.0131504 -2.7894869 -0.034967177 Thu  
## 4 E -0.6427737 -1.0047277 -0.0615593 -0.964170752 Fri  
## 5 E -0.5804713 -0.9564504 -0.0615593 -0.902085020 Fri  
## 6 E -0.5804713 -0.9564504 -0.0615593 0.698108782 Wed

Note that one predictor is labeled as “removed” and the processed data lack the sparse predictor.

## 3.10 Class distance calculations

caret contains functions to generate new predictors variables based on distances to class centroids (similar to how linear discriminant analysis works). For each level of a factor variable, the class centroid and covariance matrix is calculated.

For new samples, the Mahalanobis distance to each of the class centroids is computed and can be used as an additional predictor. This can be helpful for non–linear models when the true decision boundary is actually linear.

In cases where there are more predictors within a class than samples, the classDist function has arguments called pca and keep arguments that allow for principal components analysis within each class to be used to avoid issues with singular covariance matrices.

predict.classDist is then used to generate the class distances. By default, the distances are logged, but this can be changed via the trans argument to predict.classDist.

As an example, we can used the MDRR data.

centroids <- classDist(trainBC, trainMDRR)  
distances <- predict(centroids, testBC)  
distances <- as.data.frame(distances)  
head(distances)

This image shows a scatterplot matrix of the class distances for the held-out samples:

xyplot(dist.Active ~ dist.Inactive,  
 data = distances,   
 groups = testMDRR,   
 auto.key = list(columns = 2))

# Chapter 4: Data splitting

Contents *Simple Splitting Based on the Outcome* Splitting Based on the Predictors *Data Splitting for Time Series* Data Splitting with Important Groups

## Simple splitting based on the outcome

The function createDataPartition can be used to create balanced splits of the data. If the y argument to this function is a factor, the random sampling occurs which each class and should preserve the overall class distribution of the data. For example, to create a single 80/20% split of the iris data:

library(caret)  
set.seed(3456)  
trainIndex <- createDataPartition(iris$Species,p=.8,  
 list=FALSE,  
 times=1)  
head(trainIndex)

## Resample1  
## [1,] 1  
## [2,] 2  
## [3,] 4  
## [4,] 5  
## [5,] 6  
## [6,] 8

irisTrain <- iris[trainIndex,]  
irisTest <- iris[-trainIndex,]

The list = FALSE avoids returning the data as a list. This function also has an argument, times, that can create multiple splits at once; the data indices are returned in a list of integer vectors. Similarly, createResample can be used to make simple bootstrap samples and createFolds can be used to generate balanced cross–validation groupings from a set of data.

## Splitting based on the predictors

Also, the function maxDissim can be used to create sub-samples using a maximum dissimilarity approach. Suppose there is a dataset with samples and a larger data set with samples. We may want to create a sub-sample from that is diverse when compared to . The most dissimilar point in is-added to and the process continues.

There are many methods in R to calculate dissimilarity. `caret uses the proxy package. See the manual for the package for a list of available measures. Also, there are many ways to claculate which sample is “most similar”. The argument obj can be used to specify any function that returns a scaler measure. caret includes two functions, minDiss and sumDiss, that can be used to maximize the minimum and total dissimilarities respectively.

As an example, the figure below shows a scatter plot of two chemical descriptors for the Cox2 data. Using an initial random sample of 5 compounds, we can select 20 more compounds from the data so that the new compounds are most dissimilar from the initial 5 that were specified.

The panesl in the figure show that results using several combinations of distance metrics and scoring functions. For these data, the distance measure has less of an impact thant the scoring method for determining which compounds are most dissimilar.

library(mlbench)  
data(BostonHousing)  
  
testing <- scale(BostonHousing[,c("age","nox")])  
set.seed(5)  
  
  
## A random sample of 5 data points  
startSet <- sample(1:dim(testing)[1],5)  
samplePool <- testing[-startSet,]  
start <- testing[startSet,]  
# newSamp <- maxDissim(start, samplePool, n = 20)  
# head(newSamp)

The visualization below shows the data set (small points), the starting samples (larger blue points) and the order in which the other 20 samples are added.

## Data splitting for time series

Simple random sampling of time series is probably not the best way to resample time series data. [Hyndman and Athanasopoulos (2013)](https://www.otexts.org/fpp/2/5) discuss *rolling forecasting origin* techniques that move the training and test sets in a time. caret contains a function called createTimeSlices that can create the indices for this type of splitting.

The three parameters for this type of splitting are: - initialWindow: the initial number of consecutive values in each training set sample - horizon: The number of consecutive values in test set sample - fixedWindow: A logical: if FALSE, the training set always start at the first sample and the training set size will vary over data splits.

As an exampl,e suppose we have a time series with 20 data points. We can fixe initialWindow=5 and look at different settings of the other two arguments. In the plot below, rows in each panel correspond to different data splits (i.e. resamples) and the columns correspond to different data points. Also, red indicates samples that are in included in the training set and the blue indicates samples in the test set.

## Simple splitting with important groups

In some cases there is an important qualitative factor in the data that should be considered during (re)sampling. For example:

* in clinical trials, there may be hospital-to-hospital differences
* with longitudinal or repeated measures data, subjects (or general independent experimental unit) may have multiple rows in the data set, etc.

There may be an interest in making sure that these groups are not contained in the training and testing set since this may bias the test set performance to be more optimistic. Also, when one or more specific groups are held out, the resampling might capture the “ruggedness” of the model. In the example where clinical data is recorded over multiple sites, the resampling performance estimates partly measure how extensible the model is across sites.

To split the data based on groups, groupKFold can be used:

set.seed(3527)  
subjects <- sample(1:20,size=80,replace=TRUE)  
table(subjects)

## subjects  
## 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20   
## 3 5 4 3 2 6 4 6 3 5 5 3 5 6 4 4 7 1 3 1

folds <- groupKFold(subjects,k=15)

The results in folds can be used as an inputs into the index argument of the trainControl function. This plot shows how each subject is partitioned between the modeling and holdout sets. Note that since k was less than 20 when folds was created, there are some k was less than 20 when folds was created, there are some holdouts with model than one subject.

# Model training and tuning

<http://topepo.github.io/caret/model-training-and-tuning.html>

## Model Training and Parameter Tuning

The caret package has several functions that attempt to streamline the model building and evaluation process.

The train function can be used to - evaluate, using **resampling**, the effect of model tuning parameters on performance - choose the optimal model across these parameters - estimate model performance from a training set

First, a specific model must be chosen. Currently, 237 are available using caret; see train [model list](http://topepo.github.io/caret/available-models.html) or train [Model by Tag](http://topepo.github.io/caret/train-models-by-tag.html) for details.

On thrse pages, there are lists of tuning parameters that can potentially be optimized. User-defined models can also be created.

The first step in tuning the model (line 1 in the algortihm below) is to choose a set of parameters to evaluate. For example, if fitting a Partial Least Squares (PLS) model, the number of PLS components to evaluate must be specified.

1. Define sets of model parapmeters to evaluate
2. **for** each parameter set **do**
3. **for** each resampling iteration **do**
   1. Hold-out specific samples
   2. [Optional] pre-process the data
   3. Fit the model on the remainder
   4. Predict the hold-out samples
4. *end*
5. Calculate the average performance across hold-out predictors
6. *end*
7. Determine the optimal parameter set
8. Fit the final model to all the training data using the optimal parameter set

Once the model and tuning parameter values have been defined, the type of resampling should be also specified, Currently, *k-fold* cross-validation (once or repeated), leave-one-out-cross validation and bootstrap (simple estimation or the 632 rule) resampling methods can be used by train. After resampling, the process produces a profile of performance is available to guide the user as to which tuning parameter values should be chosen. By default, the function automatically chooses the tuning parameters associated with the best value, although different algorithms can be used (see details below)

## An example

The Sonar data are available in the ,mlbench package. Here, we load the data:

library(mlbench)  
data(Sonar)  
str(Sonar[,1:10])

## 'data.frame': 208 obs. of 10 variables:  
## $ V1 : num 0.02 0.0453 0.0262 0.01 0.0762 0.0286 0.0317 0.0519 0.0223 0.0164 ...  
## $ V2 : num 0.0371 0.0523 0.0582 0.0171 0.0666 0.0453 0.0956 0.0548 0.0375 0.0173 ...  
## $ V3 : num 0.0428 0.0843 0.1099 0.0623 0.0481 ...  
## $ V4 : num 0.0207 0.0689 0.1083 0.0205 0.0394 ...  
## $ V5 : num 0.0954 0.1183 0.0974 0.0205 0.059 ...  
## $ V6 : num 0.0986 0.2583 0.228 0.0368 0.0649 ...  
## $ V7 : num 0.154 0.216 0.243 0.11 0.121 ...  
## $ V8 : num 0.16 0.348 0.377 0.128 0.247 ...  
## $ V9 : num 0.3109 0.3337 0.5598 0.0598 0.3564 ...  
## $ V10: num 0.211 0.287 0.619 0.126 0.446 ...

The function createDataPartition can be used to create a stratified randome sample of the data into training and test sets:

library(caret)  
set.seed(998)  
  
inTraining <- createDataPartition(Sonar$Class,p=.75,list=FALSE)  
training <- Sonar[inTraining,]  
testing <- Sonar[-inTraining,]

We will use these data illustrate functionality on this (and other) pages.

## Basic parameter tuning

By default, simple bootstrap resampling is used for line 3 in the algorithm above. Others are available, such as repeated K-fold cross-validation, leave-one-out etc. The function trainControl can be used to specifiy the type of resampling:

fitControl <- trainControl(## 10-fold CV  
 method = "repeatedcv",  
 number = 10,  
 ## repeated ten times  
 repeats = 10)

More information about trainControl is given in a [section below](http://topepo.github.io/caret/model-training-and-tuning.html#custom).

The first two arguments to train are the predictor and outcome data objects, respectively. The third pargument method specifies the type of model (see [train model list](http://topepo.github.io/caret/available-models.html') or [train Models By Tag](http://topepo.github.io/caret/train-models-by-tag.html)) . To illustrate we will fit a boosted tree moel via the gbm package. The basic syntax for fitting this model using repeated cross-validation is shown below:

library(gbm)

## Warning: package 'gbm' was built under R version 3.5.1

## Loading required package: survival

## Warning: package 'survival' was built under R version 3.5.1

##   
## 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

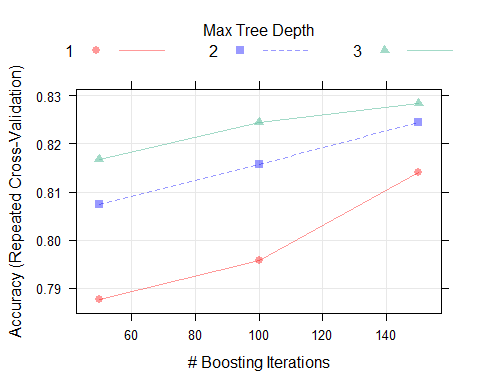
head(training)

## V1 V2 V3 V4 V5 V6 V7 V8 V9 V10  
## 1 0.0200 0.0371 0.0428 0.0207 0.0954 0.0986 0.1539 0.1601 0.3109 0.2111  
## 2 0.0453 0.0523 0.0843 0.0689 0.1183 0.2583 0.2156 0.3481 0.3337 0.2872  
## 3 0.0262 0.0582 0.1099 0.1083 0.0974 0.2280 0.2431 0.3771 0.5598 0.6194  
## 4 0.0100 0.0171 0.0623 0.0205 0.0205 0.0368 0.1098 0.1276 0.0598 0.1264  
## 6 0.0286 0.0453 0.0277 0.0174 0.0384 0.0990 0.1201 0.1833 0.2105 0.3039  
## 7 0.0317 0.0956 0.1321 0.1408 0.1674 0.1710 0.0731 0.1401 0.2083 0.3513  
## V11 V12 V13 V14 V15 V16 V17 V18 V19 V20  
## 1 0.1609 0.1582 0.2238 0.0645 0.0660 0.2273 0.3100 0.2999 0.5078 0.4797  
## 2 0.4918 0.6552 0.6919 0.7797 0.7464 0.9444 1.0000 0.8874 0.8024 0.7818  
## 3 0.6333 0.7060 0.5544 0.5320 0.6479 0.6931 0.6759 0.7551 0.8929 0.8619  
## 4 0.0881 0.1992 0.0184 0.2261 0.1729 0.2131 0.0693 0.2281 0.4060 0.3973  
## 6 0.2988 0.4250 0.6343 0.8198 1.0000 0.9988 0.9508 0.9025 0.7234 0.5122  
## 7 0.1786 0.0658 0.0513 0.3752 0.5419 0.5440 0.5150 0.4262 0.2024 0.4233  
## V21 V22 V23 V24 V25 V26 V27 V28 V29 V30  
## 1 0.5783 0.5071 0.4328 0.5550 0.6711 0.6415 0.7104 0.8080 0.6791 0.3857  
## 2 0.5212 0.4052 0.3957 0.3914 0.3250 0.3200 0.3271 0.2767 0.4423 0.2028  
## 3 0.7974 0.6737 0.4293 0.3648 0.5331 0.2413 0.5070 0.8533 0.6036 0.8514  
## 4 0.2741 0.3690 0.5556 0.4846 0.3140 0.5334 0.5256 0.2520 0.2090 0.3559  
## 6 0.2074 0.3985 0.5890 0.2872 0.2043 0.5782 0.5389 0.3750 0.3411 0.5067  
## 7 0.7723 0.9735 0.9390 0.5559 0.5268 0.6826 0.5713 0.5429 0.2177 0.2149  
## V31 V32 V33 V34 V35 V36 V37 V38 V39 V40  
## 1 0.1307 0.2604 0.5121 0.7547 0.8537 0.8507 0.6692 0.6097 0.4943 0.2744  
## 2 0.3788 0.2947 0.1984 0.2341 0.1306 0.4182 0.3835 0.1057 0.1840 0.1970  
## 3 0.8512 0.5045 0.1862 0.2709 0.4232 0.3043 0.6116 0.6756 0.5375 0.4719  
## 4 0.6260 0.7340 0.6120 0.3497 0.3953 0.3012 0.5408 0.8814 0.9857 0.9167  
## 6 0.5580 0.4778 0.3299 0.2198 0.1407 0.2856 0.3807 0.4158 0.4054 0.3296  
## 7 0.5811 0.6323 0.2965 0.1873 0.2969 0.5163 0.6153 0.4283 0.5479 0.6133  
## V41 V42 V43 V44 V45 V46 V47 V48 V49 V50  
## 1 0.0510 0.2834 0.2825 0.4256 0.2641 0.1386 0.1051 0.1343 0.0383 0.0324  
## 2 0.1674 0.0583 0.1401 0.1628 0.0621 0.0203 0.0530 0.0742 0.0409 0.0061  
## 3 0.4647 0.2587 0.2129 0.2222 0.2111 0.0176 0.1348 0.0744 0.0130 0.0106  
## 4 0.6121 0.5006 0.3210 0.3202 0.4295 0.3654 0.2655 0.1576 0.0681 0.0294  
## 6 0.2707 0.2650 0.0723 0.1238 0.1192 0.1089 0.0623 0.0494 0.0264 0.0081  
## 7 0.5017 0.2377 0.1957 0.1749 0.1304 0.0597 0.1124 0.1047 0.0507 0.0159  
## V51 V52 V53 V54 V55 V56 V57 V58 V59 V60  
## 1 0.0232 0.0027 0.0065 0.0159 0.0072 0.0167 0.0180 0.0084 0.0090 0.0032  
## 2 0.0125 0.0084 0.0089 0.0048 0.0094 0.0191 0.0140 0.0049 0.0052 0.0044  
## 3 0.0033 0.0232 0.0166 0.0095 0.0180 0.0244 0.0316 0.0164 0.0095 0.0078  
## 4 0.0241 0.0121 0.0036 0.0150 0.0085 0.0073 0.0050 0.0044 0.0040 0.0117  
## 6 0.0104 0.0045 0.0014 0.0038 0.0013 0.0089 0.0057 0.0027 0.0051 0.0062  
## 7 0.0195 0.0201 0.0248 0.0131 0.0070 0.0138 0.0092 0.0143 0.0036 0.0103  
## Class  
## 1 R  
## 2 R  
## 3 R  
## 4 R  
## 6 R  
## 7 R

set.seed(825)  
gbmFit1 <- train(Class~.,data=training,  
 method="gbm",  
 trControl=fitControl,  
 ## This last option is actualyy one  
 ## for gbm() that passes through  
 verbose=FALSE)  
  
gbmFit1

## Stochastic Gradient Boosting   
##   
## 157 samples  
## 60 predictor  
## 2 classes: 'M', 'R'   
##   
## No pre-processing  
## Resampling: Cross-Validated (10 fold, repeated 10 times)   
## Summary of sample sizes: 141, 141, 142, 141, 141, 142, ...   
## Resampling results across tuning parameters:  
##   
## interaction.depth n.trees Accuracy Kappa   
## 1 50 0.7876495 0.5712306  
## 1 100 0.7957794 0.5867732  
## 1 150 0.8140441 0.6235694  
## 2 50 0.8073676 0.6101031  
## 2 100 0.8158211 0.6263183  
## 2 150 0.8244608 0.6441406  
## 3 50 0.8167843 0.6294107  
## 3 100 0.8244461 0.6444641  
## 3 150 0.8284510 0.6522968  
##   
## Tuning parameter 'shrinkage' was held constant at a value of 0.1  
##   
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10  
## Accuracy was used to select the optimal model using the largest value.  
## The final values used for the model were n.trees = 150,  
## interaction.depth = 3, shrinkage = 0.1 and n.minobsinnode = 10.

gbmFit1 %>% plot()



For a gradient boosting machine (GBM) model, there are three main tuning parameters: - number of iterations, i.e.,trees (called n.trees in the gbm function) - complexity of the tree called interaction.depth - learning rate: how quickly the algorithm adapts called shrinkage - the minimum number of training set samples in a node to commence splitting (n.minobsinnode)

The default values tested for this model are shown in the first two columns (shrinkage and n.minobsinnode are not shown beause the grid set of candidate models all use a single value for these tuning parameters). The column labeled “Accuracy” is the overall agreement rate averaged over cross-validation iterations. The agreement standard deviation is also calculated from the cross-validation results. The column “Kappa” is Cohen’s (unweighted) Kappa statistic averaged across the resampling results. train works with specific models (see train Model List or train Models By Tag). For these models, train can automatically create a grid of tuning parameters. By default, if is the number of tuning parameters, the grid size is . As another example, regularized discriminant analysis (RDA) models have two parameters (gamma and lambda), both of which lie between zero and one. The default training grid would produce nine combinations in this two-dimensional space.

There is additional functionality in train that is described in the next section.

## Notes on reproducibility

Many models utilize random numbers during the phase where parameters are estimated. Also, the resampling indices are chosen using random numbers. There are two main ways to control the randomness in order to assure reproducible results.

* There are two approaches to ensuring that the same *resamples* are used between calls to train. The first is to use set.seed just prior to calling train. The first use of random numbers is to create the resampling information. Alternatively, if you would like to use specific splits of the data, the index argument of the trainControl function can be used. This is briefly discussed below.
* When the models are created *inside of resampling*, the seeds can also be set. While setting the seed prior to calling train may guarantee that the same random numbers are used, this is unlikely to be the case when [parallel processing](http://topepo.github.io/caret/parallel-processing.html) is used (depending which technology is utilized). To set the model fitting seeds, trainControl has an additional argument called seeds that can be used. The value for this argument is a list of integer vectors that are used as seeds. The help page for trainControl describes the appropriate format for this option.

## 5.5 Customizing the Tuning Process

There are a few ways to customize the process of selecting tuning/complexity parameters and building the final model.

### 5.5.1 Pre-processing options

As previously mentioned,train can pre-process the data in various ways prior to model fitting. The function preProcess is automatically used. This function can be used for centering and scaling, imputation (see details below), applying the spatial sign transformation and feature extraction via principal component analysis or independent component analysis.

To specify what pre-processing should occur, the train function has an argument called preProcess. This argument takes a character string of methods that would normally be passed to the method argument of the prePRocess function. Additional options for the preProcess function can be passed via the trainControl function.

These processing steps would be applied during any predictions generated using predict.train, extractPrediction or extractProbs (see details later in this document). The pre-processing would not be applied to predictions that directly use the boject$finalModel object.

For imputation, there are three methods currently implemented:

* **k-nearest neighbors** takes a sample with missing values and finds the k closest samples in the training set. The average of the k training set values for that predictor are used as a substitute for the original data. When calculating the distances to the training set samples, the predictors used in the calculation are the ones with no missing values for that sample and no missing values in the training set.
* another approach is to fit a **bagged tree model** for each predictor using the training set samples. This is usually a fairly accurate model and can handle missing values. When a predictor for a sample requires imputation, the values for the other predictors are fed through the bagged tree and the prediction is used as the new value. This model can have significant computational cost.
* the **median** of the predictor’s training set values can be used to estimate the missing data.

If there are missing values in the training set, PCA and ICA models only use complete samples.

### 5.5.2 Alternate Tuning Grids

The tuning parameter grid can be specified by the user. The argument tuneGrid can take a data frame with columns for each tuning parameter.The column names should be the same as the fitting function’s arguments. For the previously mentioned RDA example, the names would be gamma and lambda. train will tune the model over each combination of values in the rows.

For the boosted tree model, we can fix the learning rate and evaluate more than three values of n.trees:

gbmGrid <- expand.grid(interaction.depth=c(1,5,9),  
 n.trees=(1:30)\*50,  
 shrinkage=0.1,  
 n.minobsinnode=20)  
  
nrow(gbmGrid)

## [1] 90

set.seed(825)  
  
gbmFit2 <- train(Class~., data=training,  
 method="gbm",  
 trControl=fitControl,  
 verbose=FALSE,  
 ## Now specify the exact models  
 ## to evaluate:  
 tuneGrid=gbmGrid  
 )  
gbmFit2

## Stochastic Gradient Boosting   
##   
## 157 samples  
## 60 predictor  
## 2 classes: 'M', 'R'   
##   
## No pre-processing  
## Resampling: Cross-Validated (10 fold, repeated 10 times)   
## Summary of sample sizes: 141, 141, 142, 141, 141, 142, ...   
## Resampling results across tuning parameters:  
##   
## interaction.depth n.trees Accuracy Kappa   
## 1 50 0.7715196 0.5383303  
## 1 100 0.7962255 0.5884175  
## 1 150 0.8134387 0.6230354  
## 1 200 0.8204485 0.6379029  
## 1 250 0.8212402 0.6393044  
## 1 300 0.8211936 0.6388168  
## 1 350 0.8208088 0.6380987  
## 1 400 0.8227672 0.6419025  
## 1 450 0.8273824 0.6512246  
## 1 500 0.8233922 0.6430845  
## 1 550 0.8245588 0.6451007  
## 1 600 0.8213505 0.6390116  
## 1 650 0.8220588 0.6402634  
## 1 700 0.8213456 0.6385238  
## 1 750 0.8195074 0.6349871  
## 1 800 0.8201373 0.6362480  
## 1 850 0.8202623 0.6363538  
## 1 900 0.8202206 0.6363208  
## 1 950 0.8195172 0.6349773  
## 1 1000 0.8208088 0.6374650  
## 1 1050 0.8227672 0.6414218  
## 1 1100 0.8213922 0.6390257  
## 1 1150 0.8194755 0.6350967  
## 1 1200 0.8233505 0.6428458  
## 1 1250 0.8252721 0.6468659  
## 1 1300 0.8213971 0.6392640  
## 1 1350 0.8257721 0.6477302  
## 1 1400 0.8227255 0.6419176  
## 1 1450 0.8233505 0.6432857  
## 1 1500 0.8278088 0.6519294  
## 5 50 0.7898015 0.5758733  
## 5 100 0.8244804 0.6454184  
## 5 150 0.8355956 0.6673310  
## 5 200 0.8353186 0.6669529  
## 5 250 0.8306520 0.6572224  
## 5 300 0.8350221 0.6655867  
## 5 350 0.8346005 0.6651483  
## 5 400 0.8339338 0.6635681  
## 5 450 0.8365956 0.6692071  
## 5 500 0.8353088 0.6664772  
## 5 550 0.8352304 0.6663351  
## 5 600 0.8367574 0.6694915  
## 5 650 0.8361740 0.6681545  
## 5 700 0.8367206 0.6692299  
## 5 750 0.8360907 0.6680854  
## 5 800 0.8342990 0.6646367  
## 5 850 0.8362574 0.6687173  
## 5 900 0.8374706 0.6710983  
## 5 950 0.8387990 0.6735621  
## 5 1000 0.8400074 0.6759778  
## 5 1050 0.8368775 0.6698196  
## 5 1100 0.8393407 0.6745327  
## 5 1150 0.8385539 0.6730231  
## 5 1200 0.8392990 0.6741921  
## 5 1250 0.8418088 0.6792461  
## 5 1300 0.8399338 0.6758180  
## 5 1350 0.8399706 0.6757946  
## 5 1400 0.8405588 0.6770214  
## 5 1450 0.8379755 0.6717665  
## 5 1500 0.8398873 0.6753613  
## 9 50 0.7971691 0.5904836  
## 9 100 0.8311422 0.6590396  
## 9 150 0.8349706 0.6665370  
## 9 200 0.8416887 0.6798158  
## 9 250 0.8430637 0.6829102  
## 9 300 0.8398922 0.6765830  
## 9 350 0.8461520 0.6891024  
## 9 400 0.8442353 0.6849527  
## 9 450 0.8451005 0.6866426  
## 9 500 0.8386005 0.6733838  
## 9 550 0.8354338 0.6674123  
## 9 600 0.8373137 0.6709873  
## 9 650 0.8392255 0.6746129  
## 9 700 0.8412206 0.6787989  
## 9 750 0.8400074 0.6762078  
## 9 800 0.8411838 0.6786330  
## 9 850 0.8373505 0.6708973  
## 9 900 0.8380588 0.6720172  
## 9 950 0.8381324 0.6721905  
## 9 1000 0.8374706 0.6708754  
## 9 1050 0.8418873 0.6797354  
## 9 1100 0.8419289 0.6798824  
## 9 1150 0.8424755 0.6808119  
## 9 1200 0.8412623 0.6783979  
## 9 1250 0.8438456 0.6836096  
## 9 1300 0.8432206 0.6825476  
## 9 1350 0.8451373 0.6863010  
## 9 1400 0.8432623 0.6824590  
## 9 1450 0.8413873 0.6785603  
## 9 1500 0.8407206 0.6771598  
##   
## Tuning parameter 'shrinkage' was held constant at a value of 0.1  
##   
## Tuning parameter 'n.minobsinnode' was held constant at a value of 20  
## Accuracy was used to select the optimal model using the largest value.  
## The final values used for the model were n.trees = 350,  
## interaction.depth = 9, shrinkage = 0.1 and n.minobsinnode = 20.

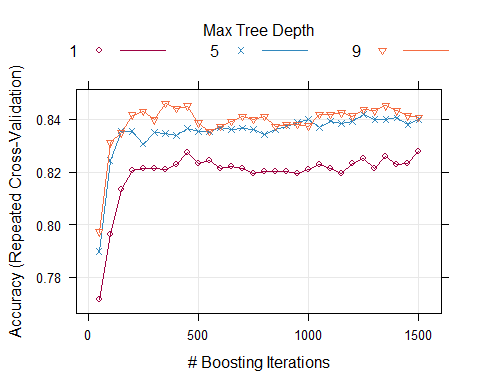
Another option is to use a random sample of possible tuning parameter combinations, i.e. “random search”(pdf). This functionality is described on [this page](http://topepo.github.io/caret/random-hyperparameter-search.html).

To use a random search, use the option search = "random" in the call to trainControl. In this situation, the tuneLength parameter defines the total number of parameter combinations that will be evaluated.

### Plotting the resampling profile

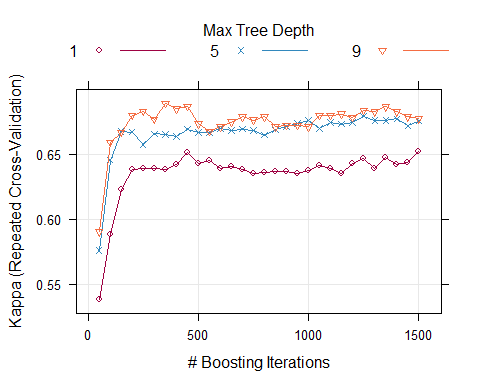
The plot function can be used to examine the relationship between the estimates of performance and the tuninng parameters. For example, a simple invokation of the function shows the results for the first performance measure: 2

trellis.par.set(caretTheme())  
plot(gbmFit2)



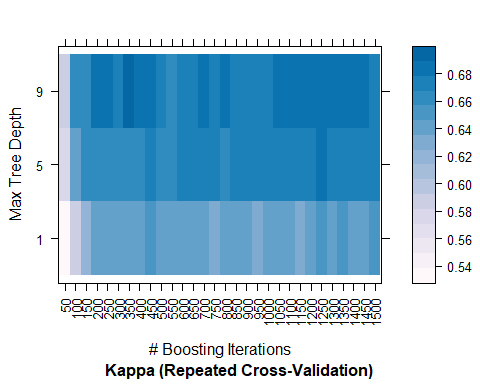
Other performance metrics can be shown using the metric option:

trellis.par.set(caretTheme())  
plot(gbmFit2, metric="Kappa")



Other types of plot are also available. See ?plot.train for more details. The code below shows a heatmap of the results:

trellis.par.set(caretTheme())  
plot(gbmFit2, metric="Kappa",plotType="level",  
 scales=list(x=list(rot=90)))



### The trainControl function

The function trainControl generates parameters that further control how models are created, with possible values.

* method:The resampling method: "boot", "cv", "LOOCV", "LGOCV", "repeatedcv", "timeslice", "none" and "oob". The last value, out-of-bag estimates, can only be used by random forest, bagged trees, bagged earth, bagged flexible discriminant analysis, or conditional tree forest models. GBM models are not included (the gbm package maintainer has indicated that it would not be a good idea to choose tuning parameter values based on the model OOB error estimates with boosted trees). Also, for leave-one-out cross-validation, no uncertainty estimates are given for the resampled performance measures.
* number and repeats: number controls with the number of folds in K-fold cross-validation or number of resampling iterations for bootstrapping and leave-group-out cross-validation. repeats applied only to repeated cross-validation. Suppose that method = "repeatedcv", number = 10 and repeats = 3,then three separate 10-fold cross-validations are used as the resampling scheme.
* verboseIter: A logical for printing a training log.
* returnData: A logical for saving the data into a slot called trainingData
* p: For leave-group out cross-validation: the training percentage
* For method="timeslice, trainControl has options initialWindow, horizon and fixedWindow that govern how cross-validation can be used for time series data.
* classProbs: a logical value determining whether class probabilities should be computed for held-out samples during resample.
* index and indexOut: optional lists with elements for each resampling iteration. Each list element is the sample rows used for training at that iteration or should be held-out. When these values are not specified, train will generate them.
* summaryFunction: a function to computed alternate performance summaries.
* selectionFunction: a function to choose the optimal tuning parameters. and examples.
* PCAthresh, ICAcomp and k: these are all options to pass to the preProcess function (when used).
* returnResamp: a character string containing one of the following values: “all”, “final” or “none”. This specifies how much of the resampled performance measures to save.
* allowParallel: a logical that governs whether train should use parallel processing (if availible).

### Alternatate performance metrics

The user can change the metric used to determine the best settings. By default, RMSE, , and the mean absolute error (MAE) are computed for regression while accuracy and Kappa are computed for classification. Also by default, the parameter values are chosen using RMSE and accuracy, respectively for regression and classification. The metric argument of the train function allows the user to control which the optimality criterion is used. For example, in problems where there are a low percentage of samples in one class, using metric = "Kappa" can improve quality of the final model.

If none of these parameters are satisfactory, the user can also compute custom performance metrics. The trainControl function has a argument called summaryFunction that specifies a function for computing performance. The function should have these arguments:

* data is reference for a data frame or matrix with columns called obs and pred for the observed and predicted outcome values (either numeric data for regression or character values for classification). Currently, class probabilities are not passed to the function. The values in data are the held-out predictions (and their associated reference values) for a single combination of tuning parameters. If the classProbs argument of the trainControl object is set to TRUE, additional columns in data will be present that contains the class probabilities. The names of these columns are the same as the class levels. Also, if weights were specified in the call to train, a column called weights will also be in the data set. Additionally, if the recipe method for train was used (see this section of documentation), other variables not used in the model will also be included. This can be accomplished by adding a role in the recipe of "performance var". An example is given in the recipe section of this site.)
* lev is a character string that has the outcome factor levels taken from the training data. For regression, a value of NULL is passed into the function.
* model is a character string for the model being used (i.e. the value passed to the method argument of train).

The output to the function should be a vector of numeric summary metrics with non-null names. By default, train evaluate classification models in terms of the predicted classes. Optionally, class probabilities can also be used to measure performance. To obtain predicted class probabilities within the resampling process, the argument classProbs in trainControl must be set to TRUE. This merges columns of probabilities into the predictions generated from each resample (there is a column per class and the column names are the class names).

As shown in the last section, custom fuctions can be used to calculate performance scores that are averaged over the **resamples**. Another built-in function, twoClasssummary, will compute the **sensitivity**, **specificity** and **are under the ROC curve**.

head(twoClassSummary)

##   
## 1 function (data, lev = NULL, model = NULL)   
## 2 {   
## 3 lvls <- levels(data$obs)   
## 4 if (length(lvls) > 2)   
## 5 stop(paste("Your outcome has", length(lvls), "levels. The twoClassSummary() function isn't appropriate."))  
## 6 requireNamespaceQuietStop("ModelMetrics")

To rebuild the boosted tree model using this criterion, we can see the relationship between the tuning parameters and the are under the ROC curve using the following code:

fitControl <- trainControl(method = "repeatedcv",  
 number = 10,  
 repeats = 10,  
 ## Estimate class probabilities  
 classProbs = TRUE,  
 ## Evaluate performance using   
 ## the following function  
 summaryFunction = twoClassSummary)  
  
set.seed(825)  
gbmFit3 <- train(Class~., data = training,  
 method="gbm",  
 trControl=fitControl,  
 verbose=FALSE,  
 tuneGrid=gbmGrid,  
 ## specify which metric to optimize  
 metric="ROC"  
 )  
gbmFit3

## Stochastic Gradient Boosting   
##   
## 157 samples  
## 60 predictor  
## 2 classes: 'M', 'R'   
##   
## No pre-processing  
## Resampling: Cross-Validated (10 fold, repeated 10 times)   
## Summary of sample sizes: 141, 141, 142, 141, 141, 142, ...   
## Resampling results across tuning parameters:  
##   
## interaction.depth n.trees ROC Sens Spec   
## 1 50 0.8720635 0.8279167 0.7089286  
## 1 100 0.8833358 0.8454167 0.7417857  
## 1 150 0.8917808 0.8520833 0.7694643  
## 1 200 0.8906746 0.8550000 0.7823214  
## 1 250 0.8935144 0.8573611 0.7814286  
## 1 300 0.8944469 0.8572222 0.7810714  
## 1 350 0.8915972 0.8598611 0.7776786  
## 1 400 0.8896999 0.8647222 0.7760714  
## 1 450 0.8927654 0.8629167 0.7873214  
## 1 500 0.8923214 0.8605556 0.7812500  
## 1 550 0.8916270 0.8626389 0.7810714  
## 1 600 0.8915253 0.8580556 0.7796429  
## 1 650 0.8915303 0.8594444 0.7796429  
## 1 700 0.8916319 0.8591667 0.7780357  
## 1 750 0.8903522 0.8606944 0.7728571  
## 1 800 0.8922148 0.8583333 0.7767857  
## 1 850 0.8920759 0.8597222 0.7753571  
## 1 900 0.8934251 0.8595833 0.7755357  
## 1 950 0.8903249 0.8597222 0.7741071  
## 1 1000 0.8920511 0.8604167 0.7757143  
## 1 1050 0.8911434 0.8629167 0.7769643  
## 1 1100 0.8909623 0.8579167 0.7798214  
## 1 1150 0.8903943 0.8568056 0.7771429  
## 1 1200 0.8919345 0.8604167 0.7810714  
## 1 1250 0.8912971 0.8583333 0.7875000  
## 1 1300 0.8909449 0.8595833 0.7783929  
## 1 1350 0.8914385 0.8627778 0.7835714  
## 1 1400 0.8924678 0.8583333 0.7823214  
## 1 1450 0.8914137 0.8569444 0.7851786  
## 1 1500 0.8916791 0.8640278 0.7866071  
## 5 50 0.8840079 0.8297222 0.7457143  
## 5 100 0.9012550 0.8677778 0.7758929  
## 5 150 0.9056101 0.8784722 0.7866071  
## 5 200 0.9032465 0.8818056 0.7828571  
## 5 250 0.9037971 0.8793056 0.7757143  
## 5 300 0.9053373 0.8865278 0.7767857  
## 5 350 0.9051587 0.8843056 0.7787500  
## 5 400 0.9058755 0.8840278 0.7773214  
## 5 450 0.9056597 0.8830556 0.7842857  
## 5 500 0.9065997 0.8816667 0.7828571  
## 5 550 0.9076215 0.8805556 0.7837500  
## 5 600 0.9074628 0.8818056 0.7857143  
## 5 650 0.9053447 0.8831944 0.7830357  
## 5 700 0.9077207 0.8827778 0.7844643  
## 5 750 0.9072321 0.8819444 0.7844643  
## 5 800 0.9062624 0.8797222 0.7833929  
## 5 850 0.9063219 0.8809722 0.7860714  
## 5 900 0.9055580 0.8820833 0.7873214  
## 5 950 0.9057788 0.8866667 0.7848214  
## 5 1000 0.9059201 0.8893056 0.7844643  
## 5 1050 0.9064732 0.8822222 0.7857143  
## 5 1100 0.9078869 0.8843056 0.7883929  
## 5 1150 0.9069519 0.8843056 0.7866071  
## 5 1200 0.9071776 0.8879167 0.7841071  
## 5 1250 0.9076711 0.8902778 0.7866071  
## 5 1300 0.9072917 0.8845833 0.7894643  
## 5 1350 0.9073189 0.8868056 0.7867857  
## 5 1400 0.9071478 0.8880556 0.7867857  
## 5 1450 0.9081969 0.8843056 0.7853571  
## 5 1500 0.9087103 0.8888889 0.7841071  
## 9 50 0.8900967 0.8418056 0.7476786  
## 9 100 0.9070486 0.8686111 0.7889286  
## 9 150 0.9129142 0.8751389 0.7896429  
## 9 200 0.9136781 0.8854167 0.7925000  
## 9 250 0.9141096 0.8830556 0.7978571  
## 9 300 0.9115427 0.8781944 0.7967857  
## 9 350 0.9120437 0.8876389 0.7996429  
## 9 400 0.9096478 0.8862500 0.7966071  
## 9 450 0.9122297 0.8887500 0.7957143  
## 9 500 0.9118849 0.8829167 0.7885714  
## 9 550 0.9117907 0.8770833 0.7887500  
## 9 600 0.9114459 0.8829167 0.7858929  
## 9 650 0.9109102 0.8865278 0.7857143  
## 9 700 0.9112500 0.8866667 0.7900000  
## 9 750 0.9113120 0.8855556 0.7885714  
## 9 800 0.9109301 0.8865278 0.7900000  
## 9 850 0.9105382 0.8843056 0.7842857  
## 9 900 0.9100347 0.8854167 0.7842857  
## 9 950 0.9103522 0.8843056 0.7857143  
## 9 1000 0.9101860 0.8841667 0.7846429  
## 9 1050 0.9115352 0.8898611 0.7875000  
## 9 1100 0.9096453 0.8877778 0.7898214  
## 9 1150 0.9087277 0.8912500 0.7873214  
## 9 1200 0.9080010 0.8890278 0.7871429  
## 9 1250 0.9074256 0.8912500 0.7900000  
## 9 1300 0.9082490 0.8888889 0.7914286  
## 9 1350 0.9077183 0.8936111 0.7901786  
## 9 1400 0.9067684 0.8898611 0.7901786  
## 9 1450 0.9089112 0.8876389 0.7887500  
## 9 1500 0.9082961 0.8887500 0.7860714  
##   
## Tuning parameter 'shrinkage' was held constant at a value of 0.1  
##   
## Tuning parameter 'n.minobsinnode' was held constant at a value of 20  
## ROC was used to select the optimal model using the largest value.  
## The final values used for the model were n.trees = 250,  
## interaction.depth = 9, shrinkage = 0.1 and n.minobsinnode = 20.

In this case, the average area under the ROC curve associated with the optimal tuning parameters was 0.914 across the 100 resamples.

## Choosing the Final Model

Another method for customizing the tuning process is to modify the algorithm that is used to select the best parameter values, given the performance numbers. By default, the train function chooses the model with the largest performance value (or smallest, for mean squared error in regression models).

Other schemes for selecting model can be used. Breiman et al (1984) suggested the “one standard error rule” for simple tree-based models. In this case, the model with the best performance value is identified and, using resampling, we can estimate the standard error of performance. The final model used was the simplest model within one standard error of the (empirically) best model. With simple trees this makes sense, since these models will start to over-fit as they become more and more specific to the training data.

train allows the user to specify alternate rules for selecting the final model. The argument selectionFunction can be used to supply a function to algorithmically determine the final model. There are three existing functions in the package: best is chooses the largest/smallest value, oneSE attempts to capture the spirit of Breiman et al (1984) and tolerance selects the least complex model within some percent tolerance of the best value. See ?best for more details.

User-defined functions can be used, as long as they have the floowing arguments: - x is a data frame containing the tune parameters and their associated performance metrics. Each row corresponds to a different tuning parameter combination. - metrica character string indicating which performance metric should be optimized (this is passed in directly from the metric argument of train - maximizeis a single logical value indicating whether larger values of the performance metric are better (this is also directly passed from the call to train).

The function should output a single integer indicating which row in x is chosen.

As an example, if we chose the previous boosted tree model on the basis of overall accuracy, we would choose: n.trees = 250, interaction.depth = 9, shrinkage = 0.1, n.minobsinnode = 20. However, the scale in this plots is fairly tight, with accuracy values ranging from 0.872 to 0.914. A less complex model (e.g. fewer, more shallow trees) might also yield acceptable accuracy.

whichTwoPct <- tolerance(gbmFit3$results,metric="ROC",  
 tol=2,maximize=T)  
cat("best model within 2pct of best:\n")

## best model within 2pct of best:

gbmFit3$results[whichTwoPct,1:6]

## shrinkage interaction.depth n.minobsinnode n.trees ROC Sens  
## 32 0.1 5 20 100 0.901255 0.8677778

This indicates that we can get a less complex model with an area under the ROC curve of 0.901 (compared to the “pick the best” value of 0.914).

The main issue with these functions is related to ordering the models from simplest to complex. In some cases, this is easy (e.g. simple trees, partial least squares), but in cases such as this model, the ordering of models is subjective. For example, is a boosted tree model using 100 iterations and a tree depth of 2 more complex than one with 50 iterations and a depth of 8? The package makes some choices regarding the orderings. In the case of boosted trees, the package assumes that increasing the number of iterations adds complexity at a faster rate than increasing the tree depth, so models are ordered on the number of iterations then ordered with depth. See ?best for more examples for specific models.

## Extracting predictions and class probabilities

As previously mentioned, objects produced by the train function contain the “optimized” model in the finalModel sub-object. Predictions can be made from these objects as usual.

In some cases, such as pls or gbm objects, additional parameters from the optimized fit may beed to be specified. In these cases, the train objects uses the results of the parameter optimization to predict new samples. For example, if predictons were created using predict.gbm, the user would have to specify the number of trees directly (there is no default). Aoso, for binary classification, the predictions from this function take the form of the probability of one of the classes, so extra steps are required to convert this to a factor vector. predict.train automatically handles these details for this.

Also, there are very few standard syntaxes for model predictions in R. For example, to get class probabilities, many predict methods have an argument called type that is used to specify whether the classes or probabilities should be generated. Different packages use values of type, such as prob, posterior, response, probability or raw. In other cases, completely different syntax is used.

For predict.train, the type options are standardized to be “class” and “probs” (the underlying code matches these to the appropriate choices for each model). For example:

predict(gbmFit3,newdata = head(testing))

## [1] R R R R M M  
## Levels: M R

predict(gbmFit3,newdata = head(testing),type="prob")

## M R  
## 1 0.2600273005 0.73997270  
## 2 0.0113631254 0.98863687  
## 3 0.0003797417 0.99962026  
## 4 0.1532818480 0.84671815  
## 5 0.9879809255 0.01201907  
## 6 0.6342884390 0.36571156

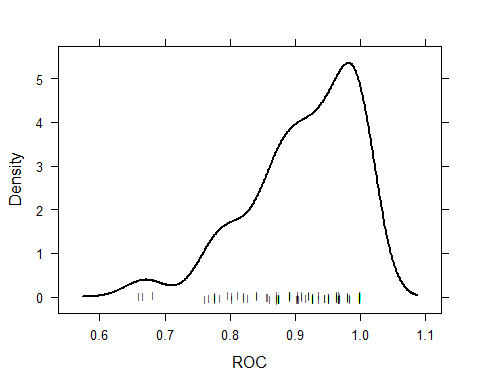
## Exploring and comparing resampling distributions

### Within-model

There are several lattice functions that can be used to explore relationships between tuning parameters and the resampling results for a specific model: - xplot and stripplot can be used to plot resampling statistics against (numeric) tuning parameters. - histogram and densityplot can be used to look at distributions of the tuning parameters across tuning parameters.

For example, the following statement create a density plot:

trellis.par.set(caretTheme())  
densityplot(gbmFit3,pch="|")



Note that if you are interested in plotting the resampling results across multiple tuning parameters, the option resamples = "all" should be used in the control object.

### Between-models

The caret package also includes functions to characterize the differences between models(generated using train,sbf or rfe) via their resampling distributions. These functions are based on the work of [Hothorn et al. (2005)](https://homepage.boku.ac.at/leisch/papers/Hothorn+Leisch+Zeileis-2005.pdf) and [and Eugster et al (2008).](https://epub.ub.uni-muenchen.de/10604/1/tr56.pdf).

First, a support vector machine model is to fit to the Sonar data. The data are centered and scaled using the preProc argument. Note that the same random number seed is set prior to the model that is identical to the seed used for the **boosted tree model**. This ensures that the same resampling sets are used, which will come in handy when we compare the resampling profiles between models.

set.seed(825)  
  
# SVM with radial basis function kernel  
svmFit <- train(Class~., data=training,  
 method="svmRadial",  
 trControl=fitControl,  
 preProc=c("center","scale"),  
 tuneLength=8,  
 metric="ROC")  
svmFit

## Support Vector Machines with Radial Basis Function Kernel   
##   
## 157 samples  
## 60 predictor  
## 2 classes: 'M', 'R'   
##   
## Pre-processing: centered (60), scaled (60)   
## Resampling: Cross-Validated (10 fold, repeated 10 times)   
## Summary of sample sizes: 141, 141, 142, 141, 141, 142, ...   
## Resampling results across tuning parameters:  
##   
## C ROC Sens Spec   
## 0.25 0.8700471 0.7572222 0.7539286  
## 0.50 0.9102133 0.8587500 0.7819643  
## 1.00 0.9344767 0.8809722 0.7807143  
## 2.00 0.9346230 0.8870833 0.7878571  
## 4.00 0.9455828 0.8979167 0.8192857  
## 8.00 0.9486012 0.9002778 0.8162500  
## 16.00 0.9491146 0.9050000 0.8216071  
## 32.00 0.9491146 0.9119444 0.8164286  
##   
## Tuning parameter 'sigma' was held constant at a value of 0.01334808  
## ROC was used to select the optimal model using the largest value.  
## The final values used for the model were sigma = 0.01334808 and C = 16.

Also, a regularized discriminant analysis model was fit.

# Regularized discriminant analysis  
set.seed(825)  
rdaFit <- train(Class ~ ., data = training,   
 method = "rda",   
 trControl = fitControl,   
 tuneLength = 4,  
 metric = "ROC")  
rdaFit

## Regularized Discriminant Analysis   
##   
## 157 samples  
## 60 predictor  
## 2 classes: 'M', 'R'   
##   
## No pre-processing  
## Resampling: Cross-Validated (10 fold, repeated 10 times)   
## Summary of sample sizes: 141, 141, 142, 141, 141, 142, ...   
## Resampling results across tuning parameters:  
##   
## gamma lambda ROC Sens Spec   
## 0.0000000 0.0000000 0.6653311 0.9052778 0.3998214  
## 0.0000000 0.3333333 0.8516543 0.8425000 0.7564286  
## 0.0000000 0.6666667 0.8697941 0.8212500 0.7969643  
## 0.0000000 1.0000000 0.8560193 0.7772222 0.7898214  
## 0.3333333 0.0000000 0.9069866 0.8884722 0.7630357  
## 0.3333333 0.3333333 0.9247817 0.9201389 0.7942857  
## 0.3333333 0.6666667 0.9247148 0.9251389 0.7830357  
## 0.3333333 1.0000000 0.8782242 0.8462500 0.7817857  
## 0.6666667 0.0000000 0.9015352 0.8804167 0.7557143  
## 0.6666667 0.3333333 0.9100471 0.9044444 0.7607143  
## 0.6666667 0.6666667 0.9033209 0.9013889 0.7687500  
## 0.6666667 1.0000000 0.8704638 0.8077778 0.7883929  
## 1.0000000 0.0000000 0.7283631 0.6652778 0.6507143  
## 1.0000000 0.3333333 0.7308557 0.6676389 0.6564286  
## 1.0000000 0.6666667 0.7322222 0.6770833 0.6633929  
## 1.0000000 1.0000000 0.7342832 0.6745833 0.6607143  
##   
## ROC was used to select the optimal model using the largest value.  
## The final values used for the model were gamma = 0.3333333 and lambda  
## = 0.3333333.

Given these models, can we make statistical statements about their performance differences? To do this, we first collect the resampling results using resamples.

resamps <- resamples(list(GBM=gbmFit3,  
 SVM=svmFit,  
 RDA=rdaFit))  
resamps

##   
## Call:  
## resamples.default(x = list(GBM = gbmFit3, SVM = svmFit, RDA = rdaFit))  
##   
## Models: GBM, SVM, RDA   
## Number of resamples: 100   
## Performance metrics: ROC, Sens, Spec   
## Time estimates for: everything, final model fit

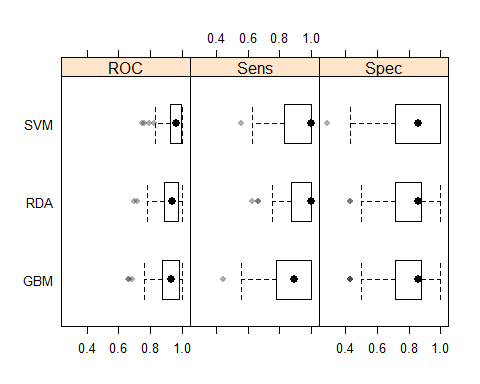
summary(resamps)

##   
## Call:  
## summary.resamples(object = resamps)  
##   
## Models: GBM, SVM, RDA   
## Number of resamples: 100   
##   
## ROC   
## Min. 1st Qu. Median Mean 3rd Qu. Max. NA's  
## GBM 0.6607143 0.8745040 0.9285714 0.9141096 0.9841890 1 0  
## SVM 0.7500000 0.9265873 0.9642857 0.9491146 0.9895833 1 0  
## RDA 0.6964286 0.8901910 0.9365079 0.9247817 0.9747024 1 0  
##   
## Sens   
## Min. 1st Qu. Median Mean 3rd Qu. Max. NA's  
## GBM 0.4444444 0.7777778 0.8888889 0.8830556 1 1 0  
## SVM 0.5555556 0.8506944 1.0000000 0.9050000 1 1 0  
## RDA 0.6250000 0.8750000 1.0000000 0.9201389 1 1 0  
##   
## Spec   
## Min. 1st Qu. Median Mean 3rd Qu. Max. NA's  
## GBM 0.4285714 0.7142857 0.8571429 0.7978571 0.875 1 0  
## SVM 0.2857143 0.7142857 0.8571429 0.8216071 1.000 1 0  
## RDA 0.4285714 0.7142857 0.8571429 0.7942857 0.875 1 0

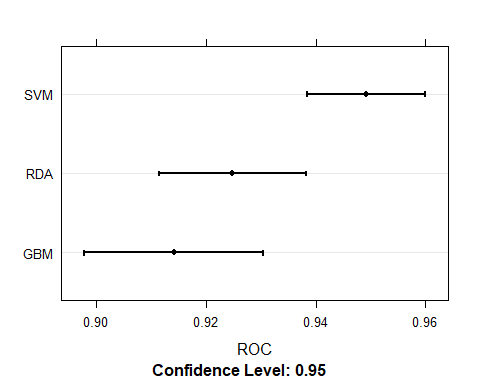
Note that, in this case, the option resamples = "final" should be user-defined in the control objects.

There are several lattice plot methods that can be used to visualize the resampling distributions: density plots, box-whisker plots, scatterplot matrices and scatterplots of summary statistics. For example:

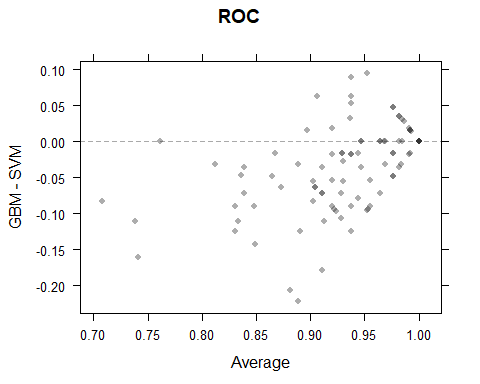
theme1 <- trellis.par.get()  
theme1$plot.symbol$col=rgb(.2, .2, .2, .4)  
theme1$plot.symbol$pch = 16  
theme1$plot.line$col = rgb(1, 0, 0, .7)  
theme1$plot.line$lwd <- 2  
trellis.par.set(theme1)  
  
bwplot(resamps, layout = c(3, 1))



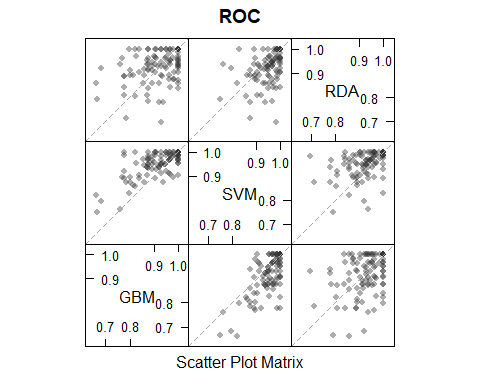
trellis.par.set(caretTheme())  
dotplot(resamps,metric="ROC")



trellis.par.set(theme1)  
xyplot(resamps,what="BlandAltman")



splom(resamps)



Other visualizations are available in densityplot.resamples and parallel.resamples.

Since models are fit on the same versions of the training data, it makes sense to make inferences on the differences between models. In this way we reduce the within-sample correlation that may exist. We can compute the differences, then use a simle t-test to evaluate the null hypothesis that there is no difference between models.

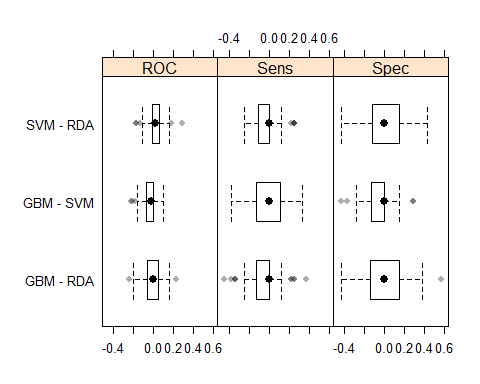
difValues <- diff(resamps)  
difValues

##   
## Call:  
## diff.resamples(x = resamps)  
##   
## Models: GBM, SVM, RDA   
## Metrics: ROC, Sens, Spec   
## Number of differences: 3   
## p-value adjustment: bonferroni

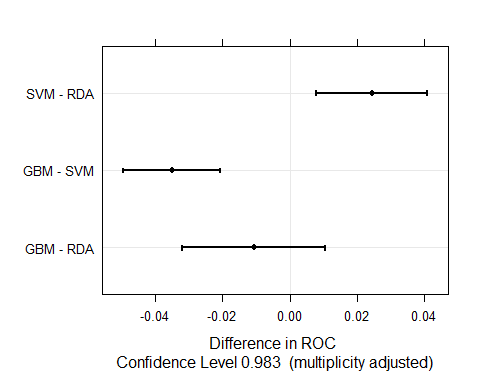
summary(difValues)

##   
## Call:  
## summary.diff.resamples(object = difValues)  
##   
## p-value adjustment: bonferroni   
## Upper diagonal: estimates of the difference  
## Lower diagonal: p-value for H0: difference = 0  
##   
## ROC   
## GBM SVM RDA   
## GBM -0.03500 -0.01067  
## SVM 1.603e-07 0.02433  
## RDA 0.670861 0.001546   
##   
## Sens   
## GBM SVM RDA   
## GBM -0.02194 -0.03708  
## SVM 0.43833 -0.01514  
## RDA 0.03554 0.60884   
##   
## Spec   
## GBM SVM RDA   
## GBM -0.023750 0.003571  
## SVM 0.1543 0.027321  
## RDA 1.0000 0.2994

trellis.par.set(theme1)  
bwplot(difValues,layout=c(3,1))



trellis.par.set(caretTheme())  
dotplot(difValues)



## Fitting models without parameter tuning

In cases where the model tuning values are known, train can be used to fit the model to the entire training set without any resampling or parameter tuning. Using the method = "none" option in trainControl can be used. For example:

fitControl <- trainControl(method="none",classProbs=T)  
  
set.seed(825)  
gbmFit4 <- train(Class ~ ., data = training,   
 method = "gbm",   
 trControl = fitControl,   
 verbose = FALSE,   
 ## Only a single model can be passed to the  
 ## function when no resampling is used:  
 tuneGrid = data.frame(interaction.depth = 4,  
 n.trees = 100,  
 shrinkage = .1,  
 n.minobsinnode = 20),  
 metric = "ROC")  
gbmFit4

## Stochastic Gradient Boosting   
##   
## 157 samples  
## 60 predictor  
## 2 classes: 'M', 'R'   
##   
## No pre-processing  
## Resampling: None

Note that plot.train, resamples, confusionMatrix.train and several other functions will not work with this object but predict.train and others will:

predict(gbmFit4, newdata=head(testing))

## [1] R R R R M M  
## Levels: M R

predict(gbmFit4, newdata=head(testing),type="prob")

## M R  
## 1 0.129181487 0.87081851  
## 2 0.039369722 0.96063028  
## 3 0.002722481 0.99727752  
## 4 0.076762607 0.92323739  
## 5 0.959529367 0.04047063  
## 6 0.622319826 0.37768017

# Available models

The models below are available in train. The code behind these protocols can be obtained using the function getModelInfo or by going to the [github repository](https://github.com/topepo/caret/tree/master/models/files).

<http://topepo.github.io/caret/available-models.html>

# Train models by tag

The following is a basic list of model types or relevant characteristics. There entries in these list are arguable. For example: random forests theoretically use **feature selection** but effectively may not, support vector machines use L2 regulazation.

### Accepts case weights

#### Adjacent categories probability model for ordinal data

method='vglmAdjCat'

Type:Type: Classification

Tuning parameters: - parallel (Parallel Curves) - link (Link Function) Required packages: VGAM

#### Bagged CART

method = 'treebag'

Type: Regression, Classification No tuning parameters for this model Required packages: ipred, plyr, e1071 A model-specific variable importance metric is available.

#### Bagged Flexible Discriminant Analysis

method = 'bagFDA'

Type: Classification Tuning parameters: - degree (Product Degree) - nprune (#Terms) Required packages: earth, mda

A model-specific variable importance metric is available. Notes: Unlike other packages used by train, the earth package is fully loaded when this model is used.

#### Bagged MARS

method='bagEarth'

Type: Regression, Classification

Tuning parameters: - nprune (#Terms) - degree (Product Degree)

Required packages: earth

A model-specific variable importance metric is available. Notes: Unlike other packages used by train, the earth package is fully loaded when this model is used.

#### Bagged MARS using gCV Pruning

method = 'bagEarthGCV'

Type: Regression, Classification

Tuning parameters: - degree (Product Degree) - Required packages: earth

A model-specific variable importance metric is available. Notes: Unlike other packages used by train, the earth package is fully loaded when this model is used.

#### Bayesian Generalized Linear Model

method='bayesglm'

Type: Regression, Classification No tuning parameters for this model Required packages: arm

<http://topepo.github.io/caret/train-models-by-tag.html#Bagging>

# Parallel Processing

In this package, resampling is promary approach for optimizing predictive models with tuninng parameters. To do this, many alternate versions of the training set are used to train the model and predict a hold-out set. This process is repeated many times to get performance estimates that generalize to new data sets.

Each of the resampled data sets is independent of the others, so there is no formal requirement that the models must be run sequentially. If a computer with multiple processor or cores is available, the computations could be spread across these “workers” to increase the computational efficiency. caret leverages one of the parallel processing frameworks in R to do just this.

The foreach package allows R code to be run either sequentially or in parallel using several different technologies, such as the multicore or Rmpi packages (see Schmidberger et al, 2009 for summaries and descriptions of the available options). There are several R packages that work with foreach to implement these techniques, such as doMC (for multicore) or doMPI (for Rmpi).

A fairly comprehensive study of the benefits of parallel processing can be found in [this blog post](http://appliedpredictivemodeling.com/blog/2018/1/17/parallel-processing).

To tune a predictive model using multiple workers, the function syntax in the caret package functions (e.g., train, rfe or sbf) do not change. A separate function is used to “register” the parallel processing technique and specify the number of workers to use. For example, to use the doParallel package with finve cores on the same machine, the package is loaded and then registed:

# install.packages("doParallel")  
library(doParallel)  
  
cl <- makePSOCKcluster(5)  
registerDoParallel(cl)  
  
## ALLsubsequent models are then run in parallel  
model <- train(y~.,data=training,method="rf")  
  
## When you are done:  
stopCluster(cl)

The syntax for other packages associated with foreach is very similar. Note that as the number of workers increases, the memory required also increase. For example, using five workers would keep a total of six versions of the data in memory. If the data are large or the computational model is demanding, performance can be affected if the amount of required memory exceeds the physical amount available. Also, for rfe and sbf, these functions may call train for some models. In this case, registering workers will actually invoke total processes.

Does this help reduce the time to fit models? A moderately sized data set (4331 rows and 8) was modeled multiple times with different number of workers for several models. Random forest was used with 2000 trees and tuned over 10 values of . Variable importance calculations were also conducted during each model fit.

Linear discriminant analysis was also run, as was a cost-sensitive radial basis function support vector machine (tuned over 15 cost values). All models were tuned using five repeats of 10-fold cross-validation. The results are shown in the figure below. The y-axis corresponds to the total execution time (encompassing model tuning and the final model fit) versus the number of workers. Random forest clearly took the longest to train and the LDA models were very computationally efficient.

The total time (in minutes) decreased as the number of workers increase but stabilized around seven workers. The data for this plot were generated in a randomized fashion so that there should be no bias in the run order. The bottom right panel shows the speed-up which is the sequential time divided by the parallel time. For example, a speed-up of three indicates that the parallel version was three times faster than the sequential version. At best, parallelization can achieve linear speed-ups; that is, for M workers, the parallel time is 1/M. For these models, the speed-up is close to linear until four or five workers are used. After this, there is a small improvement in performance. Since LDA is already computationally efficient, the speed-up levels off more rapidly than the other models. While not linear, the decrease in execution time is helpful - a nearly 10 hour model fit was decreased to about 90 minutes.

Note that some models, especially those using the RWeka package, may not be able to be run in parallel due to the underlying code structure.

train, rfe, sbf, bag and avNNet were given an additional argument in their respective control files called allowParallel that defaults to TRUE. When TRUE, the code will be executed in parallel if a parallel backend (e.g. doMC) is registered. When allowParallel = FALSE, the parallel backend is always ignored. The use case is when rfe or sbf calls train. If a parallel backend with P processors is being used, the combination of these functions will create P2 processes. Since some operations benefit more from parallelization than others, the user has the ability to concentrate computing resources for specific functions.

One additional “trick” that train exploits to increase computational efficiency is to use sub-models; a single model fit can produce predictions for multiple tuning parameters. For example, in most implementations of boosted models, a model trained on B boosting iterations can produce predictions for models for iterations less than B. Suppose a gbm model was tuned over the following grid

gbmGrid <- expand.grid(interaction.depth = c(1, 5, 9),  
 n.trees = (1:15)\*100,  
 shrinkage = 0.1,  
 n.minobsinnode = 20)

In reality, train only created objects for 3 models and derived the other predictions from these objects. This trick is used for the following models: ada, AdaBag, AdaBoost.M1, bagEarth, blackboost, blasso, BstLm, bstSm, bstTree, C5.0, C5.0Cost, cubist, earth, enet, foba, gamboost, gbm, glmboost, glmnet, kernelpls, lars, lars2, lasso, lda2, leapBackward, leapForward, leapSeq, LogitBoost, pam, partDSA, pcr, PenalizedLDA, pls, relaxo, rfRules, rotationForest, rotationForestCp, rpart, rpart2, rpartCost, simpls, spikeslab, superpc, widekernelpls, xgbDART, xgbTree.

# Random hyperparameter search

The default method for optimizing tuning parameters in train is to use a [grid search](http://topepo.github.io/caret/model-training-and-tuning.html#grids). This approach is usually effective but, in cases when there are many tuning parameters, it can be inefficient. An alternative is to use a combination of grid search and racing. Another is to use a random selection of tuning parameter combinations to cover the parameter space to a lesser extent.

There are a number of models where this can be beneficial in finding reasonable values of the tuning parameters in a relatively short time. However, there are some models where the efficiency in a small search field can cancel out other optimizations. For example, a number of models in caret utilize the “sub-model trick” where tuning parameter combinations are evaluated, potentially far fewer than M model fits are required. This approach is best leveraged when a simple grid search is used. For this reason, it may be inefficient to use random search for the following model codes: ada, AdaBag, AdaBoost.M1, bagEarth, blackboost, blasso, BstLm, bstSm, bstTree, C5.0, C5.0Cost, cubist, earth, enet, foba, gamboost, gbm, glmboost, glmnet, kernelpls, lars, lars2, lasso, lda2, leapBackward, leapForward, leapSeq, LogitBoost, pam, partDSA, pcr, PenalizedLDA, pls, relaxo, rfRules, rotationForest, rotationForestCp, rpart, rpart2, rpartCost, simpls, spikeslab, superpc, widekernelpls, xgbDART, xgbTree.

Finally, many of the models wrapped by train have a small number of parameters. The average number of parameters is 2.

To use random search, another option is available in trainControl called search. Possible values of this argument are "grid" and "random". The built-in models contained in caret contain code to generate random tuning parameter combinations. The total number of unique combinations is specified by the tuneLength option to train.

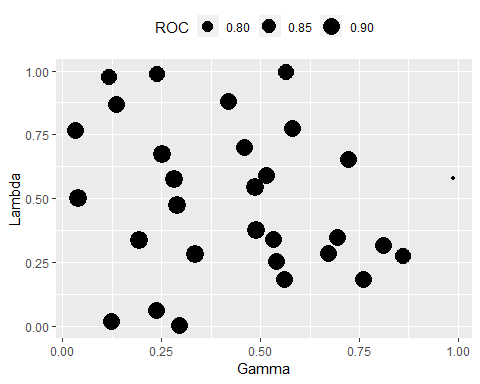
Again, we will use the sonar data from the previous training page to demonstrate the method with a regularized discriminant analysis by looking at a total of 30 tuning parameters combinations:

library(mlbench)  
data(Sonar)  
  
library(caret)  
set.seed(998)  
  
inTraining <- createDataPartition(Sonar$Class,p=.75,list=FALSE)  
training <- Sonar[inTraining,]  
testing <- Sonar[-inTraining,]  
  
fitControl <- trainControl(method="repeatedcv",  
 number=10,  
 repeats=10,  
 classProbs=T,  
 summaryFunction = twoClassSummary,  
 search="random"  
 )  
  
set.seed(825)  
rda\_fit <- train(Class~.,data=training,  
 method="rda",  
 metric="ROC",  
 tuneLength=30,  
 trControl=fitControl)  
rda\_fit

## Regularized Discriminant Analysis   
##   
## 157 samples  
## 60 predictor  
## 2 classes: 'M', 'R'   
##   
## No pre-processing  
## Resampling: Cross-Validated (10 fold, repeated 10 times)   
## Summary of sample sizes: 141, 141, 142, 141, 141, 142, ...   
## Resampling results across tuning parameters:  
##   
## gamma lambda ROC Sens Spec   
## 0.03177874 0.767664044 0.9168502 0.8998611 0.8182143  
## 0.03868192 0.499283304 0.9199752 0.9001389 0.8287500  
## 0.11834801 0.974493793 0.8831200 0.8469444 0.7630357  
## 0.12391186 0.018063038 0.9090377 0.8851389 0.7975000  
## 0.13442487 0.868918547 0.9053943 0.9012500 0.7755357  
## 0.19249104 0.335761243 0.9290451 0.9184722 0.8151786  
## 0.23568481 0.064135040 0.9126414 0.8923611 0.7782143  
## 0.23814584 0.986270274 0.8805159 0.8522222 0.7723214  
## 0.25082994 0.674919744 0.9274182 0.9337500 0.7996429  
## 0.28285931 0.576888058 0.9275099 0.9225000 0.7969643  
## 0.29099029 0.474277013 0.9261954 0.9237500 0.8051786  
## 0.29601805 0.002963208 0.9075967 0.8850000 0.7626786  
## 0.33633553 0.283586169 0.9232465 0.9187500 0.7855357  
## 0.41798776 0.881581948 0.8971677 0.8883333 0.7778571  
## 0.45885413 0.701431940 0.9130208 0.9191667 0.7678571  
## 0.48684373 0.545997273 0.9199380 0.9177778 0.7635714  
## 0.48845661 0.377704420 0.9178175 0.9105556 0.7633929  
## 0.51491517 0.592224877 0.9155010 0.9140278 0.7666071  
## 0.53206420 0.339941226 0.9154291 0.9056944 0.7623214  
## 0.54020648 0.253930177 0.9131448 0.9043056 0.7626786  
## 0.56009903 0.183772303 0.9113790 0.8958333 0.7671429  
## 0.56472058 0.995162379 0.8784102 0.8244444 0.8008929  
## 0.58045730 0.773613530 0.9015104 0.8868056 0.7694643  
## 0.67085142 0.287354882 0.9088269 0.9031944 0.7541071  
## 0.69503284 0.348973440 0.9077133 0.9105556 0.7607143  
## 0.72206263 0.653406920 0.9003894 0.8908333 0.7676786  
## 0.76035804 0.183676074 0.9026513 0.9018056 0.7414286  
## 0.81091174 0.317173641 0.8953100 0.9022222 0.7308929  
## 0.86234436 0.272931617 0.8841691 0.8976389 0.7196429  
## 0.98847635 0.580160726 0.7588616 0.7179167 0.6787500  
##   
## ROC was used to select the optimal model using the largest value.  
## The final values used for the model were gamma = 0.192491 and lambda  
## = 0.3357612.

There is currently only a ggplot method (instead of a basic plot method). The results of this function with random searching depends on the number and type of tuning parameters. In this case, it produces a scatter plot of the continuous parameters.

ggplot(rda\_fit)+  
 theme(legend.position = "top")



# Subsampling for class imbalances

In classification problems, a disparity in the frequencies of the observed classes can have a significant negative impact on model fitting. One technique for resolving such a class imbalance is to subsample the training data in a manner that mitigates the issues. Examples of sampling methods for this purpose are:

* *down-sampling*: randomly subset all the classes in the training set so that their class frequencies match the least prevalent class. For example, suppose that 80% of the training set samples are the first class and the remaining 20% are in the second class. Down-sampling would randomly sample the first class to be the same size as the second class (so that only 40% of the total training set is used to fit the model). **caret** contains a function (downSample) to do this.
* *up-sampling*: randomly sample (with replacement) the minority class to be the same size as the majority class. caret contains a function (upSample) to do this.
* *hybrid methods*: techniques such as SMOTE and ROSE down-sample the majority class and synthesize new data points in the minority class. There are two packages (**DMwR** and \*\*ROSE\*) that implement these procedures.

Note that this type of sampling is different from splitting the data into a training and test set. You would never want to artificially balance the test set; its class frequencies should be in-line with what one would see “in the wild”. Also, the above procedures are independent of resampling methods such as cross-validation and the bootstrap.

In practice, one could take the training set and, before model fitting, sample the data. There are two issues with this approach

* Firstly, during model tuning the holdout samples generated during resampling are also glanced and may not reflect the class imbalance that future predictions would encounter. This is likely to lead to overly optimistic estimates of performance.
* Secondly, the subsampling process will probably induce more model uncertainty. Would the model results differ under a different subsample? As above, the resampling statistics are more likely to make the model appear more effective than it actually is.

The alternative is to include the subsampling inside of the usual resampling procedure. This is also advocated for pre-process and featur selection steps too. The two disadvantages are that it might increase computational times and that it might also complicate the analysis in other ways (see the section [below](http://topepo.github.io/caret/subsampling-for-class-imbalances.html#complications) about the pitfalls).

## Subsampling techniques

To illustrate these mothods, let’s simulate some data within a class inbalance using this method. We will stimulate a training set where each contains 10000 samples and a minority class rate of about 5.9%:

library(caret)  
  
set.seed(2969)  
imbal\_train <- twoClassSim(10000,intercept=-20,linearVars = 20)  
imbal\_test <- twoClassSim(10000,intercept=-20,linearVars = 20)  
table(imbal\_train$Class)

##   
## Class1 Class2   
## 9411 589

Let’s create different versions of the training set prior to model tuning:

set.seed(9560)  
down\_train <- downSample(x=imbal\_train[,-ncol(imbal\_train)],  
 y=imbal\_train$Class)  
table(down\_train$Class)

##   
## Class1 Class2   
## 589 589

set.seed(9560)  
up\_train <- upSample(x=imbal\_train[,-ncol(imbal\_train)],  
 y=imbal\_train$Class)  
table(up\_train$Class)

##   
## Class1 Class2   
## 9411 9411

# install.packages("DMwR")  
library(DMwR)

## Warning: package 'DMwR' was built under R version 3.5.1

## Loading required package: grid

set.seed(9560)  
smote\_train <- SMOTE(Class~., data=imbal\_train)  
table(smote\_train$Class)

##   
## Class1 Class2   
## 2356 1767

# install.packages("ROSE")  
library(ROSE)

## Warning: package 'ROSE' was built under R version 3.5.1

## Loaded ROSE 0.0-3

set.seed(9560)  
rose\_train <- ROSE(Class~., data=imbal\_train)$data  
table(rose\_train$Class)

##   
## Class1 Class2   
## 4939 5061

For these data, we’ll use a bagged classification and estimate the area under the ROC curve using five repeats of 10-fold CV.

ctrl <- trainControl(method="repeatedcv",repeats=5,  
 classProbs=T,  
 summaryFunction=twoClassSummary)  
  
set.seed(5627)  
orig\_fit <- train(Class~., data=imbal\_train,  
 method="treebag",  
 nbagg=50,  
 metric="ROC",  
 trControl=ctrl)  
  
  
set.seed(5627)  
down\_outside <- train(Class~., data=down\_train,  
 method="treebag",  
 nbagg=50,  
 metric="ROC",  
 trControl=ctrl)  
  
set.seed(5627)  
up\_outside <- train(Class ~ ., data = up\_train,   
 method = "treebag",  
 nbagg = 50,  
 metric = "ROC",  
 trControl = ctrl)  
  
set.seed(5627)  
rose\_outside <- train(Class ~ ., data = rose\_train,   
 method = "treebag",  
 nbagg = 50,  
 metric = "ROC",  
 trControl = ctrl)  
  
set.seed(5627)  
smote\_outside <- train(Class ~ ., data = smote\_train,   
 method = "treebag",  
 nbagg = 50,  
 metric = "ROC",  
 trControl = ctrl)

We will collate the resampling results and create a wrapper to estimate the test set performance:

outside\_models <- list(original=orig\_fit,  
 down=down\_outside,  
 up=up\_outside,  
 SMOTE=smote\_outside,  
 ROSE=rose\_outside)  
  
outside\_resampling <- resamples(outside\_models)  
  
test\_rec <- function(model,data){  
 library(pROC)  
 roc\_obj <- roc(data$Class,   
 predict(model, data, type = "prob")[, "Class1"],  
 levels = c("Class2", "Class1"))  
 ci(roc\_obj)  
}  
  
outside\_test <- lapply(outside\_models, test\_roc, data = imbal\_test)  
outside\_test <- lapply(outside\_test, as.vector)  
outside\_test <- do.call("rbind", outside\_test)  
colnames(outside\_test) <- c("lower", "ROC", "upper")  
outside\_test <- as.data.frame(outside\_test)  
  
summary(outside\_resampling, metric = "ROC")

outside\_test

The training and test set estimates for the area under the ROC curve do not appear to correlate. Based on the resampling results, one would infer that up-sampling is nearly perfect and that ROSE does relatively poorly. The reason that up-sampling appears to perform so well is that the samples in the majority class are replicated and have a large potential to be in both the model building and hold-out sets. In essence, the hold-outs here are not truly independent samples.

In reality, all of the sampling methods do about the same (based on the test set). The statistics for the basic model fit with no sampling are fairly in-line with one another (0.939 via resampling and 0.922 for the test set).

## Subsampling during resampling

Recent versions of **caret** allow the user to specify subsampling when using train so that it is conducted inside of resampling. All four methods shown above can be accessed with the basic package using simple syntax. If you want to use your own technique, or want to change some of the parameters for SMOTE or ROSE, the last section below shows how to use custom subsampling.

The way to enable subsampling is to use yet another option in trainControl called sampling. The most basic syntax is to use a character string with the name of the sampling method, either "down", "up", "smote", or "rose". Note that you will need to have the DMwR and ROSE packages installed to use **SMOTE** and **ROSE**, respectively.

One complication is related to pre-processing. Should the subsampling occur before or after the pre-processing? or example, if you down-sample the data and using PCA for signal extraction, should the loadings be estimated from the entire training set? The estimate is potentially better since the entire training set is being used but the subsample may happen to capture a small potion of the PCA space. There isn’t any obvious answer.

The default behavior is to subsample the data prior to pre-processing. This can be easily changed and an example is given below.

Now let’s re-run our bagged tree models while sampling inside of cross-validation:

ctrl <- trainControl(method = "repeatedcv", repeats = 5,  
 classProbs = TRUE,  
 summaryFunction = twoClassSummary,  
 ## new option here:  
 sampling = "down")  
  
set.seed(5627)  
down\_inside <- train(Class ~ ., data = imbal\_train,  
 method = "treebag",  
 nbagg = 50,  
 metric = "ROC",  
 trControl = ctrl)  
  
## now just change that option  
ctrl$sampling <- "up"  
  
set.seed(5627)  
up\_inside <- train(Class ~ ., data = imbal\_train,  
 method = "treebag",  
 nbagg = 50,  
 metric = "ROC",  
 trControl = ctrl)  
  
ctrl$sampling <- "rose"  
  
set.seed(5627)  
rose\_inside <- train(Class ~ ., data = imbal\_train,  
 method = "treebag",  
 nbagg = 50,  
 metric = "ROC",  
 trControl = ctrl)  
  
ctrl$sampling <- "smote"  
  
set.seed(5627)  
smote\_inside <- train(Class ~ ., data = imbal\_train,  
 method = "treebag",  
 nbagg = 50,  
 metric = "ROC",  
 trControl = ctrl)

Here are the resampling and test set results:

inside\_models <- list(  
 original=orig\_fit,  
 down=down\_inside,  
 up=up\_inside,  
 SMOTE=smote\_inside,  
 ROSE=rose\_inside  
)  
inside\_test <- lapply(inside\_models,test\_roc,data=imbal\_test)  
inside\_test <- lapply(inside\_test,as.vector)  
inside\_test <- do.call("rbind",inside\_test)  
colnames(inside\_test) <- c("lower","ROC","upper")  
  
inside\_test <- as.data.frame(inside\_test)  
summary(inside\_resampling,metric="ROC")

inside\_test

The figure below shows the difference in the area under the ROC curve and the test set results for the approaches shown here. Repeating the subsampling procedures for every resample produces results that are more consistent with the test set.

## Complications

The user should be aware that there are a few things that can happening when subsampling that can cause issues in their code. As previously mentioned, when sampling occurs in relation to pre-processing is one such issue. Others are: - Sparsely represented categories in factor variables may turn into zero-variance predictors or may be completely sampled out of the model. - The underlying functions that do the sampling (e.g. SMOTE, downSample, etc) operate in very different ways and this can affect your results. For example, SMOTE and ROSE will convert your predictor input argument into a data frame (even if you start with a matrix). - Currently, sample weights are not supported with sub-sampling. - If you use tuneLength to specify the search grid, understand that the data that is used to determine the grid has not been sampled. In most cases, this will not matter but if the grid creation process is affected by the sample size, you may end up using a sub-optimal tuning grid. - For some models that require more samples than parameters, a reduction in the sample size may prevent you from being able to fit the model.

## Using custom subsampling techniques

Users have the ability to create their own type of subsampling procedure. To do this, alternative syntax is used with the sampling argument of the trainControl. Previously, we used a simple string as the value of this argument. Another way to specify the argument is to use a list with three (named) elements:

* The name value is a character string used when the train object is printed. It can be any string.
* The func element is a function that does the subsampling. It should have arguments called x and y that will contain the predictors and outcome data, respectively. The function should return a list with elements of the same name.
* The first element is a single logical value that indicates whether the subsampling should occur first relative to pre-process. A value of FALSE means that the subsampling function will receive the sampled versions of x and y.

For example, here is what the list version of the sampling argument looks like when simple down-sampling is used:

down\_inside$control$sampling

As another example, suppose we want to use SMOTE but use 10 nearest neighbors instead of the default of 5. To do this, we can create a simple wrapper around the SMOTE function and call this instead:

smotest <- list(name = "SMOTE with more neighbors!",  
 func = function (x, y) {  
 library(DMwR)  
 dat <- if (is.data.frame(x)) x else as.data.frame(x)  
 dat$.y <- y  
 dat <- SMOTE(.y ~ ., data = dat, k = 10)  
 list(x = dat[, !grepl(".y", colnames(dat), fixed = TRUE)],   
 y = dat$.y)  
 },  
 first = TRUE)

The control object would then be:

ctrl <- trainControl(method = "repeatedcv", repeats = 5,  
 classProbs = TRUE,  
 summaryFunction = twoClassSummary,  
 sampling = smotest)

# Using Recipes with train

Modeling functions in R let you specifc a model using a formula, the x/y interface, or both. Formulas are good because they will handle a lot of minutia for your (e.g., dummy variables, interaction etc.) so you don’t have to get your hands dirty.

They work [pretty well](https://rviews.rstudio.com/2017/02/01/the-r-formula-method-the-good-parts/), but also [have limitations](https://rviews.rstudio.com/2017/03/01/the-r-formula-method-the-bad-parts/),Their biggest issue is that not all modeling functions have a formula interface (although train helps solve that).

Recipes are a third method for specifying model terms but also allow for a broad set of preprocessing options for encoding, manupulating, and transforming data. They cover a lot of techniques that formulas cannnot naturally.

Recipes can be built incrementally in a way similar to how dplyr or ggplot2 are created. The package website has example of how to use the package and lists the possible techniques (called **steps**). A recipe can then be handed to train in lieu of a formula.

## Why should you learn this?

Here are two reasons.

### More versatile toos for preprocessing data

caret’S preprocessing tools have a lot of options but the list is not exhaustive and they will only be called in a specific order. If you would lik - a broader set of options, - the ability to write your own preprocessing tools, or - to call them in the order that you desire

Then you can use a recipe to do that.

### Using additional data to measure performance

In most modeling functions, including train, most variables are consigned to be either predictors or outcomes. For recipes, you might want to have specific columns of your data set be available when you compute how well the model is performing, such as:

* if different stratification variables(e.g., patients, ZIP codes etc.) are required to do correct summaries or;
* ancillary data might be needed to compute the expected profit or loss based on the model results

To get these data properly, they need to be made available and handled the same way as all of the other data. This means they should be sub- or **resampled** as all of the other data. Recipes let your do that.

## An example

The QSARdata package contains several chemistry data sets. These data sets have rows for different pontential drugs(called “compounds” here). For each compound, some important characteristic is measured. This illustration will use the AquaticTox data. The outcome is called **“Acticvity”** is a measure of how harmful the compound might be people. We want to predict this during the drug discovery phase in R&D . To do this, a set of *molecular descritors* are computed based on the compounds formula. There are a lot of different types of these, and we will use the 2-dimensional MOE descriptor set. First let’s load the package and get the data together:

library(caret)  
library(recipes)

## Warning: package 'recipes' was built under R version 3.5.1

## Loading required package: broom

## Warning: package 'broom' was built under R version 3.5.1

##   
## Attaching package: 'broom'

## The following object is masked from 'package:DMwR':  
##   
## bootstrap

##   
## Attaching package: 'recipes'

## The following object is masked from 'package:stringr':  
##   
## fixed

## The following object is masked from 'package:stats':  
##   
## step

library(dplyr)  
library(QSARdata)  
  
data(AquaticTox)  
tox <- AquaticTox\_moe2D  
ncol(tox)

## [1] 221

# Add the outcome variable to the data frame  
tox$Activity <- AquaticTox\_Outcome$Activity  
  
head(tox,3)

## Molecule moeGao\_Abra\_L  
## 1 (d)-limonene 4.729  
## 2 111-trichloro-2-methyl-2-propanolol(chlorobytanol) 4.226  
## 3 111-trichloroethane 2.790  
## moeGao\_Abra\_R moeGao\_Abra\_acidity moeGao\_Abra\_basicity moeGao\_Abra\_pi  
## 1 0.512 0.030 0.126 0.330  
## 2 0.527 0.375 0.452 0.712  
## 3 0.383 0.030 0.101 0.469  
## moe2D\_BCUT\_PEOE\_0 moe2D\_BCUT\_PEOE\_1 moe2D\_BCUT\_PEOE\_2 moe2D\_BCUT\_PEOE\_3  
## 1 -2.696 -0.009128 0.04606 2.639  
## 2 -2.556 -0.080280 0.05082 2.615  
## 3 -2.262 -0.083670 0.02702 2.350  
## moe2D\_BCUT\_SLOGP\_0 moe2D\_BCUT\_SLOGP\_1 moe2D\_BCUT\_SLOGP\_2  
## 1 -2.554 0.12300 0.1363  
## 2 -2.557 0.06489 0.6895  
## 3 -2.066 0.12300 0.6895  
## moe2D\_BCUT\_SLOGP\_3 moe2D\_BCUT\_SMR\_0 moe2D\_BCUT\_SMR\_1 moe2D\_BCUT\_SMR\_2  
## 1 2.784 -2.427 0.1057 0.2077  
## 2 2.681 -2.334 0.1057 0.5853  
## 3 2.596 -2.027 0.1057 0.5853  
## moe2D\_BCUT\_SMR\_3 moe2D\_GCUT\_PEOE\_0 moe2D\_GCUT\_PEOE\_1 moe2D\_GCUT\_PEOE\_2  
## 1 2.912 -0.7881 -0.2273 -0.1139  
## 2 2.840 -0.7825 -0.2403 -0.1342  
## 3 2.611 -0.6887 -0.2437 -0.1330  
## moe2D\_GCUT\_PEOE\_3 moe2D\_GCUT\_SLOGP\_0 moe2D\_GCUT\_SLOGP\_1  
## 1 2.451 -0.6697 -0.08693  
## 2 2.056 -0.8724 -0.08735  
## 3 1.559 -0.5478 -0.03700  
## moe2D\_GCUT\_SLOGP\_2 moe2D\_GCUT\_SLOGP\_3 moe2D\_GCUT\_SMR\_0 moe2D\_GCUT\_SMR\_1  
## 1 -0.0370 2.598 -0.5188 -0.0543  
## 2 0.3012 2.188 -0.5229 -0.0543  
## 3 0.5295 1.871 -0.4864 -0.0543  
## moe2D\_GCUT\_SMR\_2 moe2D\_GCUT\_SMR\_3 moe2D\_Kier1 moe2D\_Kier2 moe2D\_Kier3  
## 1 -0.009485 2.704 8.1 3.408 2.286  
## 2 0.425300 2.297 8.0 1.750 2.222  
## 3 0.425300 1.858 5.0 1.000 0.000  
## moe2D\_KierA1 moe2D\_KierA2 moe2D\_KierA3 moe2D\_KierFlex moe2D\_PEOE\_PC.  
## 1 6.817 2.807 1.850 1.914 0.5430  
## 2 13.390 3.154 4.267 5.278 0.6904  
## 3 11.470 2.844 0.000 6.523 0.2690  
## moe2D\_PEOE\_PC..1 moe2D\_PEOE\_RPC. moe2D\_PEOE\_RPC..1 moe2D\_PEOE\_VSA.0  
## 1 -0.5430 0.1053 0.1831 55.45  
## 2 -0.6904 0.3160 0.5575 25.59  
## 3 -0.2690 0.6986 0.3111 12.80  
## moe2D\_PEOE\_VSA.1 moe2D\_PEOE\_VSA.2 moe2D\_PEOE\_VSA.3 moe2D\_PEOE\_VSA.4  
## 1 12.8 0 0 0.00  
## 2 0.0 0 0 10.32  
## 3 0.0 0 0 0.00  
## moe2D\_PEOE\_VSA.5 moe2D\_PEOE\_VSA.6 moe2D\_PEOE\_VSA.0.1 moe2D\_PEOE\_VSA.1.1  
## 1 0 0 63.60 38.97  
## 2 0 0 37.50 88.75  
## 3 0 0 18.75 88.75  
## moe2D\_PEOE\_VSA.2.1 moe2D\_PEOE\_VSA.3.1 moe2D\_PEOE\_VSA.4.1  
## 1 0 0 0  
## 2 0 0 0  
## 3 0 0 0  
## moe2D\_PEOE\_VSA.5.1 moe2D\_PEOE\_VSA.6.1 moe2D\_PEOE\_VSA\_FHYD  
## 1 0 0.000 1.0000  
## 2 0 7.768 0.8935  
## 3 0 0.000 1.0000  
## moe2D\_PEOE\_VSA\_FNEG moe2D\_PEOE\_VSA\_FPNEG moe2D\_PEOE\_VSA\_FPOL  
## 1 0.6005 0.00000 0.0000  
## 2 0.7886 0.04571 0.1065  
## 3 0.8936 0.00000 0.0000  
## moe2D\_PEOE\_VSA\_FPOS moe2D\_PEOE\_VSA\_FPPOS moe2D\_PEOE\_VSA\_HYD  
## 1 0.3995 0.00000 170.8  
## 2 0.2114 0.06076 151.8  
## 3 0.1064 0.00000 120.3  
## moe2D\_PEOE\_VSA\_NEG moe2D\_PEOE\_VSA\_PNEG moe2D\_PEOE\_VSA\_POL  
## 1 102.6 0.000 0.00  
## 2 134.0 7.768 18.09  
## 3 107.5 0.000 0.00  
## moe2D\_PEOE\_VSA\_POS moe2D\_PEOE\_VSA\_PPOS moe2D\_Q\_PC. moe2D\_Q\_PC..1  
## 1 68.25 0.00 1.141 -1.141  
## 2 35.92 10.32 1.550 -1.550  
## 3 12.80 0.00 0.870 -0.870  
## moe2D\_Q\_RPC. moe2D\_Q\_RPC..1 moe2D\_Q\_VSA\_FHYD moe2D\_Q\_VSA\_FNEG  
## 1 0.1315 0.2629 0.7718 0.2282  
## 2 0.5613 0.4387 0.3713 0.5680  
## 3 1.0000 0.3333 0.2622 0.7378  
## moe2D\_Q\_VSA\_FPNEG moe2D\_Q\_VSA\_FPOL moe2D\_Q\_VSA\_FPOS moe2D\_Q\_VSA\_FPPOS  
## 1 0.2282 0.2282 0.7718 0.00000  
## 2 0.5680 0.6287 0.4320 0.06076  
## 3 0.7378 0.7378 0.2622 0.00000  
## moe2D\_Q\_VSA\_HYD moe2D\_Q\_VSA\_NEG moe2D\_Q\_VSA\_PNEG moe2D\_Q\_VSA\_POL  
## 1 131.80 38.97 38.97 38.97  
## 2 63.09 96.51 96.51 106.80  
## 3 31.54 88.75 88.75 88.75  
## moe2D\_Q\_VSA\_POS moe2D\_Q\_VSA\_PPOS moe2D\_SMR moe2D\_SMR\_VSA0 moe2D\_SMR\_VSA1  
## 1 131.80 0.00 4.591 0 0.00  
## 2 73.41 10.32 3.655 0 25.39  
## 3 31.54 0.00 2.593 0 0.00  
## moe2D\_SMR\_VSA2 moe2D\_SMR\_VSA3 moe2D\_SMR\_VSA4 moe2D\_SMR\_VSA5  
## 1 4.411 6.371 0 74.25  
## 2 0.000 0.000 0 0.00  
## 3 0.000 0.000 0 0.00  
## moe2D\_SMR\_VSA6 moe2D\_SMR\_VSA7 moe2D\_SlogP moe2D\_SlogP\_VSA0  
## 1 0 98.75 3.309 0.00  
## 2 0 184.10 2.547 25.39  
## 3 0 150.80 2.796 0.00  
## moe2D\_SlogP\_VSA1 moe2D\_SlogP\_VSA2 moe2D\_SlogP\_VSA3 moe2D\_SlogP\_VSA4  
## 1 0 0 0 4.411  
## 2 0 0 0 0.000  
## 3 0 0 0 0.000  
## moe2D\_SlogP\_VSA5 moe2D\_SlogP\_VSA6 moe2D\_SlogP\_VSA7 moe2D\_SlogP\_VSA8  
## 1 6.371 0 17.64 56.61  
## 2 0.000 0 0.00 0.00  
## 3 0.000 0 0.00 0.00  
## moe2D\_SlogP\_VSA9 moe2D\_TPSA moe2D\_VAdjEq moe2D\_VAdjMa moe2D\_VDistEq  
## 1 98.75 0.00 0.7219 4.322 2.344  
## 2 184.10 20.23 0.7579 3.807 1.550  
## 3 150.80 0.00 0.9044 3.000 0.971  
## moe2D\_VDistMa moe2D\_Weight moe2D\_a\_IC moe2D\_a\_ICM moe2D\_a\_acc  
## 1 6.306 136.2 24.99 0.9612 0  
## 2 5.706 177.5 26.20 1.7460 1  
## 3 4.250 133.4 12.49 1.5610 0  
## moe2D\_a\_aro moe2D\_a\_count moe2D\_a\_don moe2D\_a\_heavy moe2D\_a\_hyd  
## 1 0 26 0 10 10  
## 2 0 15 1 8 6  
## 3 0 8 0 5 4  
## moe2D\_a\_nBr moe2D\_a\_nC moe2D\_a\_nCl moe2D\_a\_nF moe2D\_a\_nH moe2D\_a\_nN  
## 1 0 10 0 0 16 0  
## 2 0 4 3 0 7 0  
## 3 0 2 3 0 3 0  
## moe2D\_a\_nO moe2D\_apol moe2D\_b\_1rotN moe2D\_b\_1rotR moe2D\_b\_ar  
## 1 0 28.27 1 0.1 0  
## 2 1 19.05 0 0.0 0  
## 3 0 12.06 0 0.0 0  
## moe2D\_b\_count moe2D\_b\_double moe2D\_b\_heavy moe2D\_b\_rotN moe2D\_b\_rotR  
## 1 26 2 10 1 0.1000  
## 2 14 0 7 1 0.1429  
## 3 7 0 4 0 0.0000  
## moe2D\_b\_single moe2D\_b\_triple moe2D\_balabanJ moe2D\_bpol moe2D\_chi0  
## 1 24 0 2.281 17.490 7.56  
## 2 14 0 4.020 8.912 7.00  
## 3 7 0 3.024 4.540 4.50  
## moe2D\_chi0\_C moe2D\_chi0v moe2D\_chi0v\_C moe2D\_chi1 moe2D\_chi1\_C  
## 1 7.56 6.983 6.983 4.698 4.698  
## 2 3.00 6.849 3.000 3.250 1.250  
## 3 1.50 4.902 1.500 2.000 0.500  
## moe2D\_chi1v moe2D\_chi1v\_C moeGao\_chi2\_C moeGao\_chi2v\_C moeGao\_chi3c  
## 1 4.009 4.009 4.264 3.368 0.7887  
## 2 3.174 1.250 1.000 1.000 2.5000  
## 3 2.201 0.500 0.000 0.000 2.0000  
## moeGao\_chi3c\_C moeGao\_chi3cv moeGao\_chi3cv\_C moeGao\_chi3p moeGao\_chi3p\_C  
## 1 0.7887 0.5526 0.5526 3.003 3.003  
## 2 0.2500 2.3900 0.2500 2.250 0.000  
## 3 0.0000 2.6580 0.0000 0.000 0.000  
## moeGao\_chi3pv moeGao\_chi3pv\_C moeGao\_chi4c moeGao\_chi4c\_C moeGao\_chi4ca  
## 1 2.335 2.335 0.0 0 1.953  
## 2 2.081 0.000 0.5 0 0.000  
## 3 0.000 0.000 0.5 0 0.000  
## moeGao\_chi4ca\_C moeGao\_chi4cav moeGao\_chi4cav\_C moeGao\_chi4cv  
## 1 1.953 1.413 1.413 0.0000  
## 2 0.000 0.000 0.000 0.4763  
## 3 0.000 0.000 0.000 0.7289  
## moeGao\_chi4cv\_C moeGao\_chi4pc moeGao\_chi4pc\_C moeGao\_chi4pcv  
## 1 0 1.449 1.449 1.009  
## 2 0 4.500 0.000 3.971  
## 3 0 0.000 0.000 0.000  
## moeGao\_chi4pcv\_C moe2D\_chiral moe2D\_chiral\_u moe2D\_density  
## 1 1.009 1 1 0.586  
## 2 0.000 0 0 1.056  
## 3 0.000 0 0 1.206  
## moe2D\_diameter moe2D\_kS\_aaCH moe2D\_kS\_aaN moe2D\_kS\_aaaC moe2D\_kS\_aasC  
## 1 6 0 0 0 0  
## 2 3 0 0 0 0  
## 3 2 0 0 0 0  
## moe2D\_kS\_dCH2 moe2D\_kS\_dO moe2D\_kS\_dsCH moe2D\_kS\_dsN moe2D\_kS\_dssC  
## 1 3.975 0 2.35 0 2.904  
## 2 0.000 0 0.00 0 0.000  
## 3 0.000 0 0.00 0 0.000  
## moe2D\_kS\_sBr moe2D\_kS\_sCH3 moe2D\_kS\_sCl moe2D\_kS\_sF moe2D\_kS\_sNH2  
## 1 0 4.355 0.00 0 0  
## 2 0 2.861 15.87 0 0  
## 3 0 1.484 15.18 0 0  
## moe2D\_kS\_sOH moe2D\_kS\_ssCH2 moe2D\_kS\_ssNH moe2D\_kS\_ssO moe2D\_kS\_sssCH  
## 1 0.000 3.816 0 0 0.7673  
## 2 8.958 0.000 0 0 0.0000  
## 3 0.000 0.000 0 0 0.0000  
## moe2D\_kS\_sssN moe2D\_kS\_ssssC moe2D\_kS\_tCH moe2D\_kS\_tNH moe2D\_kS\_tsC  
## 1 0 0.000 0 0 0  
## 2 0 -2.856 0 0 0  
## 3 0 -1.083 0 0 0  
## moe2D\_lip\_acc moe2D\_lip\_don moe2D\_lip\_violation moe2D\_logP.o.w.  
## 1 0 0 0 3.604  
## 2 1 1 0 1.973  
## 3 0 0 0 2.139  
## moe2D\_logS moe2D\_mr moe2D\_opr\_brigid moe2D\_opr\_leadlike moe2D\_opr\_nring  
## 1 -3.005 4.500 7 1 1  
## 2 -2.450 3.554 0 1 0  
## 3 -2.326 2.471 0 1 0  
## moe2D\_opr\_nrot moe2D\_opr\_violation moe2D\_petitjean moe2D\_petitjeanSC  
## 1 2 0 0.5000 1.0  
## 2 0 0 0.3333 0.5  
## 3 0 0 0.5000 1.0  
## moe2D\_radius moe2D\_reactive moe2D\_rings moe2D\_vdw\_area moe2D\_vdw\_vol  
## 1 3 0 1 170.8 232.5  
## 2 2 0 0 169.9 168.0  
## 3 1 0 0 120.3 110.6  
## moe2D\_vsa\_acc moe2D\_vsa\_don moe2D\_vsa\_hyd moe2D\_vsa\_other moe2D\_vsa\_pol  
## 1 0 0 158.1 0 0.00  
## 2 0 0 147.6 0 13.57  
## 3 0 0 118.2 0 0.00  
## moe2D\_weinerPath moe2D\_weinerPol moe2D\_zagreb Activity  
## 1 120 11 46 5.29  
## 2 58 9 38 3.12  
## 3 16 0 20 3.40

We will build a model on these data to predict the activity. Some notes: - A common aspect to chemical descriptors is that they are *highly correlated*. Many descriptors often measure some variation of the same thing. For example, in these data, there are 56 potential predictors that measure different flavors of surface area. It might be a good idea to *reduce the dimensionality* of these data by pre-filtering the predictors and/or using a dimension reduction technique. - Other descriptors are counts of certain types of aspects of the molecule. For example, one predictor is the number of Bromine atoms. The vast majority of compounds lack Bromine and this leads to a near-zero variance situation discussed previously. It might be a good idea to pre-filter these.

Also , to demonstrate the utility of recipes, suppose that we could score potential drugs on the basis of how manufacturable they might be. We might eant to build a model on the entire data set but only evaluate it on compounds that could be reasonably manufactured. For illustration, we’lll assume that, as a compounds molecule weight increases, its manufacturability *dercreases*. For this purpose, we create a new variable (manufacturability) that is neither an outcome or predictor but will be needed to compute performance.

tox <- tox %>%  
 select(-Molecule) %>%  
 ## Suppose the easy of manufacturability is   
 ## related to the molecular weight of the compound  
 mutate(manufacturability = 1/moe2D\_Weight) %>%  
 mutate(manufacturability = manufacturability/sum(manufacturability))  
  
head(tox,3)

## moeGao\_Abra\_L moeGao\_Abra\_R moeGao\_Abra\_acidity moeGao\_Abra\_basicity  
## 1 4.729 0.512 0.030 0.126  
## 2 4.226 0.527 0.375 0.452  
## 3 2.790 0.383 0.030 0.101  
## moeGao\_Abra\_pi moe2D\_BCUT\_PEOE\_0 moe2D\_BCUT\_PEOE\_1 moe2D\_BCUT\_PEOE\_2  
## 1 0.330 -2.696 -0.009128 0.04606  
## 2 0.712 -2.556 -0.080280 0.05082  
## 3 0.469 -2.262 -0.083670 0.02702  
## moe2D\_BCUT\_PEOE\_3 moe2D\_BCUT\_SLOGP\_0 moe2D\_BCUT\_SLOGP\_1  
## 1 2.639 -2.554 0.12300  
## 2 2.615 -2.557 0.06489  
## 3 2.350 -2.066 0.12300  
## moe2D\_BCUT\_SLOGP\_2 moe2D\_BCUT\_SLOGP\_3 moe2D\_BCUT\_SMR\_0 moe2D\_BCUT\_SMR\_1  
## 1 0.1363 2.784 -2.427 0.1057  
## 2 0.6895 2.681 -2.334 0.1057  
## 3 0.6895 2.596 -2.027 0.1057  
## moe2D\_BCUT\_SMR\_2 moe2D\_BCUT\_SMR\_3 moe2D\_GCUT\_PEOE\_0 moe2D\_GCUT\_PEOE\_1  
## 1 0.2077 2.912 -0.7881 -0.2273  
## 2 0.5853 2.840 -0.7825 -0.2403  
## 3 0.5853 2.611 -0.6887 -0.2437  
## moe2D\_GCUT\_PEOE\_2 moe2D\_GCUT\_PEOE\_3 moe2D\_GCUT\_SLOGP\_0  
## 1 -0.1139 2.451 -0.6697  
## 2 -0.1342 2.056 -0.8724  
## 3 -0.1330 1.559 -0.5478  
## moe2D\_GCUT\_SLOGP\_1 moe2D\_GCUT\_SLOGP\_2 moe2D\_GCUT\_SLOGP\_3  
## 1 -0.08693 -0.0370 2.598  
## 2 -0.08735 0.3012 2.188  
## 3 -0.03700 0.5295 1.871  
## moe2D\_GCUT\_SMR\_0 moe2D\_GCUT\_SMR\_1 moe2D\_GCUT\_SMR\_2 moe2D\_GCUT\_SMR\_3  
## 1 -0.5188 -0.0543 -0.009485 2.704  
## 2 -0.5229 -0.0543 0.425300 2.297  
## 3 -0.4864 -0.0543 0.425300 1.858  
## moe2D\_Kier1 moe2D\_Kier2 moe2D\_Kier3 moe2D\_KierA1 moe2D\_KierA2  
## 1 8.1 3.408 2.286 6.817 2.807  
## 2 8.0 1.750 2.222 13.390 3.154  
## 3 5.0 1.000 0.000 11.470 2.844  
## moe2D\_KierA3 moe2D\_KierFlex moe2D\_PEOE\_PC. moe2D\_PEOE\_PC..1  
## 1 1.850 1.914 0.5430 -0.5430  
## 2 4.267 5.278 0.6904 -0.6904  
## 3 0.000 6.523 0.2690 -0.2690  
## moe2D\_PEOE\_RPC. moe2D\_PEOE\_RPC..1 moe2D\_PEOE\_VSA.0 moe2D\_PEOE\_VSA.1  
## 1 0.1053 0.1831 55.45 12.8  
## 2 0.3160 0.5575 25.59 0.0  
## 3 0.6986 0.3111 12.80 0.0  
## moe2D\_PEOE\_VSA.2 moe2D\_PEOE\_VSA.3 moe2D\_PEOE\_VSA.4 moe2D\_PEOE\_VSA.5  
## 1 0 0 0.00 0  
## 2 0 0 10.32 0  
## 3 0 0 0.00 0  
## moe2D\_PEOE\_VSA.6 moe2D\_PEOE\_VSA.0.1 moe2D\_PEOE\_VSA.1.1  
## 1 0 63.60 38.97  
## 2 0 37.50 88.75  
## 3 0 18.75 88.75  
## moe2D\_PEOE\_VSA.2.1 moe2D\_PEOE\_VSA.3.1 moe2D\_PEOE\_VSA.4.1  
## 1 0 0 0  
## 2 0 0 0  
## 3 0 0 0  
## moe2D\_PEOE\_VSA.5.1 moe2D\_PEOE\_VSA.6.1 moe2D\_PEOE\_VSA\_FHYD  
## 1 0 0.000 1.0000  
## 2 0 7.768 0.8935  
## 3 0 0.000 1.0000  
## moe2D\_PEOE\_VSA\_FNEG moe2D\_PEOE\_VSA\_FPNEG moe2D\_PEOE\_VSA\_FPOL  
## 1 0.6005 0.00000 0.0000  
## 2 0.7886 0.04571 0.1065  
## 3 0.8936 0.00000 0.0000  
## moe2D\_PEOE\_VSA\_FPOS moe2D\_PEOE\_VSA\_FPPOS moe2D\_PEOE\_VSA\_HYD  
## 1 0.3995 0.00000 170.8  
## 2 0.2114 0.06076 151.8  
## 3 0.1064 0.00000 120.3  
## moe2D\_PEOE\_VSA\_NEG moe2D\_PEOE\_VSA\_PNEG moe2D\_PEOE\_VSA\_POL  
## 1 102.6 0.000 0.00  
## 2 134.0 7.768 18.09  
## 3 107.5 0.000 0.00  
## moe2D\_PEOE\_VSA\_POS moe2D\_PEOE\_VSA\_PPOS moe2D\_Q\_PC. moe2D\_Q\_PC..1  
## 1 68.25 0.00 1.141 -1.141  
## 2 35.92 10.32 1.550 -1.550  
## 3 12.80 0.00 0.870 -0.870  
## moe2D\_Q\_RPC. moe2D\_Q\_RPC..1 moe2D\_Q\_VSA\_FHYD moe2D\_Q\_VSA\_FNEG  
## 1 0.1315 0.2629 0.7718 0.2282  
## 2 0.5613 0.4387 0.3713 0.5680  
## 3 1.0000 0.3333 0.2622 0.7378  
## moe2D\_Q\_VSA\_FPNEG moe2D\_Q\_VSA\_FPOL moe2D\_Q\_VSA\_FPOS moe2D\_Q\_VSA\_FPPOS  
## 1 0.2282 0.2282 0.7718 0.00000  
## 2 0.5680 0.6287 0.4320 0.06076  
## 3 0.7378 0.7378 0.2622 0.00000  
## moe2D\_Q\_VSA\_HYD moe2D\_Q\_VSA\_NEG moe2D\_Q\_VSA\_PNEG moe2D\_Q\_VSA\_POL  
## 1 131.80 38.97 38.97 38.97  
## 2 63.09 96.51 96.51 106.80  
## 3 31.54 88.75 88.75 88.75  
## moe2D\_Q\_VSA\_POS moe2D\_Q\_VSA\_PPOS moe2D\_SMR moe2D\_SMR\_VSA0 moe2D\_SMR\_VSA1  
## 1 131.80 0.00 4.591 0 0.00  
## 2 73.41 10.32 3.655 0 25.39  
## 3 31.54 0.00 2.593 0 0.00  
## moe2D\_SMR\_VSA2 moe2D\_SMR\_VSA3 moe2D\_SMR\_VSA4 moe2D\_SMR\_VSA5  
## 1 4.411 6.371 0 74.25  
## 2 0.000 0.000 0 0.00  
## 3 0.000 0.000 0 0.00  
## moe2D\_SMR\_VSA6 moe2D\_SMR\_VSA7 moe2D\_SlogP moe2D\_SlogP\_VSA0  
## 1 0 98.75 3.309 0.00  
## 2 0 184.10 2.547 25.39  
## 3 0 150.80 2.796 0.00  
## moe2D\_SlogP\_VSA1 moe2D\_SlogP\_VSA2 moe2D\_SlogP\_VSA3 moe2D\_SlogP\_VSA4  
## 1 0 0 0 4.411  
## 2 0 0 0 0.000  
## 3 0 0 0 0.000  
## moe2D\_SlogP\_VSA5 moe2D\_SlogP\_VSA6 moe2D\_SlogP\_VSA7 moe2D\_SlogP\_VSA8  
## 1 6.371 0 17.64 56.61  
## 2 0.000 0 0.00 0.00  
## 3 0.000 0 0.00 0.00  
## moe2D\_SlogP\_VSA9 moe2D\_TPSA moe2D\_VAdjEq moe2D\_VAdjMa moe2D\_VDistEq  
## 1 98.75 0.00 0.7219 4.322 2.344  
## 2 184.10 20.23 0.7579 3.807 1.550  
## 3 150.80 0.00 0.9044 3.000 0.971  
## moe2D\_VDistMa moe2D\_Weight moe2D\_a\_IC moe2D\_a\_ICM moe2D\_a\_acc  
## 1 6.306 136.2 24.99 0.9612 0  
## 2 5.706 177.5 26.20 1.7460 1  
## 3 4.250 133.4 12.49 1.5610 0  
## moe2D\_a\_aro moe2D\_a\_count moe2D\_a\_don moe2D\_a\_heavy moe2D\_a\_hyd  
## 1 0 26 0 10 10  
## 2 0 15 1 8 6  
## 3 0 8 0 5 4  
## moe2D\_a\_nBr moe2D\_a\_nC moe2D\_a\_nCl moe2D\_a\_nF moe2D\_a\_nH moe2D\_a\_nN  
## 1 0 10 0 0 16 0  
## 2 0 4 3 0 7 0  
## 3 0 2 3 0 3 0  
## moe2D\_a\_nO moe2D\_apol moe2D\_b\_1rotN moe2D\_b\_1rotR moe2D\_b\_ar  
## 1 0 28.27 1 0.1 0  
## 2 1 19.05 0 0.0 0  
## 3 0 12.06 0 0.0 0  
## moe2D\_b\_count moe2D\_b\_double moe2D\_b\_heavy moe2D\_b\_rotN moe2D\_b\_rotR  
## 1 26 2 10 1 0.1000  
## 2 14 0 7 1 0.1429  
## 3 7 0 4 0 0.0000  
## moe2D\_b\_single moe2D\_b\_triple moe2D\_balabanJ moe2D\_bpol moe2D\_chi0  
## 1 24 0 2.281 17.490 7.56  
## 2 14 0 4.020 8.912 7.00  
## 3 7 0 3.024 4.540 4.50  
## moe2D\_chi0\_C moe2D\_chi0v moe2D\_chi0v\_C moe2D\_chi1 moe2D\_chi1\_C  
## 1 7.56 6.983 6.983 4.698 4.698  
## 2 3.00 6.849 3.000 3.250 1.250  
## 3 1.50 4.902 1.500 2.000 0.500  
## moe2D\_chi1v moe2D\_chi1v\_C moeGao\_chi2\_C moeGao\_chi2v\_C moeGao\_chi3c  
## 1 4.009 4.009 4.264 3.368 0.7887  
## 2 3.174 1.250 1.000 1.000 2.5000  
## 3 2.201 0.500 0.000 0.000 2.0000  
## moeGao\_chi3c\_C moeGao\_chi3cv moeGao\_chi3cv\_C moeGao\_chi3p moeGao\_chi3p\_C  
## 1 0.7887 0.5526 0.5526 3.003 3.003  
## 2 0.2500 2.3900 0.2500 2.250 0.000  
## 3 0.0000 2.6580 0.0000 0.000 0.000  
## moeGao\_chi3pv moeGao\_chi3pv\_C moeGao\_chi4c moeGao\_chi4c\_C moeGao\_chi4ca  
## 1 2.335 2.335 0.0 0 1.953  
## 2 2.081 0.000 0.5 0 0.000  
## 3 0.000 0.000 0.5 0 0.000  
## moeGao\_chi4ca\_C moeGao\_chi4cav moeGao\_chi4cav\_C moeGao\_chi4cv  
## 1 1.953 1.413 1.413 0.0000  
## 2 0.000 0.000 0.000 0.4763  
## 3 0.000 0.000 0.000 0.7289  
## moeGao\_chi4cv\_C moeGao\_chi4pc moeGao\_chi4pc\_C moeGao\_chi4pcv  
## 1 0 1.449 1.449 1.009  
## 2 0 4.500 0.000 3.971  
## 3 0 0.000 0.000 0.000  
## moeGao\_chi4pcv\_C moe2D\_chiral moe2D\_chiral\_u moe2D\_density  
## 1 1.009 1 1 0.586  
## 2 0.000 0 0 1.056  
## 3 0.000 0 0 1.206  
## moe2D\_diameter moe2D\_kS\_aaCH moe2D\_kS\_aaN moe2D\_kS\_aaaC moe2D\_kS\_aasC  
## 1 6 0 0 0 0  
## 2 3 0 0 0 0  
## 3 2 0 0 0 0  
## moe2D\_kS\_dCH2 moe2D\_kS\_dO moe2D\_kS\_dsCH moe2D\_kS\_dsN moe2D\_kS\_dssC  
## 1 3.975 0 2.35 0 2.904  
## 2 0.000 0 0.00 0 0.000  
## 3 0.000 0 0.00 0 0.000  
## moe2D\_kS\_sBr moe2D\_kS\_sCH3 moe2D\_kS\_sCl moe2D\_kS\_sF moe2D\_kS\_sNH2  
## 1 0 4.355 0.00 0 0  
## 2 0 2.861 15.87 0 0  
## 3 0 1.484 15.18 0 0  
## moe2D\_kS\_sOH moe2D\_kS\_ssCH2 moe2D\_kS\_ssNH moe2D\_kS\_ssO moe2D\_kS\_sssCH  
## 1 0.000 3.816 0 0 0.7673  
## 2 8.958 0.000 0 0 0.0000  
## 3 0.000 0.000 0 0 0.0000  
## moe2D\_kS\_sssN moe2D\_kS\_ssssC moe2D\_kS\_tCH moe2D\_kS\_tNH moe2D\_kS\_tsC  
## 1 0 0.000 0 0 0  
## 2 0 -2.856 0 0 0  
## 3 0 -1.083 0 0 0  
## moe2D\_lip\_acc moe2D\_lip\_don moe2D\_lip\_violation moe2D\_logP.o.w.  
## 1 0 0 0 3.604  
## 2 1 1 0 1.973  
## 3 0 0 0 2.139  
## moe2D\_logS moe2D\_mr moe2D\_opr\_brigid moe2D\_opr\_leadlike moe2D\_opr\_nring  
## 1 -3.005 4.500 7 1 1  
## 2 -2.450 3.554 0 1 0  
## 3 -2.326 2.471 0 1 0  
## moe2D\_opr\_nrot moe2D\_opr\_violation moe2D\_petitjean moe2D\_petitjeanSC  
## 1 2 0 0.5000 1.0  
## 2 0 0 0.3333 0.5  
## 3 0 0 0.5000 1.0  
## moe2D\_radius moe2D\_reactive moe2D\_rings moe2D\_vdw\_area moe2D\_vdw\_vol  
## 1 3 0 1 170.8 232.5  
## 2 2 0 0 169.9 168.0  
## 3 1 0 0 120.3 110.6  
## moe2D\_vsa\_acc moe2D\_vsa\_don moe2D\_vsa\_hyd moe2D\_vsa\_other moe2D\_vsa\_pol  
## 1 0 0 158.1 0 0.00  
## 2 0 0 147.6 0 13.57  
## 3 0 0 118.2 0 0.00  
## moe2D\_weinerPath moe2D\_weinerPol moe2D\_zagreb Activity manufacturability  
## 1 120 11 46 5.29 0.002713747  
## 2 58 9 38 3.12 0.002082323  
## 3 16 0 20 3.40 0.002770707

For this analysis, we will compute the RMSE using weights based on the manufacturability columns such that a difficult compound has less impact on the RMSE.

model\_stats <- function(data,lev=NULL, model=NULL){  
 stats <- defaultSummary(data,lev=lev,model=model)  
   
 wt\_rmse <- function(pred,obs,wts,na.rm=T)  
 sqrt(weighted.mean(pred-obs)^2, wts, na.rm=na.rm)  
   
 res <- wt\_rmse(pred=data$pred,  
 obs=data$obs,  
 wts=data$manufacturability)  
 c(wRMSE=res,stats)  
}

There is no way to include this extra variable using the default train method or using train.formula.

Now let’s create a recipe incrementally. First, we will use the formula methods to declare the outcome and predictors but change the analysis role of the manufacturability variabile so that it will only be available when summarizing the model fit.

tox\_recipe <- recipe(Activity~., data=tox) %>%   
 add\_role(manufacturability, new\_role="performance var")

## Warning: Changing role(s) for manufacturability

tox\_recipe

## Data Recipe  
##   
## Inputs:  
##   
## role #variables  
## outcome 1  
## performance var 1  
## predictor 220

Using this new role, the manufacturability column will be available when the summary function is executed and the appropriate rows of the data set will be exposed during resampling. For example, if one were debug the model\_stats function during of a model, the data object might look like this:

Browse[1]> head(data)

More than one variable can have this role so that multiple columns can be made available. Now let’s add some steps to the recipe. First, we remove sparse and unbalanced predictors:

tox\_recipe <- tox\_recipe %>%   
 step\_nzv(all\_predictors())  
tox\_recipe

## Data Recipe  
##   
## Inputs:  
##   
## role #variables  
## outcome 1  
## performance var 1  
## predictor 220  
##   
## Operations:  
##   
## Sparse, unbalanced variable filter on all\_predictors()

Note that we have only specified what will *happen once the recipe* is executed. This is only a specification that uses a generic declaration of all\_predictors.

As mentioned above, there are a lot of different surface area predictors and they tend to have very high correlations with one another. We’ll add one or more predictors to the model in place of these predictors using principal component analysis. The step will retain the number of components required to capture 95% of the information contained in these 56 predictors. We’ll name these new predictors surf\_area\_1, surf\_area\_2 etc.

tox\_recipe <- tox\_recipe %>%   
 step\_pca(contains("VSA"),prefix="surf\_area\_",threshold=.95)

Now, lets specific that the third step in the recipe is to reduce the number of predictors so that no pair has an absolute correlation greater than 0.90. However, we might want to keep the surface are principal components so we **exclude** thse from the filter (using the minus sign)

tox\_recipe <- tox\_recipe %>%   
 step\_corr(all\_predictors(),-starts\_with("surf\_area\_"),threshold=.90)

Finally, we can center and scale all of the predictors that are available at the end of the recipe:

tox\_recipe <- tox\_recipe %>%   
 step\_center(all\_predictors()) %>%   
 step\_scale(all\_predictors())  
tox\_recipe

## Data Recipe  
##   
## Inputs:  
##   
## role #variables  
## outcome 1  
## performance var 1  
## predictor 220  
##   
## Operations:  
##   
## Sparse, unbalanced variable filter on all\_predictors()  
## PCA extraction with contains("VSA")  
## Correlation filter on 2 items  
## Centering for all\_predictors()  
## Scaling for all\_predictors()

Let’s use this recipe to fit a SVM model and pick the tuning parameters that minimize the weighted RMSE value:

tox\_ctrl <- trainControl(method = "cv", summaryFunction = model\_stats)  
set.seed(888)  
tox\_svm <- train(tox\_recipe, tox,  
 method = "svmRadial",   
 metric = "wRMSE",  
 maximize = FALSE,  
 tuneLength = 10,  
 trControl = tox\_ctrl)  
tox\_svm

What variables were generated by the recipe?

## originally:  
ncol(tox)-2

## [1] 220

## after the recipe was executed:  
predictors(tox\_svm)

The trained recipe is available in the train object and now shows specific variables involved in each step:

tox\_svm$recipe

## Case weights

For models that accept them, case weights can be passed to the model fitting routines using a role of "case weight".

# Using your own model in train

The package contains a large number of predictive model interfaces. However, you may want to create your own because:

* you are testing out a novel model or the package doesn’t have a model that you are interested in
* you would like to run an existing model in the package your own way
* there are **pre-processing** or **sampling steps** not contained in the package or you just don’t like the way the package does things

You can still get the benefits of the caret infrastructure by creating your own model.

Currently, hen you specify the type of model that you are interested in (e.g. type = "lda"), the train function runs another function called getModelInfo to retrieve the specifics of that model from the existing catalog. For example:

ldaModelInfo <- getModelInfo(model="lda",regex=FALSE)[[1]]  
## Model components  
names(ldaModelInfo)

## [1] "label" "library" "loop" "type" "parameters"  
## [6] "grid" "fit" "predict" "prob" "predictors"  
## [11] "tags" "levels" "sort"

To use your own model, you can pass a list of these components to type. This page will describe those components in detail.

## Illustrative example 1: SVMs with Laplacian Kernels

The package currently contains support vector machine (SVM) models using linear, polynomial and radial basis function kernels. The kernlab package has other functions, including the Laplacian Kernel. We will illustrate the model components for this model, which has two parameters: the standard cost parameter for SVMs and one kernel parameter (sigma)

## Model Components

You can pass a list of information to the method argument in train. For models that are built-in to the package, you can just pass the method name as before.

There are some basic components of the list for custom models. A brief description is below for each then, after setting up and example, each will be described in detail. The list should have the following elements:

* library is a character vector of package names that will be needed to fit the model or calculate predictions. NULL can also be used.
* type is a simple character vector with values "Classification", "Regression" or both.
* parameters is a data frame with three simple attributes for each tuning parameter (if any): the argument name (e.g. mtry), the type of data in the parameter grid and textual labels for the parameter.
* grid is a function that is used to create the tuning grid (unless the user gives the exact values of the parameters via tuneGrid) fit is a function that fits the model
* predict is the function that creates predictions
* prob is a function that can be used to create class probabilities (if applicable) sort is a function that sorts the parameter from most complex to least
* loop is an **optional** function for advanced users for models that can create multiple submodel predictions from the same object.
* levels is an **optional** function, primarily for classification models using S4 methods to return the factor levels of the outcome.
* tags is an **optional** character vector that has subjects associated with the model, such as Tree-Based Model or Embedded Feature Selection. This string is used by the package to create additional documentation pages on the package website. label is an optional character string that names the model (e.g. “Linear Discriminant Analysis”).
* predictors is an **optional** function that returns a character vector that contains the names of the predictors that we used in the prediction equation.
* varImp is an **optional** function that calculates variable importance metrics for the model (if any).
* oob is another **optional** function that calculates out-of-bag performance estimates from the model object. Most models do not have this capability but some (e.g. random forests, bagged models) do.
* notes is an **optional** character vector that can be used to document non-obvious aspects of the model. For example, there are two Bayesian lasso models (blasso and blassoAveraged) and this field is used to describe the differences between the two models.
* check is an **optional** function that can be used to check the system/install to make sure that any atypical software requirements are available to the user. The input is pkg, which is the same character string given by the library. This function is run after the checking function to see if the packages specified in library are installed. As an example, the model pythonKnnReg uses certain python libraries and the user should have python and these libraries installed. The model file demonstrates how to check for python libraries prior to running the R model.

In the caret package, the subdirectory models has all the code for each model that train interfaces with and these can be used as prototypes for your model.

Let’s create a new model for a classification support vector machine using the Laplacian kernel function. We will use the kernlab package’s ksvm function. The kernel has two parameters: the standard cost parameter for SVMs and one kernel parameter (sigma).

To start, we’ll create a new list:

lpSVM <- list(type="Classification",  
 library="kernlab",  
 loop=NULL)

This model can also be used to regression, but we will constratin things here for simplificity. For other SVM models, the type value would be c("Classification", "Regression").

The library value checks to see if this package is installed and loads it whenever it is needed (e.g., before modeling or prediction). **Note:** caret will check to see if these packages are installed but will *not* explicitly load them. As such unctions that are used from the package should be referenced by namespace. This is discussed more below when describing the fit function.

### The parameters element

We have to create some basic information for the parameters in the form of a data frame. The first column is the name of the parameter. The convention is to use the argument name in the model function (e.g. the ksvm function here). Those values are C and sigma. Each is a number and we can give them labels of "Cost" and "Sigma", respectively. The parameters element would then be:

prm <- data.frame(parameter=c("C","Sigma"),  
 class=rep("numeric",2),  
 label=c("Cost","Sigma"))

Now, we assign it to the model list:

lpSVM$parameters <- prm  
summary(lpSVM)

## Length Class Mode   
## type 1 -none- character  
## library 1 -none- character  
## loop 0 -none- NULL   
## parameters 3 data.frame list

Values of type can indicate numeric, character or logical data types.

### The grid Element

This should be a function that takes parameters: x and y (for the predictors and outcome data), len (the number of values per tuning parameter) as well as search. len is the value of tuneLength that is potentially passed in through train. search can be either "grid" or "random". This can be used to setup a grid for searching or random values for random search.

The output should be a data frame of tuning parameter combinations with a column for each parameter. The column names should be the parameter name (e.g. the values of prm$parameter). In our case, let’s vary the cost parameter on the log 2 scale. For the sigma parameter, we can use the kernlab function sigest to pre-estimate the value. Following ksvm we take the average of the low and high estimates. Here is a function we could use:

svmGrid <- function(x,y,len=NULL, search="grid"){  
 library(kernlab)  
 ## This produces low, middle and high value for sigma  
 ## i.e. a vector with 3 elements  
 sigmas <- kernlabb::sigest(as.matrix(x),na.action=na.omit, scaled=TRUE)  
 ## To use a grid search  
 if (search =="grid"){  
 out <- expand.grid(sigma=mean(as.vector(sigmas[-2])),  
 C=2^(1:len)-3)  
 }else{  
 ## For random search, define ranges for the parameters then  
 ## generate random values for them  
 rng <- extendrange(log(sigmas),f=.75)  
 out <- data.frame(sigma=exp(runif(len,min=rng[1],max=rng[2])),  
 C=2^runif(len,min=-5,max=8))  
 }  
 out  
}

Why did we use kernelab::sigest instead of sigest? As previously mentioned, caret will not execute library(kernlab) unless you explicitly code it in these functions. Since it not explicityl loaded, you have to call it *using the namespace operator* ::.

Again, the user can pass their own grid via train’s tuneGrid option or they can use this code to create a default grid. We assign this function to the overall model list:

lpSVM$grid <- svmGrid

### The fit Element

Here is where we fit the model. This fit function has several arguments: - x, y: the current data used to fit the model - wts: optional instance weights (not applicable for this particular model) - param: the current tuning parameter values - lev: the class levels of the outcome (or NULL in regression) - last: a logical for whether the current fit is the final fit - weights - classProbs: a logical for whether class probabilities should be computed.

Here is something we could use for this model:

svmFit <- function(x, y, wts, param, lev, last, weights, classProbs, ...) {   
 kernlab::ksvm(  
 x = as.matrix(x), y = y,  
 kernel = "rbfdot",  
 kpar = list(sigma = param$sigma),  
 C = param$C,  
 prob.model = classProbs,  
 ...  
 )  
 }  
   
lpSVM$fit <- svmFit

A few note about this: - Notice that the package is not loaded in the code. It is loaded prior to this function being called so it won’t hurt if you load it again (but that’s not needed). - The ksvm function requires a matrix or predictors. If the original data were a data frame, this would throw and error. - The tuning parameters are references in the param data frame. There is always a single row in this data frame. - The probability model is fit based on the value of classProbs. This value is determined by the value given in trainControl. - The three dots allow the user to pass options in from train to, in this case, the ksvm function. For example, if the use wanted to set the cache size for the function, they could list cache = 80 and this argument will be pass from train to ksvm. - Any pre-processing that was requested in the call to train have been done. For example, if preProc = "center" was originally requested, the columns of x seen within this function are mean centered. - Again, the namespace operator :: is used for rbfdot and ksvm to ensure that the function can be found.

### 13.3.4 The predict Element

This is a function that produces a vector or predictions. In our case these are class predictions but they could be numbers for regression models.

The arguments are:

* modelFit: the model produced by the fit code shown above.
* newdata: the predictor values of the instances being predicted (e.g. out-of-bag samples)
* preProc
* submodels: this an optional list of tuning parameters only used with the loop element discussed below. In most cases, it will be NULL.

Our function will be very simple:

svmPred <- function(modelFit, newdata, preProc = NULL, submodels = NULL)  
 kernlab::predict(modelFit, newdata)  
lpSVM$predict <- svmPred

The function predict.ksvm will automatically create a factor vector as output. The function could also produce character values. Either way, the innards of train will make them factors and ensure that the same levels as the original data are used.

### 13.3.5 The prob element

If a regression model is being used or if the classification model does not create class probabilities a value of NULL can be used here instead of a function. Otherwise, the function arguments are the same as the pred function. The output should be a matrix or data frame of class probabilities with a column for each class. The column names should be the class levels.

We can use:

svmProb <- function(modelFit, newdata, preProc = NULL, submodels = NULL)  
 kernlab::predict(modelFit, newdata, type = "probabilities")  
lpSVM$prob <- svmProb

If you look at some of the SVM examples in the models directory, the real functions used by train are much more complicated so that they can deal with model failures, probabilities that do not sum to 1 etc.

## The sort element

This is an optional function that sorts the tuning parameters from the simplest model to the most complex. There are times where this ordering is not obvious. This information is used when the performance values are tied across multiple parameters. We would probably want to choose the least complex model in those cases.

Here, we will sort by the cost value. Smaller values of C produce smoother class boundaries than larger values:

svmSort <- function(x) x[order(x$C),]  
lpSVM$sort <- svmSort

### The levels element

train ensures that classification models always predict factors with the same levels. To do this at prediction time, the package needs to know the levels from the model object (specifically, the finalModels slot of the train object).

For model functions using S3 methods, train automatically attaches a character vector called obsLevels to the object and the package code uses this value. However, this strategy does not work for S4 methods. In these cases, the package will use the code found in the levels slot of the model list.

For example, the ksvm function uses S4 methods but, unlike most model functions, has a built–in function called lev that will extract the class `levels (if any). In this case, our levels code would be:

lpSVM$levels <- function(x) kernlab::lev(x)

In most other cases, the levels will beed to be extracted from data contained in the fitted model object. As another example, objects created using the ctree function in the party package would need to use:

function(x) levels(x@data@get("response")[,1])

## function(x) levels(x@data@get("response")[,1])

Again, this is only used for classification models using S4 methods. We should now be ready to fit our model.

library(mlbench)  
data(Sonar)  
   
library(caret)  
set.seed(998)  
inTraining <- createDataPartition(Sonar$Class, p = .75, list = FALSE)  
training <- Sonar[ inTraining,]  
testing <- Sonar[-inTraining,]  
  
fitControl <- trainControl(method = "repeatedcv",  
 ## 10-fold CV...  
 number = 10,  
 ## repeated ten times  
 repeats = 10)  
   
set.seed(825)  
Laplacian <- train(Class ~ ., data = training,   
 method = lpSVM,   
 preProc = c("center", "scale"),  
 tuneLength = 8,  
 trControl = fitControl)  
Laplacian

A plot of the data shows that the model doesn’t change when the cost value is above 16.

ggplot(Lapacian)+scale\_x\_log10()

## Illustrative Example 2: Something more complicated - LogitBoost

### The loop element

This function can be used to create custom loops for models to tune over. In most cases, the function can just retrun the existing tuning grid.

For example, a LogitBoost model can be trained over the number of boosting iterationds. IN the caTools package, the LogitBoost function can be used to fit this model. For example:

library(caTools)  
  
mod <- LogitBoost(as.matrix(x), y, nIter = 51)

If we were to tune the model evaluating models where the number of iterations was 11, 21, 31, 41 and 51, the grid could be

lbGrid <- data.frame(nIter = seq(11, 51, by = 10))   
lbGrid

## nIter  
## 1 11  
## 2 21  
## 3 31  
## 4 41  
## 5 51

During resampling, train could loop over all five rows in lbGrid and fit five models. However, the predict.LogitBoost function has an argument called nIter that can produce, in this case, predictions from mod for all five models.

Instead of train fitting five models, we could fit a single model with nIter=class=“hl num”>51 and derive preidctions for all five models using only mod.

The terminology used here is that nIter is a sequential tuning parameter (and the other parameters would be considered *fixed*).

The loop argument for model is used to produce two objects - loop: this is the actual loop that is used by train. - submodels is a *list* that has as many elements as there are rows in loop. The list has all the extra parameter settings that can be derived for each model.

Going back to the LogitBoost example, we could have:

loop <- data.frame(.nIter=51)  
loop

## .nIter  
## 1 51

submodels <- list(data.frame(nIter=seq(11,41,by=10)))  
submodels

## [[1]]  
## nIter  
## 1 11  
## 2 21  
## 3 31  
## 4 41

For this case, train first fits the nIter = 51 model. When the model is predicted, that code has a for loop that iterates over the elements of submodel[[1]] to get the predictions for the other 4 models.

In the end, predictions for all five models (for nIter = seq(11, 51, by = 10)) with a single model fit.

There are other models built-in to caret that are used this way. There are a number of models that have multiple sequential tuning parameters.

If the loop argument is left NULL the results of tuneGrid are used as the simple loop and is recommended for most situations. Note that the machinery that is used to “derive” the extra predictions is up to the user to create, typically in the predict and prob elements of the custom model object.

For the LogitBoost model, some simple code to create these objects would be:

fullGrid <- data.frame(nIter=seq(11,51,by=10))  
  
## Get the largest value of nIter to fit the "full" model  
loop <- fullGrid[which.max(fullGrid$nIter),,drop=FALSE]  
loop

## nIter  
## 5 51

submodels <- fullGrid[-which.max(fullGrid$nIter),,drop=FALSE]  
## This needs to be encased in a list in case there are more  
## than one tuning parameter  
submodels <- list(submodels)   
submodels

## [[1]]  
## nIter  
## 1 11  
## 2 21  
## 3 31  
## 4 41

For the LogitBoost custom model object, we could use this code in the predict slot:

lbPred <- function(modelFit, newdata, preProc = NULL, submodels = NULL) {  
 ## This model was fit with the maximum value of nIter  
 out <- caTools::predict.LogitBoost(modelFit, newdata, type="class")  
   
 ## In this case, 'submodels' is a data frame with the other values of  
 ## nIter. We loop over these to get the other predictions.  
 if(!is.null(submodels)) {  
 ## Save \_all\_ the predictions in a list  
 tmp <- out  
 out <- vector(mode = "list", length = nrow(submodels) + 1)  
 out[[1]] <- tmp  
   
 for(j in seq(along = submodels$nIter)) {  
 out[[j+1]] <- caTools::predict.LogitBoost(  
 modelFit,  
 newdata,  
 nIter = submodels$nIter[j])  
   
 }  
 }  
 out   
}

A few more notes:

* The code in the fit element does not have to change.
* The prob slot works in the same way. The only difference is that the values saved in the outgoing lists are matrices or data frames of probabilities for each class.
* After model training (i.e. predicting new samples), the value of submodels is set to NULL and the code produces a single set of predictions.
* If the model had one sequential parameter and one fixed parameter, the loop data frame would have two columns (one for each parameter). If the model is tuned over more than one value of the fixed parameter, the submodels list would have more than one element. If loop had 10 rows, then length(submodels) would be 10 and loop[i,] would be linked to submodels[[i]].
* In this case, the prediction function was called by namespace too (i.e. caTools::predict.LogitBoost). This may not seem necessary but what functions are available can vary depending on what parallel processing technology is being used. For example, the nature of forking used by doMC and doParallel tends to have easier access to functions while PSOCK methods in doParallel do not. It may be easier to take the safe path of using the namespace operator wherever possible to avoid errors that are difficult to track down.

Here is a slimmed down version of the logitBoost code already in the package:

lbFuncs <- list(library = "caTools",  
 loop = function(grid) {   
 loop <- grid[which.max(grid$nIter),,drop = FALSE]  
 submodels <- grid[-which.max(grid$nIter),,drop = FALSE]  
 submodels <- list(submodels)   
 list(loop = loop, submodels = submodels)  
 },  
 type = "Classification",  
 parameters = data.frame(parameter = 'nIter',  
 class = 'numeric',  
 label = '# Boosting Iterations'),  
 grid = function(x, y, len = NULL, search = "grid") {  
 out <- if(search == "grid")   
 data.frame(nIter = 1 + ((1:len)\*10)) else   
 data.frame(nIter = sample(1:500, size = len))  
 out  
 },  
 fit = function(x, y, wts, param, lev, last, weights, classProbs, ...) {  
 caTools::LogitBoost(as.matrix(x), y, nIter = param$nIter)  
 },  
 predict = function(modelFit, newdata, preProc = NULL, submodels = NULL) {  
 out <- caTools::predict.LogitBoost(modelFit, newdata, type="class")  
 if(!is.null(submodels)) {   
 tmp <- out  
 out <- vector(mode = "list", length = nrow(submodels) + 1)  
 out[[1]] <- tmp  
   
 for(j in seq(along = submodels$nIter)) {  
 out[[j+1]] <- caTools::predict.LogitBoost(  
 modelFit,  
 newdata,  
 nIter = submodels$nIter[j]  
 )  
 }  
 }  
 out   
 },  
 prob = NULL,  
 sort = function(x) x)

Should you care about this? Let’s tune the model over the same dataset for the SVM model above and see how long it takes:

set.seed(825)  
lb1 <- system.time(train(Class ~ ., data = training,   
 method = lbFuncs,   
 tuneLength = 3,  
 trControl = fitControl))  
lb1

## Now get rid of the submodel parts  
lbFuncs2 <- lbFuncs  
lbFuncs2$predict <- function(modelFit, newdata, preProc = NULL, submodels = NULL)   
 caTools::predict.LogitBoost(modelFit, newdata, type = "class")  
lbFuncs2$loop <- NULL   
  
set.seed(825)  
lb2 <- system.time(train(Class ~ ., data = training,   
 method = lbFuncs2,   
 tuneLength = 3,  
 trControl = fitControl))  
lb2

On a data set with 157 instances and 60 predictors and a model that is tuned over only 3 parameter values, there is a 1.37-fold speed-up. If the model were more computationally taxing or the data set were larger or the number of tune parameters that were evaluated was larger, the speed-up would increase. Here is a plot of the speed-up for a few more values of tuneLength:

bigGrid <- data.frame(nIter=seq(1,151,by=10))  
results <- bigGrid  
results$SpeedUp <- NA  
  
for(i in 2:nrow(bigGrid)){   
 rm(lb1, lb2)  
 set.seed(825)  
 lb1 <- system.time(train(Class ~ ., data = training,   
 method = lbFuncs,   
 tuneGrid = bigGrid[1:i,,drop = FALSE],  
 trControl = fitControl))  
   
 set.seed(825)  
 lb2 <- system.time(train(Class ~ ., data = training,   
 method = lbFuncs2,   
 tuneGrid = bigGrid[1:i,,drop = FALSE],  
 trControl = fitControl))  
 results$SpeedUp[i] <- lb2[3]/lb1[3]  
 }  
  
ggplot(results, aes(x = nIter, y = SpeedUp)) +   
 geom\_point() + geom\_smooth(method = "lm") +   
 xlab("LogitBoost Iterations") +   
 ylab("Speed-Up")

The speed-ups show a significant decrease in training time using this method.

Note: The previous examples were run using parallel processing. The remainder in this chapter are run sequentially and, for simplicity, the namespace operator is not used in the custom code modules below.

## Illustrative Example3: Nonstandard Formulas

(Note: the previous third illustration (“SMOTE During Resampling”) is no longer needed due to the inclusion of subsampling via train.)

One limitation of train is that it requires the use of basic model formulas. There are several functions that use special formulas or operators on predictors that won’t (and perhaps should not) work in the top level call to train. However, we can still fit these models.

Here is an example using the mboost function in the **mboost** package from the help page.

# install.packages("mboost")  
# install.packages("TH.data")  
  
library(mboost)

## Warning: package 'mboost' was built under R version 3.5.1

## Loading required package: stabs

## Warning: package 'stabs' was built under R version 3.5.1

## This is mboost 2.9-1. See 'package?mboost' and 'news(package = "mboost")'  
## for a complete list of changes.

##   
## Attaching package: 'mboost'

## The following object is masked from 'package:MLmetrics':  
##   
## AUC

## The following object is masked from 'package:magrittr':  
##   
## extract

## The following object is masked from 'package:tidyr':  
##   
## extract

## The following object is masked from 'package:ggplot2':  
##   
## %+%

data("bodyfat", package="TH.data")  
  
head(bodyfat)

## age DEXfat waistcirc hipcirc elbowbreadth kneebreadth anthro3a anthro3b  
## 47 57 41.68 100.0 112.0 7.1 9.4 4.42 4.95  
## 48 65 43.29 99.5 116.5 6.5 8.9 4.63 5.01  
## 49 59 35.41 96.0 108.5 6.2 8.9 4.12 4.74  
## 50 58 22.79 72.0 96.5 6.1 9.2 4.03 4.48  
## 51 60 36.42 89.5 100.5 7.1 10.0 4.24 4.68  
## 52 61 24.13 83.5 97.0 6.5 8.8 3.55 4.06  
## anthro3c anthro4  
## 47 4.50 6.13  
## 48 4.48 6.37  
## 49 4.60 5.82  
## 50 3.91 5.66  
## 51 4.15 5.91  
## 52 3.64 5.14

mod <- mboost(DEXfat ~ btree(age) + bols(waistcirc) + bbs(hipcirc),  
 data = bodyfat)  
mod

##   
## Model-based Boosting  
##   
## Call:  
## mboost(formula = DEXfat ~ btree(age) + bols(waistcirc) + bbs(hipcirc), data = bodyfat)  
##   
##   
## Squared Error (Regression)   
##   
## Loss function: (y - f)^2   
##   
##   
## Number of boosting iterations: mstop = 100   
## Step size: 0.1   
## Offset: 30.78282   
## Number of baselearners: 3

# model based boosting

We can create a custom model that mimics this code so that we can obtain resampling estimates for this specific model:

modelInfo <- list(label = "Model-based Gradient Boosting",  
 library = "mboost",  
 type = "Regression",  
 parameters = data.frame(parameter = "parameter",  
 class = "character",  
 label = "parameter"),  
 grid = function(x, y, len = NULL, search = "grid")   
 data.frame(parameter = "none"),  
 loop = NULL,  
 fit = function(x, y, wts, param, lev, last, classProbs, ...) {   
 ## mboost requires a data frame with predictors and response  
 dat <- if(is.data.frame(x)) x else as.data.frame(x)  
 dat$DEXfat <- y  
 mod <- mboost(  
 DEXfat ~ btree(age) + bols(waistcirc) + bbs(hipcirc),  
 data = dat  
 )  
 },  
 predict = function(modelFit, newdata, submodels = NULL) {  
 if(!is.data.frame(newdata)) newdata <- as.data.frame(newdata)  
 ## By default a matrix is returned; we convert it to a vector  
 predict(modelFit, newdata)[,1]  
 },  
 prob = NULL,  
 predictors = function(x, ...) {  
 unique(as.vector(variable.names(x)))  
 },  
 tags = c("Ensemble Model", "Boosting", "Implicit Feature Selection"),  
 levels = NULL,  
 sort = function(x) x)  
  
## Just use the basic formula method so that these predictors  
## are passed 'as-is' into the model fitting and prediction  
## functions.  
set.seed(307)  
mboost\_resamp <- train(DEXfat ~ age + waistcirc + hipcirc,   
 data = bodyfat,   
 method = modelInfo,  
 trControl = trainControl(method = "repeatedcv",  
 repeats = 5))

## Warning in bsplines(mf[[i]], knots = args$knots[[i]]$knots, boundary.knots  
## = args$knots[[i]]$boundary.knots, : Some 'x' values are beyond  
## 'boundary.knots'; Linear extrapolation used.  
  
## Warning in bsplines(mf[[i]], knots = args$knots[[i]]$knots, boundary.knots  
## = args$knots[[i]]$boundary.knots, : Some 'x' values are beyond  
## 'boundary.knots'; Linear extrapolation used.  
  
## Warning in bsplines(mf[[i]], knots = args$knots[[i]]$knots, boundary.knots  
## = args$knots[[i]]$boundary.knots, : Some 'x' values are beyond  
## 'boundary.knots'; Linear extrapolation used.  
  
## Warning in bsplines(mf[[i]], knots = args$knots[[i]]$knots, boundary.knots  
## = args$knots[[i]]$boundary.knots, : Some 'x' values are beyond  
## 'boundary.knots'; Linear extrapolation used.  
  
## Warning in bsplines(mf[[i]], knots = args$knots[[i]]$knots, boundary.knots  
## = args$knots[[i]]$boundary.knots, : Some 'x' values are beyond  
## 'boundary.knots'; Linear extrapolation used.  
  
## Warning in bsplines(mf[[i]], knots = args$knots[[i]]$knots, boundary.knots  
## = args$knots[[i]]$boundary.knots, : Some 'x' values are beyond  
## 'boundary.knots'; Linear extrapolation used.  
  
## Warning in bsplines(mf[[i]], knots = args$knots[[i]]$knots, boundary.knots  
## = args$knots[[i]]$boundary.knots, : Some 'x' values are beyond  
## 'boundary.knots'; Linear extrapolation used.  
  
## Warning in bsplines(mf[[i]], knots = args$knots[[i]]$knots, boundary.knots  
## = args$knots[[i]]$boundary.knots, : Some 'x' values are beyond  
## 'boundary.knots'; Linear extrapolation used.  
  
## Warning in bsplines(mf[[i]], knots = args$knots[[i]]$knots, boundary.knots  
## = args$knots[[i]]$boundary.knots, : Some 'x' values are beyond  
## 'boundary.knots'; Linear extrapolation used.  
  
## Warning in bsplines(mf[[i]], knots = args$knots[[i]]$knots, boundary.knots  
## = args$knots[[i]]$boundary.knots, : Some 'x' values are beyond  
## 'boundary.knots'; Linear extrapolation used.

mboost\_resamp

## Model-based Gradient Boosting   
##   
## 71 samples  
## 3 predictor  
##   
## No pre-processing  
## Resampling: Cross-Validated (10 fold, repeated 5 times)   
## Summary of sample sizes: 64, 63, 65, 65, 64, 63, ...   
## Resampling results:  
##   
## RMSE Rsquared MAE   
## 4.024352 0.9057944 3.156712

## Illustraive example 4: PLS feature extraction pre-processing

PCA is a common tool for feature extraction prior to modeling but is *unsupervised* **Partial Least Squares (PLS)** is essentially a supervised version of PCA. For some data sets, there may be some benefit to using PLS to generate new features from the original data (the PLS scores) then use those as an input into a different predictive model. PLS requires parameter tuning. In the example below, we use PLS on a data set with highly correlated predictors then use the PLS scores in a random forest model.

The “trick” here is to save the PLS loadings along with the random forest model fit so that the loadings can be used on future samples for prediction. Also, the PLS and random forest models are jointly tuned instead of an initial modeling process that finalizes the PLS model, then builds the random forest model separately. In this was we optimize both at once. Another important point is that the resampling results reflect the variability in the random forest and PLS models. If we did PLS up-front then resampled the random forest model, we would under-estimate the noise in the modeling process.

The tecator spectroscopy data are used:

data(tecator)  
  
set.seed(930)  
colnames(absorp) <- paste("x", 1:ncol(absorp))  
  
## We will model the protein content data  
trainMeats <- createDataPartition(endpoints[,3], p = 3/4)  
absorpTrain <- absorp[trainMeats[[1]], ]  
proteinTrain <- endpoints[trainMeats[[1]], 3]  
absorpTest <- absorp[-trainMeats[[1]], ]  
proteinTest <- endpoints[-trainMeats[[1]], 3]

Here is the model code:

pls\_rf <- list(label = "PLS-RF",  
 library = c("pls", "randomForest"),  
 type = "Regression",  
 ## Tune over both parameters at the same time  
 parameters = data.frame(parameter = c('ncomp', 'mtry'),  
 class = c("numeric", 'numeric'),  
 label = c('#Components',   
 '#Randomly Selected Predictors')),  
 grid = function(x, y, len = NULL, search = "grid") {  
 if(search == "grid") {  
 grid <- expand.grid(ncomp = seq(1, min(ncol(x) - 1, len), by = 1),  
 mtry = 1:len)  
 } else {  
 grid <- expand.grid(ncomp = sample(1:ncol(x), size = len),  
 mtry = sample(1:ncol(x), size = len))  
 }  
 ## We can't have mtry > ncomp  
 grid <- subset(grid, mtry <= ncomp)  
 },  
 loop = NULL,  
 fit = function(x, y, wts, param, lev, last, classProbs, ...) {   
 ## First fit the pls model, generate the training set scores,  
 ## then attach what is needed to the random forest object to   
 ## be used later  
   
 ## plsr only has a formula interface so create one data frame  
 dat <- x  
 dat$y <- y  
 pre <- plsr(y~ ., data = dat, ncomp = param$ncomp)  
 scores <- predict(pre, x, type = "scores")  
 colnames(scores) <- paste("score", 1:param$ncomp, sep = "")  
 mod <- randomForest(scores, y, mtry = param$mtry, ...)  
 mod$projection <- pre$projection  
 mod  
 },  
 predict = function(modelFit, newdata, submodels = NULL) {   
 ## Now apply the same scaling to the new samples  
 scores <- as.matrix(newdata) %\*% modelFit$projection  
 colnames(scores) <- paste("score", 1:ncol(scores), sep = "")  
 scores <- as.data.frame(scores)  
 ## Predict the random forest model  
 predict(modelFit, scores)  
 },  
 prob = NULL,  
 varImp = NULL,  
 predictors = function(x, ...) rownames(x$projection),  
 levels = function(x) x$obsLevels,  
 sort = function(x) x[order(x[,1]),])

We fit the models and look at the resampling results for the joint model:

meatCtrl <- trainControl(method = "repeatedcv", repeats = 5)  
  
## These will take a while for these data  
set.seed(184)  
plsrf <- train(x = as.data.frame(absorpTrain), y = proteinTrain,   
 method = pls\_rf,  
 preProc = c("center", "scale"),  
 tuneLength = 10,  
 ntree = 1000,  
 trControl = meatCtrl)  
ggplot(plsrf, plotType = "level")

## How does random forest do on its own?  
set.seed(184)  
rfOnly <- train(absorpTrain, proteinTrain,   
 method = "rf",  
 tuneLength = 10,  
 ntree = 1000,  
 trControl = meatCtrl)  
getTrainPerf(rfOnly)

## How does random forest do on its own?  
set.seed(184)  
plsOnly <- train(absorpTrain, proteinTrain,   
 method = "pls",  
 tuneLength = 20,  
 preProc = c("center", "scale"),  
 trControl = meatCtrl)  
getTrainPerf(plsOnly)

The test set results indicate that these data like the linear model more than anything:

postResample(predict(plsrf, absorpTest), proteinTest)

postResample(predict(rfOnly, absorpTest), proteinTest)

ostResample(predict(plsOnly, absorpTest), proteinTest)

## 13.8 illustrative example 5: Optimizing probability threshold for class imbalances

This description was originally posted on [this blog](http://appliedpredictivemodeling.com/blog/).

One of the toughest problems in predictive model occurs when the classes have a severe imbalance. In our book, we spend an entire chapter on this subject itself. One consequence of this is that the performance is generally very biased against the class with the smallest frequencies. For example, if the data have a majority of samples belonging to the first class and very few in the second class, most predictive models will maximize accuracy by predicting everything to be the first class. As a result there’s usually great sensitivity but poor specificity. As a demonstration will use a simulation system [described here](http://appliedpredictivemodeling.com/blog/2013/4/11/a-classification-simulation-system). By default it has about a 50-50 class frequency but we can change this by altering the function argument called intercept:

library(caret)  
  
set.seed(442)  
trainingSet <- twoClassSim(n = 500, intercept = -16)  
testingSet <- twoClassSim(n = 500, intercept = -16)  
  
## Class frequencies  
table(trainingSet$Class)

##   
## Class1 Class2   
## 450 50

There is almost a 9:1 imbalance in these data. Let’s use a standard random forest model with these data using the default value of mtry. We’ll also use repeated 10-fold cross validation to get a sense of performance:

set.seed(949)  
mod0 <- train(Class ~ ., data = trainingSet,  
 method = "rf",  
 metric = "ROC",  
 tuneGrid = data.frame(mtry = 3),  
 ntree = 1000,  
 trControl = trainControl(method = "repeatedcv",  
 repeats = 5,  
 classProbs = TRUE,  
 summaryFunction = twoClassSummary))  
getTrainPerf(mod0)

## TrainROC TrainSens TrainSpec method  
## 1 0.9576444 0.9982222 0.332 rf

## Get the ROC curve  
??roc  
library()  
roc0 <- roc(testingSet$Class,   
 predict(mod0, testingSet, type = "prob")[,1],   
 levels = rev(levels(testingSet$Class)))  
roc0

## Now plot  
plot(roc0, print.thres = c(.5), type = "S",  
 print.thres.pattern = "%.3f (Spec = %.2f, Sens = %.2f)",  
 print.thres.cex = .8,   
 legacy.axes = TRUE)

The area under the ROC curve is very high, indicating that the model has very good predictive power for these data. The plot shows the default probability cut off value of 50%. The sensitivity and specificity values associated with this point indicate that performance is not that good when an actual call needs to be made on a sample.

One of the most common ways to deal with this is to determine an alternate probability cut off using the ROC curve. But to do this well, another set of data (not the test set) is needed to set the cut off and the test set is used to validate it. We don’t have a lot of data this is difficult since we will be spending some of our data just to get a single cut off value.

Alternatively the model can be tuned, using resampling, to determine any model tuning parameters as well as an appropriate cut off for the probabilities.

Suppose the model has one tuning parameter and we want to look at four candidate values for tuning. Suppose we also want to tune the probability cut off over 20 different thresholds. Now we have to look at 20×4=80 different models (and that is for each resample). One other feature that has been opened up his ability to use sequential parameters: these are tuning parameters that don’t require a completely new model fit to produce predictions. In this case, we can fit one random forest model and get it’s predicted class probabilities and evaluate the candidate probability cutoffs using these same hold-out samples. Here is what the model code looks like:

# Get the model code for the original random forest method:  
  
thresh\_code <- getModelInfo("rf", regex = FALSE)[[1]]  
thresh\_code$type <- c("Classification")  
## Add the threshold as another tuning parameter  
thresh\_code$parameters <- data.frame(parameter = c("mtry", "threshold"),  
 class = c("numeric", "numeric"),  
 label = c("#Randomly Selected Predictors",  
 "Probability Cutoff"))  
## The default tuning grid code:  
thresh\_code$grid <- function(x, y, len = NULL, search = "grid") {  
 p <- ncol(x)  
 if(search == "grid") {  
 grid <- expand.grid(mtry = floor(sqrt(p)),   
 threshold = seq(.01, .99, length = len))  
 } else {  
 grid <- expand.grid(mtry = sample(1:p, size = len),  
 threshold = runif(1, 0, size = len))  
 }  
 grid  
 }  
  
## Here we fit a single random forest model (with a fixed mtry)  
## and loop over the threshold values to get predictions from the same  
## randomForest model.  
thresh\_code$loop = function(grid) {   
 library(plyr)  
 loop <- ddply(grid, c("mtry"),  
 function(x) c(threshold = max(x$threshold)))  
 submodels <- vector(mode = "list", length = nrow(loop))  
 for(i in seq(along = loop$threshold)) {  
 index <- which(grid$mtry == loop$mtry[i])  
 cuts <- grid[index, "threshold"]   
 submodels[[i]] <- data.frame(threshold = cuts[cuts != loop$threshold[i]])  
 }   
 list(loop = loop, submodels = submodels)  
 }  
  
## Fit the model independent of the threshold parameter  
thresh\_code$fit = function(x, y, wts, param, lev, last, classProbs, ...) {   
 if(length(levels(y)) != 2)  
 stop("This works only for 2-class problems")  
 randomForest(x, y, mtry = param$mtry, ...)  
 }  
  
## Now get a probability prediction and use different thresholds to  
## get the predicted class  
thresh\_code$predict = function(modelFit, newdata, submodels = NULL) {  
 class1Prob <- predict(modelFit,   
 newdata,   
 type = "prob")[, modelFit$obsLevels[1]]  
 ## Raise the threshold for class #1 and a higher level of  
 ## evidence is needed to call it class 1 so it should   
 ## decrease sensitivity and increase specificity  
 out <- ifelse(class1Prob >= modelFit$tuneValue$threshold,  
 modelFit$obsLevels[1],   
 modelFit$obsLevels[2])  
 if(!is.null(submodels)) {  
 tmp2 <- out  
 out <- vector(mode = "list", length = length(submodels$threshold))  
 out[[1]] <- tmp2  
 for(i in seq(along = submodels$threshold)) {  
 out[[i+1]] <- ifelse(class1Prob >= submodels$threshold[[i]],  
 modelFit$obsLevels[1],   
 modelFit$obsLevels[2])  
 }  
 }   
 out   
 }  
  
## The probabilities are always the same but we have to create  
## mulitple versions of the probs to evaluate the data across  
## thresholds  
thresh\_code$prob = function(modelFit, newdata, submodels = NULL) {  
 out <- as.data.frame(predict(modelFit, newdata, type = "prob"))  
 if(!is.null(submodels)) {  
 probs <- out  
 out <- vector(mode = "list", length = length(submodels$threshold)+1)  
 out <- lapply(out, function(x) probs)  
 }   
 out   
 }

Basically, we define a list of model components (such as the fitting code, the prediction code, etc.) and feed this into the train function instead of using a pre-listed model string (such as method = “rf”). For this model and these data, there was an 8% increase in training time to evaluate 20 additional values of the probability cut off.

How do we optimize this model? Normally we might look at the area under the ROC curve as a metric to choose our final values. In this case the ROC curve is independent of the probability threshold so we have to use something else. A common technique to evaluate a candidate threshold is see how close it is to the perfect model where sensitivity and specificity are one. Our code will use the distance between the current model’s performance and the best possible performance and then have train minimize this distance when choosing it’s parameters. Here is the code that we use to calculate this:

library(caret)  
  
fourStats <- function (data, lev = levels(data$obs), model = NULL) {  
 ## This code will get use the area under the ROC curve and the  
 ## sensitivity and specificity values using the current candidate  
 ## value of the probability threshold.  
 out <- c(twoClassSummary(data, lev = levels(data$obs), model = NULL))  
   
 ## The best possible model has sensitivity of 1 and specificity of 1.   
 ## How far are we from that value?  
 coords <- matrix(c(1, 1, out["Spec"], out["Sens"]),   
 ncol = 2,   
 byrow = TRUE)  
 colnames(coords) <- c("Spec", "Sens")  
 rownames(coords) <- c("Best", "Current")  
 c(out, Dist = dist(coords)[1])  
}  
  
set.seed(949)  
mod1 <- train(Class ~ ., data = trainingSet,  
 method = thresh\_code,  
 ## Minimize the distance to the perfect model  
 metric = "Dist",  
 maximize = FALSE,  
 tuneLength = 20,  
 ntree = 1000,  
 trControl = trainControl(method = "repeatedcv",  
 repeats = 5,  
 classProbs = TRUE,  
 summaryFunction = fourStats))  
  
mod1

Using ggplot(mod1) will show the performance profile. Instead here is a plot of the sensitivity, specificity, and distance to the perfect model:

library(reshape2)  
metrics <- mod1$results[, c(2, 4:6)]  
metrics <- melt(metrics, id.vars = "threshold",   
 variable.name = "Resampled",  
 value.name = "Data")  
  
ggplot(metrics, aes(x = threshold, y = Data, color = Resampled)) +   
 geom\_line() +   
 ylab("") + xlab("Probability Cutoff") +  
 theme(legend.position = "top")

You can see that as we increase the probability cut off for the first class it takes more and more evidence for a sample to be predicted as the first class. As a result the sensitivity goes down when the threshold becomes very large. The upside is that we can increase specificity in the same way. The blue curve shows the distance to the perfect model. The value of 0.89 was found to be optimal.

Now we can use the test set ROC curve to validate the cut off we chose by resampling. Here the cut off closest to the perfect model is 0.88. We were able to find a good probability cut off value without setting aside another set of data for tuning the cut off.

One great thing about this code is that it will automatically apply the optimized probability threshold when predicting new samples.

## Illustrative example 6: Offset in generalized linear models

Like the mboost example above, a custom method is required since a formula element is used to set the offset variable. Here is an example from ?glm:

We can write a small custom method to duplicate this model. Two details of note:

* If we have factors in the data and do not want train to convert them to dummy variables, the formula method for train should be avoided. We can let glm do that inside the custom method. This would help glm understand that the dummy variable columns came from the same original factor. This will avoid errors in other functions used with glm (e.g. anova).
* The slot for x should include any variables that are on the right-hand side of the model formula, including the offset column.

Here is the custom model:

offset\_mod <- getModelInfo("glm", regex = FALSE)[[1]]  
offset\_mod$fit <- function(x, y, wts, param, lev, last, classProbs, ...) {  
 dat <- if(is.data.frame(x)) x else as.data.frame(x)  
 dat$Postwt <- y  
 glm(Postwt ~ Prewt + Treat + offset(Prewt), family = gaussian, data = dat)  
}  
  
mod <- train(x = anorexia[, 1:2], y = anorexia$Postwt, method = offset\_mod)  
coef(mod$finalModel)

# Adaptive Resampling

Models can benefit significantly from tuning but the optimal values are rarely known beforehand. train can be used to define a grid of possible points and resampling can be used to generate good estimates of performance for each tuning parameter combination.

However, in the nominal resampling process, all the tuning parameter combinations are computed for all the resamples before a choice is made about which parameters are good and which are poor.

caret contains the ability to adaptively resample the tuninig parameter grid in a way that concentrates on values that are in the neighborhood of the optimal settings. See [this paper](https://arxiv.org/abs/1405.6974).

To illustrate, we will use the Sonar data from one of the previous pages.

library(mlbench)  
data("Sonar")  
  
library(caret)  
  
inTraining <- createDataPartition(Sonar$Class, p=.75, list=FALSE)  
training <- Sonar[inTraining,]  
testing <- Sonar[-inTraining,]

We will tune a support vector machine model using the same tuning strategy as before but with [random search](https://topepo.github.io/caret/random-hyperparameter-search.html):

svmControl <- trainControl(method = "repeatedcv",  
 number = 10, repeats = 10,  
 classProbs = TRUE,  
 summaryFunction = twoClassSummary,  
 search = "random")  
set.seed(825)  
svmFit <- train(Class ~ ., data = training,  
 method = "svmRadial",   
 trControl = svmControl,   
 preProc = c("center", "scale"),  
 metric = "ROC",  
 tuneLength = 15)  
svmFit

## Support Vector Machines with Radial Basis Function Kernel   
##   
## 157 samples  
## 60 predictor  
## 2 classes: 'M', 'R'   
##   
## Pre-processing: centered (60), scaled (60)   
## Resampling: Cross-Validated (10 fold, repeated 10 times)   
## Summary of sample sizes: 141, 141, 142, 141, 141, 142, ...   
## Resampling results across tuning parameters:  
##   
## sigma C ROC Sens Spec   
## 0.001762205 0.07151747 0.8261508 0.5013889 0.8791071  
## 0.002009736 113.04693607 0.8854712 0.8454167 0.7592857  
## 0.002039922 171.55325987 0.8869519 0.8436111 0.7664286  
## 0.002554611 170.97876908 0.8959970 0.8545833 0.7732143  
## 0.003056140 0.27512524 0.8272247 0.5645833 0.8423214  
## 0.003229103 0.06530343 0.8250818 0.5266667 0.8708929  
## 0.003548214 0.03174726 0.8250843 0.5373611 0.8625000  
## 0.005031725 9.48257431 0.8997569 0.8343056 0.7732143  
## 0.006202988 110.85514925 0.9230332 0.8888889 0.7962500  
## 0.006841192 0.74677405 0.8816022 0.7961111 0.7437500  
## 0.007744235 4.99791627 0.9176538 0.8670833 0.7966071  
## 0.016339163 20.36369457 0.9536012 0.9238889 0.8317857  
## 0.017966687 459.42513969 0.9537946 0.9256944 0.8350000  
## 0.038701701 9.53882007 0.9473264 0.9355556 0.8078571  
## 0.068469613 275.28084205 0.9362872 0.9305556 0.7316071  
##   
## ROC was used to select the optimal model using the largest value.  
## The final values used for the model were sigma = 0.01796669 and C  
## = 459.4251.

Using this method, the optimal tuning parameters were a RBF kernel parameter of 0.0174 and a cost value of 459.4251397. To use the adaptive procedure, the trainControl option needs some additional arguments:

* min is the minimum number of resamples that will be used for each tuning parameter. The default value is 5 and increasing it will decrease the speed-up generated by adaptive resampling but should also increase the likelihood of finding a good model.
* alpha is a confidence level that is used to remove parameter settings. To date, this value has not shown much of an effect.
* method is either "gls" for a linear model or "BT" for a Bradley-Terry model. The latter may be more useful when you expect the model to do very well (e.g. an area under the ROC curve near 1) or when there are a large number of tuning parameter settings.
* complete is a logical value that specifies whether train should generate the full resampling set if it finds an optimal solution before the end of resampling. If you want to know the optimal parameter settings and don’t care much for the estimated performance value, a value of FALSE would be appropriate here.

The new code is below. Recall that setting the random number seed just prior to the model fit will ensure the same resamples as well as the same random grid.

adaptControl <- trainControl(method="adaptive\_cv",  
 number=10,  
 repeats=10,  
 adaptive=list(min=5, alpha=0.05,  
 method="gls",complete=T),  
 classProbs=TRUE,  
 summaryFunction = twoClassSummary,  
 search="random")  
  
  
set.seed(825)  
svmAdapt <- train(Class ~ ., data = training,  
 method = "svmRadial",   
 trControl = adaptControl,   
 preProc = c("center", "scale"),  
 metric = "ROC",  
 tuneLength = 15)

##   
## Attaching package: 'kernlab'

## The following object is masked from 'package:purrr':  
##   
## cross

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

The search finalized the tuning parameters on the 13th iteration of resampling and was 2.1-fold faster than the original analysis. Here, the optimal tuning parameters were a RBF kernel parameter of 0.0158 and a cost value of 20.3636946. These are close to the previous settings and result in a difference in the area under the ROC curve of 0.002 and the adaptive approach used 1308 fewer models.

Remember that this methodology is experimental, so please send any questions or bug reports to the package maintainer.

# Variable importance

**Variable importance evaluation** functions can be separated into two groups: those that use the model information and those that do not. The advantage of using a model-based approach is that is more closely tied to the model performance and that it may be able to incorporate the correlation structure between the predictors into the importance calculation. Regardless of how the importance is calculated:

* For most classification models, each predictor will have a separate variable importance for each class (the exceptions are classification trees, bagged trees and boosted trees).
* All measures of importance are scaled to have a maximum value of 100, unless the scale argument of varImp.train is set to FALSE.

## Model specific metrics

The following methods for estimating the contribution of each variable to the model are available:

* **Linear Models**: the absolute value of the *t-statistic* for each model parameter is used.
* **Random Forest**: from the R package: “For each tree, the *prediction accuracy on the out-of-bag portion* of the data is recorded. Then the same is done after permuting each predictor variable. The difference between the two accuracies are then averaged over all trees, and normalized by the standard error. For regression, the MSE is computed on the out-of-bag data for each tree, and then the same computed after permuting a variable. The differences are averaged and normalized by the standard error. If the standard error is equal to 0 for a variable, the division is not done.”
* **Partial Least Squares**: the variable importance measure here is based on *weighted sums of the absolute regression coefficients*. The weights are a function of the reduction of the sums of squares across the number of PLS components and are computed separately for each outcome. Therefore, the contribution of the coefficients are weighted proportionally to the reduction in the sums of squares.
* **Recursive Partitioning**: The reduction in the loss function (e.g. mean squared error) attributed to each variable at each split is tabulated and the sum is returned. Also, since there may be candidate variables that are important but are not used in a split, the top competing variables are also tabulated at each split. This can be turned off using themaxcompete argument in rpart.control. This method does not currently provide class-specific measures of importance when the response is a factor.
* **Bagged Trees**: The same methodology as a single tree is applied to all bootstrapped trees and the total importance is returned
* **Boosted Trees**: This method uses the same approach as a single tree, but sums the importances over each boosting iteration (see the gbm package vignette).
* **Multivariate Adaptive Regression Splines**: MARS models include a backwards elimination feature selection routine that looks at reductions in the generalized cross-validation (GCV) estimate of error. The varImp function tracks the changes in model statistics, such as the GCV, for each predictor and accumulates the reduction in the statistic when each predictor’s feature is added to the model. This total reduction is used as the variable importance measure. If a predictor was never used in any MARS basis function, it has an importance value of zero. There are three statistics that can be used to estimate variable importance in MARS models. Using varImp(object, value = "gcv") tracks the reduction in the generalized cross-validation statistic as terms are added. However, there are some cases when terms are retained in the model that result in an increase in GCV. Negative variable importance values for MARS are set to zero. Terms with non-zero importance that were not included in the final, pruned model are also listed as zero. Alternatively, using varImp(object, value = “rss”) monitors the change in the residual sums of squares (RSS) as terms are added, which will never be negative. Also, the option varImp(object, value = "nsubsets") returns the number of times that each variable is involved in a subset (in the final, pruned model). Prior to June 2008, varImp used an internal function to estimate importance for MARS models. Currently, it is a wrapper around the evimp function in the earth package.
* **Nearest shrunken centroids**: The difference between the class centroids and the overall centroid is used to measure the variable influence (see pamr.predict). The larger the difference between the class centroid and the overall center of the data, the larger the separation between the classes. The training set predictions must be supplied when an object of class pamrtrained is given to varImp.
* **Cubist**: The Cubist output contains variable usage statistics. It gives the percentage of times where each variable was used in a condition and/or a linear model. Note that this output will probably be inconsistent with the rules shown in the output from summary.cubist. At each split of the tree, Cubist saves a linear model (after feature selection) that is allowed to have terms for each variable used in the current split or any split above it. Quinlan (1992) discusses a smoothing algorithm where each model prediction is a linear combination of the parent and child model along the tree. As such, the final prediction is a function of all the linear models from the initial node to the terminal node. The percentages shown in the Cubist output reflects all the models involved in prediction (as opposed to the terminal models shown in the output). The variable importance used here is a linear combination of the usage in the rule conditions and the model.

## Model Independent Metrics

If there is no model-specific way to estimate importance (or the argument useModel = FALSE is used in varImp) the importance of each predictor is evaluated individually using a “filter” approach.

For classification, ROC curve analysis is conducted on each predictor. For two class problems, a series of cutoffs is applied to the predictor data to predict the class. The sensitivity and specificity are computed for each cutoff and the ROC curve is computed. The trapezoidal rule is used to compute the area under the ROC curve. This area is used as the measure of variable importance. For multi-class outcomes, the problem is decomposed into all pair-wise problems and the area under the curve is calculated for each class pair (i.e. class 1 vs. class 2, class 2 vs. class 3 etc.). For a specific class, the maximum area under the curve across the relevant pair-wise AUC’s is used as the variable importance measure.

For regression, the relationship between each predictor and the outcome is evaluated. An argument, nonpara, is used to pick the model fitting technique. When nonpara = FALSE, a linear model is fit and the absolute value of the t-value for the slope of the predictor is used. Otherwise, a loess smoother is fit between the outcome and the predictor. The R2 statistic is calculated for this model against the intercept only null model. This number is returned as a relative measure of variable importance.

## An example

On the model training web, several models were fit to the example data. The boosted tree model has a built-in variable importance score but neither the support vector machine or the regularized discriminant analysis model do.

gbmImp <- varImp(gmbFit3, scale=FALSE)  
gbmImp

The function automatically scales the importance scores to be between 0 and 100. Using scale = FALSE avoids this normalization step.

To get the area under the ROC curve for each predictor, the filterVarImp function can be used. The area under the ROC curve is computed for each class.

roc\_imp <- filterVarImp(x=training[,-ncol(training)],y=training$Class)  
head(roc\_imp)

## M R  
## V1 0.6636497 0.6636497  
## V2 0.6488911 0.6488911  
## V3 0.6296477 0.6296477  
## V4 0.7007502 0.7007502  
## V5 0.6596543 0.6596543  
## V6 0.6382909 0.6382909

Alternatively, for models where no built-in importance score is implemented (or exists), the varImp can still be used to get scores. For SVM classification models, the default behavior is to compute the area under the ROC curve.

roc\_imp2 <- varImp(svmFit, scale=FALSE)  
roc\_imp2

For importance scores generated from varImp.train, a plot method can be used to visualize the results. In the plot below, the top option is used to make the image more readable.

plot(gbmImp,top=20)

# Miscellaneous model functions

## Yet Another k-Nearest Neighbor Function

knn3 is a function for *k-nearest* neighbor classification. This particular implementation is a modification of the knn C code and returns the vote information for all of the classes ( knn only returns the probability for the winning class). There is a formula interface via

caret::knn3(formula, data)  
  
## or by passing the training data directly  
## x is a matrix or data frame, y is a factor vector  
knn3(x, y)

There are also print and predict methods. For the Sonar data in the mlbench package, we can fit an 11-nearest neighbor model:

library(caret)  
library(mlbench)  
  
data(Sonar)  
set.seed(808)  
  
inTrain <- createDataPartition(Sonar$Class,p=2/3, list=FALSE)  
## save the predictors and class in different objects  
sonarTrain <- Sonar[inTrain, -ncol(Sonar)]  
sonarTest <- Sonar[-inTrain, -ncol(Sonar)]  
  
trainClass <- Sonar[inTrain, "Class"]  
testClass <- Sonar[-inTrain,"Class"]  
  
centerScale <- preProcess(sonarTrain)  
centerScale

## Created from 139 samples and 60 variables  
##   
## Pre-processing:  
## - centered (60)  
## - ignored (0)  
## - scaled (60)

training <- predict(centerScale, sonarTrain)  
testing <- predict(centerScale, sonarTest)  
  
knnFit <- knn3(training, trainClass, k=11)  
knnFit

## 11-nearest neighbor model  
## Training set outcome distribution:  
##   
## M R   
## 74 65

predict(knnFit, head(testing),type="prob")

## M R  
## [1,] 0.2727273 0.7272727  
## [2,] 0.2727273 0.7272727  
## [3,] 0.1818182 0.8181818  
## [4,] 0.1818182 0.8181818  
## [5,] 0.5454545 0.4545455  
## [6,] 0.0000000 1.0000000

Similarly, caret contains a k-nearest neighbor regression function, knnreg. It returns the average outcome for the neighbor.

## 16.2 Partial Least Squares Discriminant Analysis

The plsda function is a wrapper for the plsr function in the pls package that does not require a formula interface and can take factor outcomes as arguments. The classes are broken down into dummy variables (one for each class). These 0/1 dummy variables are modeled by partial least squares.

From this model, there are two approaches to computing the class predictions and probabilities:

* the softmax technique can be used on a per-sample basis to normalize the scores so that they are more “probability like”" (i.e. they sum to one and are between zero and one). For a vector of model predictions for each class X, the softmax class probabilities are computed as. The predicted class is simply the class with the largest model prediction, or equivalently, the largest class probability. This is the default behavior for plsda.
* Bayes rule can be applied to the model predictions to form posterior probabilities. Here, the model predictions for the training set are used along with the training set outcomes to create conditional distributions for each class. When new samples are predicted, the raw model predictions are run through these conditional distributions to produce a posterior probability for each class (along with the prior). Bayes rule can be used by specifying probModel = "Bayes". An additional parameter, prior, can be used to set prior probabilities for the classes.

The advantage to using Bayes rule is that the full training set is used to directly compute the class probabilities (unlike the softmax function which only uses the current sample’s scores). This creates more realistic probability estimates but the disadvantage is that a separate Bayesian model must be created for each value of ncomp, which is more time consuming.

For the sonar data set, we can fit two PLS models using each technique and predict the class probabilities for the test set.

plsFit <- plsda(training, trainClass, ncomp=20)  
plsFit

## Partial least squares classification, fitted with the kernel algorithm.  
## The softmax function was used to compute class probabilities.

plsBayesFit <- plsda(training, trainClass, ncomp=20,  
 probMethod = "Bayes")  
plsBayesFit

## Partial least squares classification, fitted with the kernel algorithm.  
## Bayes rule was used to compute class probabilities.

predict(plsFit, head(testing),type="prob")

## , , 20 comps  
##   
## M R  
## 4 0.6227621 0.3772379  
## 6 0.5240553 0.4759447  
## 12 0.3883679 0.6116321  
## 16 0.1925378 0.8074622  
## 17 0.1800970 0.8199030  
## 19 0.1336795 0.8663205

predict(plsBayesFit, head(testing), type = "prob")

## , , ncomp20  
##   
## M R  
## 4 0.950775270 0.04922473  
## 6 0.585468690 0.41453131  
## 12 0.076098620 0.92390138  
## 16 0.002768591 0.99723141  
## 17 0.003715369 0.99628463  
## 19 0.023820018 0.97617998

Similar to plsda, caret also contains a function splsda that allows for classification using sparse PLS. A dummy matrix is created for each class and used with the spls function in the spls package. The same approach to estimating class probabilities is used forplsda and splsda.

## Bagging

The bag function offers a general platform for bagging classification and regression models. Like rfe and sbf, it is open and models are specified by declaring functions for the model fitting and prediction code (and several built-in sets of functions exist in the package). The function bagControl has options to specify the functions (more details below).

The function also has a few non-standard features: - The argument var can enable random sampling of the predictors at each bagging iteration. This is to de-correlate the bagged models in the same spirit of random forests (although here the sampling is done once for the whole model). The default is to use all the predictors for each model.

* The bagControl function has a logical argument called downSample that is useful for classification models with severe class imbalance. The bootstrapped data set is reduced so that the sample sizes for the classes with larger frequencies are the same as the sample size for the minority class.
* If a parallel backend for the **foreach** package has been loaded and registered, the bagged models can be trained in parallel.

The function’s control function requires the following arguments:

### The fit function

inputs - x: a data frame of the training set predictor data. - y: the training set outcomes. - ... arguments passed from train to this function

The output is the object corresponding to the trained model and any other objects required for prediction. A simple example for a linear discriminant analysis model from the **MASS** package is:

function(x, y, ...) {  
 library(MASS)  
 lda(x, y, ...)  
}

### The pred function

This should be a function that produces predictors for new samples.

Inputs: - object: the object generated by the fit module. - x: a matrix or data frame of predictor data.

The output is either a number vector (for regression), a factor (or character) vector for classification or a matrix/data frame of class probabilities. For classification, it is probably better to average class probabilities instead of using the votes of the class predictions. Using the lda example again:

## predict.lda returns the class and the class probabilities  
## we will average the probabilities, so these are saved  
function(object, x) predict(object, x)$posterior

### 16.4.3 The aggregate function

This should be a function that takes the predictions from the constituent models and converts them to a single prediction per sample.

Inputs: - x: a list of objects returned by the pred module. - type: an optional string that describes the type of output (e.g. “class”, “prob” etc.).

The output is either a number vector (for regression), a factor (or character) vector for classification or a matrix/data frame of class probabilities. For the linear discriminant model above, we saved the matrix of class probabilities. To average them and generate a class prediction, we could use:

function(x, type = "class") {  
 ## The class probabilities come in as a list of matrices  
 ## For each class, we can pool them then average over them  
   
 ## Pre-allocate space for the results  
 pooled <- x[[1]] \* NA  
 n <- nrow(pooled)  
 classes <- colnames(pooled)  
 ## For each class probability, take the median across   
 ## all the bagged model predictions  
 for(i in 1:ncol(pooled))  
 {  
 tmp <- lapply(x, function(y, col) y[,col], col = i)  
 tmp <- do.call("rbind", tmp)  
 pooled[,i] <- apply(tmp, 2, median)  
 }  
 ## Re-normalize to make sure they add to 1  
 pooled <- apply(pooled, 1, function(x) x/sum(x))  
 if(n != nrow(pooled)) pooled <- t(pooled)  
 if(type == "class")  
 {  
 out <- factor(classes[apply(pooled, 1, which.max)],  
 levels = classes)  
 } else out <- as.data.frame(pooled)  
 out  
}

For example, to bag a conditional inference tree (from the party package):

library(caret)  
library(party)

## Loading required package: mvtnorm

## Loading required package: modeltools

## Loading required package: stats4

##   
## Attaching package: 'modeltools'

## The following object is masked from 'package:kernlab':  
##   
## prior

## The following object is masked from 'package:stabs':  
##   
## parameters

## Loading required package: strucchange

## Loading required package: zoo

##   
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':  
##   
## as.Date, as.Date.numeric

## Loading required package: sandwich

##   
## Attaching package: 'strucchange'

## The following object is masked from 'package:stringr':  
##   
## boundary

##   
## Attaching package: 'party'

## The following object is masked from 'package:mboost':  
##   
## varimp

set.seed(998)  
inTraining <- createDataPartition(Sonar$Class, p = .75, list = FALSE)  
training <- Sonar[ inTraining,]  
testing <- Sonar[-inTraining,]  
set.seed(825)  
baggedCT <- bag(x = training[, names(training) != "Class"],  
 y = training$Class,  
 B = 50,  
 bagControl = bagControl(fit = ctreeBag$fit,  
 predict = ctreeBag$pred,  
 aggregate = ctreeBag$aggregate))   
summary(baggedCT)

##   
## Call:  
## bag.default(x = training[, names(training) != "Class"], y  
## = training$Class, B = 50, bagControl = bagControl(fit =  
## ctreeBag$fit, predict = ctreeBag$pred, aggregate = ctreeBag$aggregate))  
##   
## Out of bag statistics (B = 50):  
##   
## Accuracy Kappa  
## 0.0% 0.5000 -0.004065  
## 2.5% 0.5357 0.014293  
## 25.0% 0.6667 0.333794  
## 50.0% 0.7027 0.397853  
## 75.0% 0.7531 0.489808  
## 97.5% 0.8009 0.573063  
## 100.0% 0.8226 0.641430

## Model averaged neural networks

The avNNet fits multiple neural network models to the same data set and predicts using the average of the predictions coming from each constituent model. The models can be different either due to different random number seeds to initialize the network or by fitting the models on bootstrap samples of the original training set (i.e. bagging the neural network). For classification models, the class probabilities are averaged to produce the final class prediction (as opposed to voting from the individual class predictions.

As an example, the model can be fit via train:

set.seed(825)   
avNnetFit <- train(x = training,  
 y = trainClass,  
 method = "avNNet",   
 repeats = 15,  
 trace = FALSE)

## Neural Networks with a Principal Component Step

Neural networks can be affected by severe amounts of multicollinearity in the predictors. Th function pcaNNet is a wrapper around the preProcess and nnet functions that will run principal component analysis on the predictors before using them as inputs into a neural network.

The function will keep enough components that will capture some pre-defined threshold on the cumulative proportion of variance (see the thresh argument). For new samples, the same transformation is applied to the new predictor values (based on the loadings from the training set). The function is available for both regression and classification.

This function is deprecated in favor of the train function using method = "nnet" and preProc = "pca".

## Independent component regression

The icr function can be used to fit a model analogous to principal component regression (PCR), but using independent component analysis (ICA). The predictor data are centered and projected to the ICA components. These components are then regressed against the outcome. The user needed to specify the number of components to keep.

The model uses the preProcess function to compute the latent variables using the fastICA package.

Like PCR, there is no guarantee that there will be a correlation between the new latent variable and the outcomes.

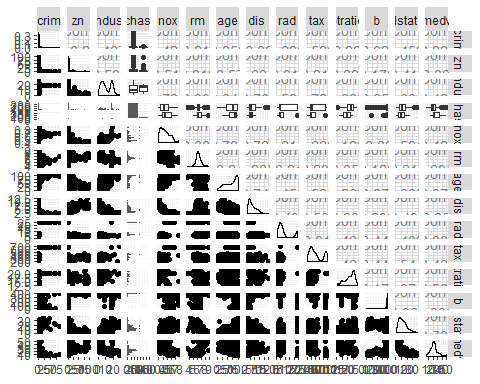
# Measuring performance

## Measures for Regression

The function postResample can be used to estimate the root mean squared error (RMSE), simpled and the mean absolute error (MAE) for numeric outcomes. For example:

library(caret)  
library(AppliedPredictiveModeling)  
library(mlbench)  
data(BostonHousing)  
  
set.seed(280)  
bh\_index <- createDataPartition(BostonHousing$medv, p=.75, list=FALSE)  
bh\_tr <- BostonHousing[bh\_index,]  
bh\_te <- BostonHousing[-bh\_index,]  
  
set.seed(7229)  
lm\_fit <- train(medv~. + rm:lstat,  
 data=bh\_tr,  
 method="lm")  
bh\_pred <- predict(lm\_fit, bh\_te)  
  
GGally::ggpairs(BostonHousing)

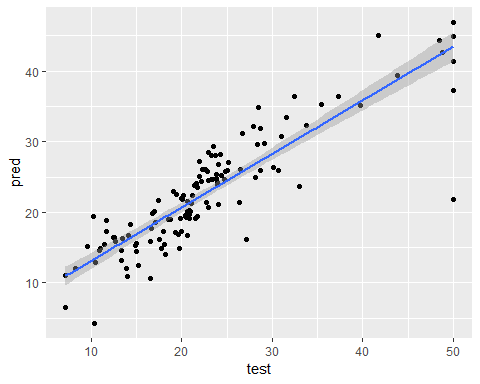
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.  
## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.



lm\_fit

## Linear Regression   
##   
## 381 samples  
## 13 predictor  
##   
## No pre-processing  
## Resampling: Bootstrapped (25 reps)   
## Summary of sample sizes: 381, 381, 381, 381, 381, 381, ...   
## Resampling results:  
##   
## RMSE Rsquared MAE   
## 4.250365 0.777112 2.928805  
##   
## Tuning parameter 'intercept' was held constant at a value of TRUE

library(tidyverse)  
  
summary <- tibble::tibble(  
 test = bh\_te$medv,  
 pred = bh\_pred)  
summary %>%   
 ggplot(aes(test, pred))+  
 geom\_point()+  
 geom\_smooth(method = "lm")



postResample(pred=bh\_pred, obs=bh\_te$medv)

## RMSE Rsquared MAE   
## 4.4412987 0.7781877 3.0926842

A note about how is calculated by caret: it takes the straightforward approach of computing the correlation between the observed and predicted values (i.e. R) and squaring the value. When the model is poor, this can lead to differences between this estimator and the more widely known estimate derived form linear regression models. Mostly notably, the correlation approach will not generate negative values of (which are theoretically invalid). A comparison of these and other estimators can be found in Kvalseth 1985.

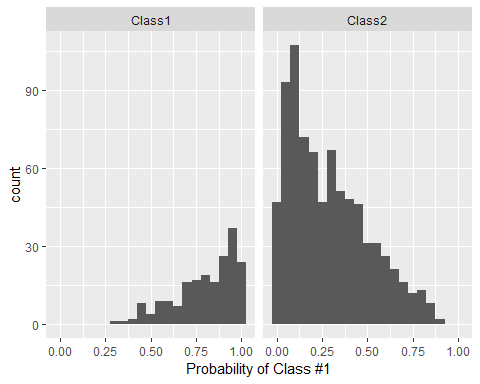
## Measures for predicted classes

Before proceeding, let’s make up some test set data:

set.seed(144)  
true\_class <- factor(sample(paste0("Class",1:2),  
 size=1000,  
 prob=c(.2, .8),replace=TRUE))  
  
true\_class <- sort(true\_class)  
class1\_probs <- rbeta(sum(true\_class == "Class1"), 4, 1)  
class2\_probs <- rbeta(sum(true\_class == "Class2"), 1, 2.5)  
test\_set <- data.frame(obs = true\_class,  
 Class1 = c(class1\_probs, class2\_probs))  
test\_set$Class2 <- 1 - test\_set$Class1  
test\_set$pred <- factor(ifelse(test\_set$Class1 >= .5, "Class1", "Class2"))

We would expect that this model will do well on these data:

ggplot(test\_set, aes(x=Class1))+  
 geom\_histogram(binwidth=.05)+  
 facet\_wrap(~obs)+  
 xlab("Probability of Class #1")



Generating the predicted classes based on the typical 50% cutoff for the probabilities, we can compute the confusion matrix, which shows a cross-tabulation of the observed and predicted classes. The confusionMatrix function can be used to generate these results:

confusionMatrix(data=test\_set$pred, reference = test\_set$obs)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction Class1 Class2  
## Class1 183 141  
## Class2 13 663  
##   
## Accuracy : 0.846   
## 95% CI : (0.8221, 0.8678)  
## No Information Rate : 0.804   
## P-Value [Acc > NIR] : 0.0003424   
##   
## Kappa : 0.6081   
## Mcnemar's Test P-Value : < 2.2e-16   
##   
## Sensitivity : 0.9337   
## Specificity : 0.8246   
## Pos Pred Value : 0.5648   
## Neg Pred Value : 0.9808   
## Prevalence : 0.1960   
## Detection Rate : 0.1830   
## Detection Prevalence : 0.3240   
## Balanced Accuracy : 0.8792   
##   
## 'Positive' Class : Class1   
##

For two classes, this function assumes that the class corresponding to an event is the first class level (but this can be changed using the positive argument.

Note that there are a number of statistics shown here. The **“no-information rate”** is the largest proportion of the observed classes (there were more class 2 data than class 1 in this test set). A hypothesis test is also computed to evaluate whether the overall accuracy rate is greater than the rate of the largest class. Also, the prevalence of the “positive event” is computed from the data (unless passed in as an argument), the detection rate (the rate of true events also predicted to be events) and the detection prevalence (the prevalence of predicted events).

If the prevalence of the event is different than those seen in the test set, the prevalence option can be used to adjust this.

Suppose 2\*2 table: |Predicted/Reference|Event|No event| |:—|:—| |Event|A|B| |No Event|C|D|

The formulas used here are:

When there are three or more classes, confusionMatrix will show the confusion matrix and a set of “one-versus-all” results. For example, in a three class problem, the sensitivity of the first class is calculated against all the samples in the second and third classes (and so on).

The confusionMatrix matrix frames the errors in terms of sensitivity and specificity. In the case of information retrieval, the precision and recall might be more appropriate. In this case, the option mode can be used to get those statistics:

confusionMatrix(data=test\_set$pred, reference=test\_set$obs, mode="prec\_recall")

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction Class1 Class2  
## Class1 183 141  
## Class2 13 663  
##   
## Accuracy : 0.846   
## 95% CI : (0.8221, 0.8678)  
## No Information Rate : 0.804   
## P-Value [Acc > NIR] : 0.0003424   
##   
## Kappa : 0.6081   
## Mcnemar's Test P-Value : < 2.2e-16   
##   
## Precision : 0.5648   
## Recall : 0.9337   
## F1 : 0.7038   
## Prevalence : 0.1960   
## Detection Rate : 0.1830   
## Detection Prevalence : 0.3240   
## Balanced Accuracy : 0.8792   
##   
## 'Positive' Class : Class1   
##

Again, the positive argument can be used to control which factor level is associated with a “found” or “important” document or sample.

There are individual functions called sensitivity, specificity, posPredValue, negPredValue, precision, recall, and F\_meas.

Also, a resampled estimate of the training set can also be obtained using confusionMatrix.train. For each resampling iteration, a confusion matrix is created from the hold-out samples and these values can be aggregated to diagnose issues with the model fit.

These values are the percentages that hold-out samples landed in the confusion matrix during resampling. There are several methods for normalizing these values. See ?confusionMatrix.train for details.

The default performance function used by train is postResample, which generates the accuracy and Kappa statistics:

postResample(pred=test\_set$pred, obs=test\_set$obs)

## Accuracy Kappa   
## 0.8460000 0.6081345

As shown below, another function called twoClassSummary can be used to get the sensitivity and specificity using the default probability cutoff. Another function, multiClassSummary, can do similar calculations when there are three or more classes but both require class probabilities for each class.

## Measures for class probabilities

For data with two classes, there are specialized functions for measuring model performance. First, the twoClassSummary function computes the area under the ROC curve and the specificity and sensitivity under the 50% cutoff. Note that:

* this function uses the first class level to define the “event” of interest. To change this, use the lev option to the function
* there must be columns in the data for each of the class probabilities (named the same as the outcome’s class levels)

twoClassSummary(test\_set, lev=levels(test\_set$obs))

## ROC Sens Spec   
## 0.9560044 0.9336735 0.8246269

A similar function can be used to get the analugous precision-recall values and the area under the precision-recall curve:

prSummary(test\_set, lev=levels(test\_set$obs))

## AUC Precision Recall F   
## 0.8582695 0.5648148 0.9336735 0.7038462

This function requires that the MLmetrics package is installed.

For multi-class problems, there are additional functions that can be used to calculate performance. One, mnLogLoss computes the negative of the multinomial log-likelihood (smaller is better) based on the class probabilities. This can be used to optimize tuning parameters but can lead to results that are inconsistent with other measures (e.g. accuracy or the area under the ROC curve), especially when the other measures are near their best possible values. The function has similar arguments to the other functions described above. Here is the two-class data from above:

mnLogLoss(test\_set, lev=levels(test\_set$obs))

## logLoss   
## 0.370626

Additionally, the function multiClassSummary computes a number of relevant metrics:

* the overall accuracy and Kappa statistics using the predicted classes
* the negative of the multinomial log loss (if class probabilities are available)
* averages of the “one versus all” statistics such as sensitivity, specificity, the area under the ROC curve, etc.

## Lift curves

The lift function can be used to evaluate probabilities thresholds that can capture a certain percentage of hits. The function requires a set of sample probability predictions (not from the training set) and the true class labels. For example, we can simulate two-class samples using the twoClassSim function and fit a set of models to the training set:

set.seed(2)  
lift\_training <- twoClassSim(1000)  
lift\_testing <- twoClassSim(1000)  
  
ctrl <- trainControl(method = "cv", classProbs = TRUE,  
 summaryFunction = twoClassSummary)  
  
set.seed(1045)  
fda\_lift <- train(Class ~ ., data = lift\_training,  
 method = "fda", metric = "ROC",  
 tuneLength = 20,  
 trControl = ctrl)  
set.seed(1045)  
lda\_lift <- train(Class ~ ., data = lift\_training,  
 method = "lda", metric = "ROC",  
 trControl = ctrl)  
  
library(C50)  
set.seed(1045)  
c5\_lift <- train(Class ~ ., data = lift\_training,  
 method = "C5.0", metric = "ROC",  
 tuneLength = 10,  
 trControl = ctrl,  
 control = C5.0Control(earlyStopping = FALSE))  
  
## Generate the test set results  
lift\_results <- data.frame(Class = lift\_testing$Class)  
lift\_results$FDA <- predict(fda\_lift, lift\_testing, type = "prob")[,"Class1"]  
lift\_results$LDA <- predict(lda\_lift, lift\_testing, type = "prob")[,"Class1"]  
lift\_results$C5.0 <- predict(c5\_lift, lift\_testing, type = "prob")[,"Class1"]  
head(lift\_results)

The lift function does the calculations and the corresponding plot function is used to plot the lift curve (although some call this the **gain curve**). The value argument creates reference lines:

trellis.par.set(caretTheme())  
lift\_obj <- lift(Class ~ FDA + LDA + C5.0, data = lift\_results)  
plot(lift\_obj, values = 60, auto.key = list(columns = 3,  
 lines = TRUE,  
 points = FALSE))

There is also a ggplot method for lift objects:

ggplot(lift.obj, values=60)

From this we can see that, to find 60 percent of the hits, a little more than 30 percent of the data can be sampled (when ordered by the probability predictions). The LDA model does somewhat worse than the other two models.

## Calibration curves

Calibration curves can be used to characterisze how consistent the predicted class probabilities are with the observed event rates.

Other functions in the gbm package, the rms package (and others) can also produce calibrartion curves. The format for the function is very similar to the lift function:

trellis.par.set(caretTheme())  
cal\_obj <- calibration(Class~FDA+LDA+C5.0,  
 data=lift\_results,  
 cuts=13)  
  
plot(cal\_obj,type="l",auto.key=list(columns=3,  
 lines=TRUE,  
 points=FALSE))

There is also a ggplot method that shows the confidence intervals for the proportions inside of the subsets:

ggplot(cal\_obj)

# Feature selection overview

## Models with built-in feature selection

Many models that can be accessed using caret’s train function produce prediction equations that do not necessarily use all the predictors.

These models are thought to have built-in feature selection: ada, AdaBag, AdaBoost.M1, adaboost, bagEarth, bagEarthGCV, bagFDA, bagFDAGCV, bartMachine, blasso, BstLm, bstSm, C5.0, C5.0Cost, C5.0Rules, C5.0Tree, cforest, chaid, ctree, ctree2, cubist, deepboost, earth, enet, evtree, extraTrees, fda, gamboost, gbm\_h2o, gbm, gcvEarth, glmnet\_h2o, glmnet, glmStepAIC, J48, JRip, lars, lars2, lasso, LMT, LogitBoost, M5, M5Rules, msaenet, nodeHarvest, OneR, ordinalNet, ORFlog, ORFpls, ORFridge, ORFsvm, pam, parRF, PART, penalized, PenalizedLDA, qrf, ranger, Rborist, relaxo, rf, rFerns, rfRules, rotationForest, rotationForestCp, rpart, rpart1SE, rpart2, rpartCost, rpartScore, rqlasso, rqnc, RRF, RRFglobal, sdwd, smda, sparseLDA, spikeslab, wsrf, xgbDART, xgbLinear, xgbTree.

any of the functions have an ancillary method called predictors that returns a vector indicating which predictors were used in the final model.

In many cases, using these models with built-in feature selection will be more efficient than algorithms where the search routine for the right predictors is external to the model. Built-in feature selection typically couples the predictor search algorithm with the parameter estimation and are usually optimized with a single objective function (e.g. error rates or likelihood).

## Feature selection methods

Apart from models with built-in feature selection, most approaches for reducing the number of predictors can be placed into two main categories. Using the terminology of John, Kohavi, and Pfleger (1994):

* **Wrapper methods** evaluate multiple models using procedures that add and/or remove predictors to find the optimal combination that maximizes model performance. In essence, wrapper methods are search algorithms that treat the predictors as the inputs and utilize model performance as the output to be optimized. caret has wrapper methods based on recursive feature elimination, genetic algorithms, and simulated annealing.
* **Filter methods** evaluate the relevance of the predictors outside of the predictive models and subsequently model only the predictors that pass some criterion. For example, for classification problems, each predictor could be individually evaluated to check if there is a plausible relationship between it and the observed classes. Only predictors with important relationships would then be included in a classification model. [Saeys, Inza, and Larranaga (2007)](https://scholar.google.com/scholar?q=%22A+review+of+feature+selection+techniques+in+bioinformatics) surveys filter methods. caret has a general framework for using [univariate filters](http://topepo.github.io/caret/feature-selection-using-univariate-filters.html).

Both approaches have advantages and drawbacks. **Filter methods** are usually more computationally efficient than wrapper methods, but the selection criterion is not directly related to the effectiveness of the model.

Also, most **filter methods** evaluate each predictor separately and, consequently, redundant (i.e. highly-correlated) predictors may be selected and important interactions between variables will not be able to be quantified. The downside of **the wrapper method** is that many models are evaluated (which may also require parameter tuning) and thus an increase in computation time. There is also an increased risk of over-fitting with wrappers.

## External validation

It is important to realize that feature selection is part of the model building process and, as such, should be externally validated. Just as parameter tuning can result in over-fitting, feature selection can over-fit to the predictors (especially when search wrappers are used). In each of the caret functions for feature selection, the selection process is included in any resampling loops. See

See [Ambroise and McLachlan (2002)](https://scholar.google.com/scholar?q=%22Selection+bias+in+gene+extraction+on+t%20e+basis+of+microarray+gene-expression+data) for a demonstration of this issue.

# Feature selection using univariate filters

## Univariate filters

Another approach to feature selection is to pre-screen the predictors using simple univariate statistical methods then only use those that pass some criterion in the subsequent model steps. Similar to recursive selection, cross-validation of the subsequent models will be biased as the remaining predictors have already been evaluate on the data set. Proper performance estimates via resampling should include the feature selection step.

As an example, it has been suggested for classification models, that predictors can be filtered by conducting some sort of k-sample test (where is the number of classes) to see if the mean of the predictor is different between the classes. Wilcoxon tests, t-tests and ANOVA models are sometimes used. Predictors that have statistically significant differences between the classes are then used for modeling.

The caret function sbf (for selection by filter) can be used to cross-validate such feature selection schemes. Similar to rfe, functions can be passed into sbf for the computational components: univariate filtering, model fitting, prediction and performance summaries (details are given below).

The function is applied to the entire training set and also to different resampled versions of the data set. From this, generalizable estimates of perfomance can be computed that properly take into account the feature selection step. Also, the results of the predictor filters can be tracked over resamples to understand the uncertainty in the filtering.

## Basic syntax

Similar to the rfe function, the syntax for sbf is:

sbf(predictors, outcome, sbfControl = sbfControl(), ...)  
## or  
sbf(formula, data, sbfControl = sbfControl(), ...)

In this case, the details are specificed using the sbfControl function. Here, the argument functions dictates what the different components should do. This argument should have elements called filter, fit, pred and summary.

### The score function

This function takes as inputs the predictors and the outcome in objects called x and y, respectively. By default, each predictor in x is passed to the score function individually. In this case, the function should return a single score. Alternatively, all the predictors can be exposed to the function using the multivariate argument to sbfControl. In this case, the output should be a named vector of scores where the names correspond to the column names of x.

There are two built-in functions called anovaScores and gamScores. anovaScores treats the outcome as the independent variable and the predictor as the outcome. In this way, the null hypothesis is that the mean predictor values are equal across the different classes. For regression, gamScores fits a smoothing spline in the predictor to the outcome using a generalized additive model and tests to see if there is any functional relationship between the two. In each function the p-value is used as the score.

### The filter function

This function takes as inputs the scores coming out of the score function (in an argument called score). The function also has the training set data as inputs (arguments are called x and y). The output should be a named logical vector where the names correspond to the column names of x. Columns with values of TRUE will be used in the subsequent model.

### The fit function

The component is very similar to the rfe-specific function described above. For sbf, there are no first or last arguments. The function should have arguments x, y and .... The data within x have been filtered using the filter function described above. The output of the fit function should be a fitted model.

With some data sets, no predictors will survive the filter. In these cases, a model with predictors cannot be computed, but the lack of viable predictors should not be ignored in the final results. To account for this issue, caret contains a model function called nullModel that fits a simple model that is independent of any of the predictors. For problems where the outcome is numeric, the function predicts every sample using the simple mean of the training set outcomes. For classification, the model predicts all samples using the most prevalent class in the training data.

This function can be used in the fit component function to “error-trap” cases where no predictors are selected. For example, there are several built-in functions for some models. The object rfSBF is a set of functions that may be useful for fitting random forest models with filtering. The fit function here uses nullModel to check for cases with no predictors:

rfSBF$fit

## function (x, y, ...)   
## {  
## if (ncol(x) > 0) {  
## loadNamespace("randomForest")  
## randomForest::randomForest(x, y, ...)  
## }  
## else nullModel(y = y)  
## }  
## <bytecode: 0x7f99bbccc9c8>  
## <environment: namespace:caret>

### The summary and pred Functions

The summary function is used to calculate model performance on held-out samples. The pred function is used to predict new samples using the current predictor set. The arguments and outputs for these two functions are identical to the previously discussed summary and pred functions in previously described sections.

## The example

Returning to the example from (Friedman, 1991), we can fit another random forest model with the predictors pre-filtered using the generalized additive model approach described previously.

filterCtrl <- sbfControl(functions = rfSBF, method = "repeatedcv", repeats = 5)  
set.seed(10)  
rfWithFilter <- sbf(x, y, sbfControl = filterCtrl)  
rfWithFilter

##   
## Selection By Filter  
##   
## Outer resampling method: Cross-Validated (10 fold, repeated 5 times)   
##   
## Resampling performance:  
##   
## RMSE Rsquared MAE RMSESD RsquaredSD MAESD  
## 3.45 0.5509 2.926 0.592 0.2132 0.5867  
##   
## Using the training set, 6 variables were selected:  
## real2, real4, real5, bogus2, bogus17...  
##   
## During resampling, the top 5 selected variables (out of a possible 12):  
## real2 (100%), real4 (100%), real5 (100%), bogus44 (72%), bogus2 (52%)  
##   
## On average, 5.6 variables were selected (min = 3, max = 8)

In this case, the training set indicated that 6 should be used in the random forest model, but the resampling results indicate that there is some variation in this number. Some of the informative predictors are used, but a few others are erroneous retained.

Similar to rfe, there are methods for predictors, densityplot, histogram and varImp.

# Recursive Feature Elimination

## Backward Selection

First, the algorithm fits the model to all predictors. Each predictor is ranked using it’s importance to the model. Let be a sequence of ordered numbers which are candidate values for the number of predictors to retain ( > , …). At each iteration of feature selection, the Si top ranked predictors are retained, the model is refit and performance is assessed. The value of with the best performance is determined and the top predictors are used to fit the final model. Algorithm 1 has a more complete definition.

The algorithm has an optional step (line 1.9) where the predictor rankings are recomputed on the model on the reduced feature set. Svetnik et al (2004) showed that, for random forest models, there was a decrease in performance when the rankings were re-computed at every step. However, in other cases when the initial rankings are not good (e.g. linear models with highly collinear predictors), re-calculation can slightly improve performance.

Algorithm 1. Tune/train the model on the training set using all predictors 2. Calculate model performance 3. Calculate variable importance or rankings 4. **for** Each *subset size* , \***do** 5. Keep the most important variables 6. [Optional] Pre-process the data 7. Tune/train the model on the training set using predictors 8. Calculate model performance 9. [Optional] Recauclate the ranmings for each predictor 10. **end** 11. Calculate the peformance profile over the 12. Determine the appropriate number of predictors 13. Use the model corresponding to the optimal

One potential issue over-fitting to the predictor set such that the wrapper procedure could focus on nuances of the training data that are not found in future samples (i.e. over-fitting to predictors and samples).

For example, suppose a very large number of uninformative predictors were collected and one such predictor randomly correlated with the outcome. The RFE algorithm would give a good rank to this variable and the prediction error (on the same data set) would be lowered. It would take a different test/validation to find out that this predictor was uninformative. The was referred to as “selection bias” by [Ambroise and McLachlan (2002)](http://www.pnas.org/content/99/10/6562.short).

In the current RFE algorithm, the training data is being used for at least three purposes: predictor selection, model fitting and performance evaluation. Unless the number of samples is large, especially in relation to the number of variables, one static training set may not be able to fulfill these needs.

## Resampling and external validation

Since feature selection is part of the model building process, resampling methods (e.g. cross-validation, the bootstrap) should factor in the variability caused by feature selection when calculating performance. For example, the RFE procedure in Algorithm 1 can estimate the model performance on line 1.7, which during the selection process. [Ambroise and McLachlan (2002)](http://www.pnas.org/content/99/10/6562.short) and [Svetnik et al (2004)](https://rd.springer.com/chapter/10.1007%2F978-3-540-25966-4_33) showed that improper use of resampling to measure performance will result in models that perform poorly on new samples.

To get performance estimates that incorporate the variation due to feature selection, it is suggested that the steps in Algorithm 1 be encapsulated inside an outer layer of resampling (e.g. 10-fold cross-validation). Algorithm 2 shows a version of the algorithm that uses resampling.

While this will provide better estimates of performance, it is more computationally burdensome. For users with access to machines with multiple processors, the first For loop in Algorithm 2 (line 2.1) can be easily parallelized. Another complication to using resampling is that multiple lists of the “best” predictors are generated at each iteration. At first this may seem like a disadvantage, but it does provide a more probabilistic assessment of predictor importance than a ranking based on a single fixed data set. At the end of the algorithm, a consensus ranking can be used to determine the best predictors to retain.

Algorithm 2: Recursive feature elimination incorporating resampling 1.**for** Each resampling iteration **do** 2. Partition data into training and test/hold-back set via resampling 3. Tune/train the model on the training set using all predictors 4. Predict the held-back samples 5. Calculate variable importantce or rankings 6. **for** Each subset size , i=1…S **do** 7. Keep the most imporatnt variables 8. [Optional] Pre-process the data 9. Tune/train the model on the training set using predictors 10. Predict the held-back samples 11. [Optional] Recalculate the rankings for each predictor 12. **End** 13.**End** 14.Calculate the performance profile over the using the held-back samples 15.Determine the appropriate number of predictors 16.Estimate the final list of predictors to keep in the final model 17.Fit the final model based on the optimal using the original training set

## Recursive feature elimination via caret

In caret, Algorithm 1 is implemented by the function rfeIter. The resampling-based Algorithm 2 is in the rfe function. Given the potential selection bias issues, this document focuses on rfe. There are several arguments:

* x, a matrix or data frame of predictor variables
* y, a vector (numeric or factor) of outcomes
* sizes, a integer vector for the specific subset sizes that should be tested (which need not to include ncol(x))
* rfeControl, a list of options that can be used to specify the model and the methods for prediction, ranking etc.

For a specific model, a set of functions must be specified in rfeControl$functions. Sections below has descriptions of these sub-functions. There are a number of pre-defined sets of functions for several models, including: linear regression (in the object lmFuncs), random forests (rfFuncs), naive Bayes (nbFuncs), bagged trees (treebagFuncs) and functions that can be used with caret’s train function (caretFuncs). The latter is useful if the model has tuning parameters that must be determined at each iteration.

## An example

library(caret)  
library(mlbench)  
library(Hmisc)

## Warning: package 'Hmisc' was built under R version 3.5.1

## Loading required package: Formula

##   
## Attaching package: 'Hmisc'

## The following objects are masked from 'package:TeachingDemos':  
##   
## cnvrt.coords, subplot

## The following objects are masked from 'package:dplyr':  
##   
## src, summarize

## The following objects are masked from 'package:base':  
##   
## format.pval, units

library(randomForest)

## Warning: package 'randomForest' was built under R version 3.5.1

## randomForest 4.6-14

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

##   
## Attaching package: 'randomForest'

## The following object is masked from 'package:dplyr':  
##   
## combine

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

To test the algorithm, the “Friedman 1” benchmark (Friedman, 1991) was used. There are five informative variables generated by the equation.

In the simulation used here:

n <- 100  
p <- 40  
sigma <- 1  
set.seed(1)  
sim <- mlbench.friedman1(n, sd = sigma)  
colnames(sim$x) <- c(paste("real", 1:5, sep = ""),  
 paste("bogus", 1:5, sep = ""))  
bogus <- matrix(rnorm(n \* p), nrow = n)  
colnames(bogus) <- paste("bogus", 5+(1:ncol(bogus)), sep = "")  
x <- cbind(sim$x, bogus)  
y <- sim$y

f the 50 predictors, there are 45 pure noise variables: 5 are uniform on

and 40 are random univariate standard normals. The predictors are centered and scaled:

normalization <- preProcess(x)  
x <- predict(normalization, x)  
x <- as.data.frame(x)  
subsets <- c(1:5, 10,15,20,25)

The simulation will fit models with subset sizes of 25, 20, 15, 10, 5, 4, 3, 2, 1.

As previously mentioned, to fit linear models, the lmFuncs set of functions can be used. To do this, a control object is created with the rfeControl function. We also specify that repeated 10-fold cross-validation should be used in line 2.1 of Algorithm 2. The number of folds can be changed via the number argument to rfeControl (defaults to 10). The verbose option prevents copious amounts of output from being produced.

set.seed(10)  
  
ctrl <- rfeControl(functions=lmFuncs,  
 method = "repeatedcv",  
 repeats=5,  
 verbose=FALSE)  
  
lmProfile <- rfe(x,y,  
 sizes=subsets,  
 rfeControl=ctrl)

The output shows that the best subset size was estimated to be 4 predictors. This set includes informative variables but did not include them all. The predictors function can be used to get a text string of variable names that were picked in the final model. The lmProfile is a list of class "rfe" that contains an object fit that is the final linear model with the remaining terms. The model can be used to get predictions for future or test samples.

predictors(lmProfile)

## [1] "real4" "real5" "real2" "real1"

lmProfile$fit

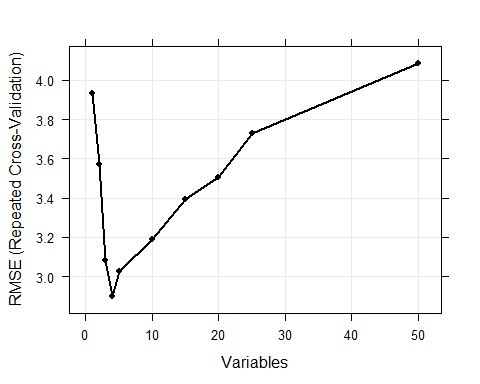
##   
## Call:  
## lm(formula = y ~ ., data = tmp)  
##   
## Coefficients:  
## (Intercept) real4 real5 real2 real1   
## 14.613 2.857 1.965 1.625 1.359

head(lmProfile$resample)

## Variables RMSE Rsquared MAE Resample  
## 4 4 2.940756 0.5542831 2.587031 Fold01.Rep1  
## 14 4 2.704529 0.6583788 2.081745 Fold02.Rep1  
## 24 4 4.599872 0.2181731 3.580316 Fold03.Rep1  
## 34 4 2.601659 0.8478823 2.243885 Fold04.Rep1  
## 44 4 1.979220 0.8571701 1.727537 Fold05.Rep1  
## 54 4 2.978896 0.8301785 2.342059 Fold06.Rep1

there are also several plot methods to visualize the results. plot(lmProfile) produces the peformance profile across different subset sizes, as shown in the figure below.

trellis.par.set(caretTheme())  
plot(lmProfile,type=c("g","o"))



Also the resampling results are stored in the sub-object lmProfile$resample and can be used with several lattice functions. Univariate lattice functions (densityplot, histogram) can be used to plot the resampling distribution while bivariate functions (xyplot, stripplot) can be used to plot the distributions for different subset sizes. In the latter case, the option returnResamp`` = "all" in rfeControl can be used to save all the resampling results. Example images are shown below for the random forest model.

## Helper functions

To use feature elimination for an arbitorary model, a set of functions must be passed to rfe for each of the steps in Algorithm 2.

This section defines those functions and uses the existing random forest functions as an illustrative example. caret contains a list called rfFuncs, but this document will use a more simple version that will be better for illustrating the ideas. A set of simplified functions sed here and called rfRFE.

rfRFE <- list(summary = defaultSummary,  
 fit = function(x, y, first, last, ...){  
 library(randomForest)  
 randomForest(x, y, importance = first, ...)  
 },  
 pred = function(object, x) predict(object, x),  
 rank = function(object, x, y) {  
 vimp <- varImp(object)  
 vimp <- vimp[order(vimp$Overall,decreasing = TRUE),,drop = FALSE]  
 vimp$var <- rownames(vimp)   
 vimp  
 },  
 selectSize = pickSizeBest,  
 selectVar = pickVars)

### The summary Functions

The summary function takes the observed and predicted values and computes one or more performance metrics (see line 2.14). The input is a data frame with columns obs and pred. The output should be a named vector of numeric variables. Note that the metric argument of the rfe function should reference one of the names of the output of summary. The example function is:

rfRFE$summary

## function (data, lev = NULL, model = NULL)   
## {  
## if (is.character(data$obs))   
## data$obs <- factor(data$obs, levels = lev)  
## postResample(data[, "pred"], data[, "obs"])  
## }  
## <bytecode: 0x0000000019beaff8>  
## <environment: namespace:caret>

wo functions in caret that can be used as the summary funciton are defaultSummary and twoClassSummary (for classification problems with two classes).

### The fit function

This function builds the model based on the current data set (lines 2.3, 2.9 and 2.17). The arguments for the function must be:

* x: the current training set of predictor data with the appropriate subset of variables
* y: the current outcome data (either a numeric or factor vector)
* first: a single logical value for whether the current predictor set has all possible variables (e.g. line 2.3)
* last: similar to first, but TRUE when the last model is fit with the final subset size and predictors. (line 2.17)
* ...: optional arguments to pass to the fit function in the call to rfe

The function should return a model object that can be used to generate predictions. For random forest, the fit function is simple:

rfRFE$fit

## function(x, y, first, last, ...){  
## library(randomForest)  
## randomForest(x, y, importance = first, ...)  
## }

For feature selection without re-ranking at each iteration, the random forest variable importances only need to be computed on the first iterations when all of the predictors are in the model. This can be accomplished using importance`` = first.

### The pred function

This function returns a vector of predictions (numeric or factors) from the current model (lines 2.4 and 2.10). The input arguments must be

* object: the model generated by the fit function
* x: the current set of predictor set for the held-back samples

For random forests, the function is a simple wrapper for the predict function:

rfRFE$pred

## function(object, x) predict(object, x)

For classification, it is probably a good idea to ensure that the resulting factor variables of predictions has the same level as the input data.

### The rank function

This function is used to return the predictors in the order of the most important to the least important (lines 2.5 and 2.11). Inputs are:

* object: the model generated by the fit function
* x: the current set of predictor set for the training samples
* y: the current training outcomes

The function should return a data frame with a column called var that has the current variable names. The first row should be the most important predictor etc. Other columns can be included in the output and will be returned in the final rfe object.

For random forests, the function below uses caret’s varImp function to extract the random forest importances and orders them. For classification, randomForest will produce a column of importances for each class. In this case, the default ranking function orders the predictors by the averages importance across the classes.

rfRFE$rank

## function(object, x, y) {  
## vimp <- varImp(object)  
## vimp <- vimp[order(vimp$Overall,decreasing = TRUE),,drop = FALSE]  
## vimp$var <- rownames(vimp)   
## vimp  
## }

### The selectSize function

This function determines the optimal number of predictors based on the resampling output (line 2.15). Inputs for the function are:

* x: a matrix with columns for the performance metrics and the number of variables, called Variables
* metric: a character string of the performance measure to optimize (e.g. RMSE, Accuracy)
* maximize: a single logical for whether the metric should be maximized This function should return an integer corresponding to the optimal subset size.

caret comes with two examples functions for this purpose: pickSizeBest and pickSizeTolerance. The former simply selects the subset size that has the best value. The latter takes into account the whole profile and tries to pick a subset size that is small without sacrificing too much performance. For example, suppose we have computed the RMSE over a series of variables sizes:

example <- data.frame(RMSE = c(3.215, 2.819, 2.414, 2.144,   
 2.014, 1.997, 2.025, 1.987,   
 1.971, 2.055, 1.935, 1.999,   
 2.047, 2.002, 1.895, 2.018),  
 Variables = 1:16)

These are depicted in the figure below. The solid circle identifies the subset size with the absolute smallest RMSE. However, there are many smaller subsets that produce approximately the same performance but with fewer predictors. In this case, we might be able to accept a slightly larger error for less predictors.

The pickSizeTolerance determines the absolute best value then the percent difference of the other points to this value. In the case of RMSE, this would be

where, is the absolute best error rate. These “tolerance” values are plotted in the buttomo panel. The solid triangle is the smallest subset size that is within 10% of the optimal value.

This approach can produce good results for many of the tree based models, such as random forest, where there is a plateau of good performance for larger subset sizes. For trees, this is usually because unimportant variables are infrequently used in splits and do not significantly affect performance.

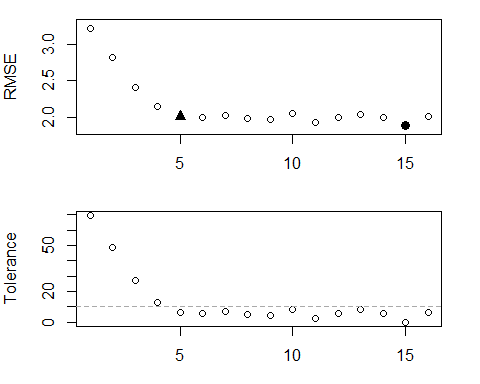
## Find the row with the absolute smallest RMSE  
smallest <- pickSizeBest(example, metric = "RMSE", maximize = FALSE)  
smallest

## [1] 15

## Now one that is within 10% of the smallest  
within10Pct <- pickSizeTolerance(example, metric = "RMSE", tol = 10, maximize = FALSE)  
within10Pct

## [1] 5

minRMSE <- min(example$RMSE)  
example$Tolerance <- (example$RMSE - minRMSE)/minRMSE \* 100   
  
## Plot the profile and the subsets selected using the   
## two different criteria  
  
par(mfrow = c(2, 1), mar = c(3, 4, 1, 2))  
  
plot(example$Variables[-c(smallest, within10Pct)],   
 example$RMSE[-c(smallest, within10Pct)],  
 ylim = extendrange(example$RMSE),  
 ylab = "RMSE", xlab = "Variables")  
  
points(example$Variables[smallest],   
 example$RMSE[smallest], pch = 16, cex= 1.3)  
  
points(example$Variables[within10Pct],   
 example$RMSE[within10Pct], pch = 17, cex= 1.3)  
   
with(example, plot(Variables, Tolerance))  
abline(h = 10, lty = 2, col = "darkgrey")



### The selectVar function

After the optimal subset size is determined, this function will be used to calculate the best rankings for each variable across all the resampling iterations (line 2.16). Inputs for the function are:

* y: a list of variables importance for each resampling iteration and each subset size (generated by the user-defined rank function). In the example, each each of the cross-validation groups the output of the rank function is saved for each of the 10 subset sizes (including the original subset). If the rankings are not recomputed at each iteration, the values will be the same within each cross-validation iteration.
* size: the integer returned by the selectSize function

This function should return a character string of predictor names (of length size) in the order of most important to least important

For random forests, only the first importance calculation (line 2.5) is used since these are the rankings on the full set of predictors. These importances are averaged and the top predictors are returned.

rfRFE$selectVar

## function (y, size)   
## {  
## finalImp <- ddply(y[, c("Overall", "var")], .(var), function(x) mean(x$Overall,   
## na.rm = TRUE))  
## names(finalImp)[2] <- "Overall"  
## finalImp <- finalImp[order(finalImp$Overall, decreasing = TRUE),   
## ]  
## as.character(finalImp$var[1:size])  
## }  
## <bytecode: 0x00000000200290c8>  
## <environment: namespace:caret>

Note that if the predictor rankings are recomputed at each iteration (line 2.11) the user will need to write their own selection function to use the other ranks.

## The example

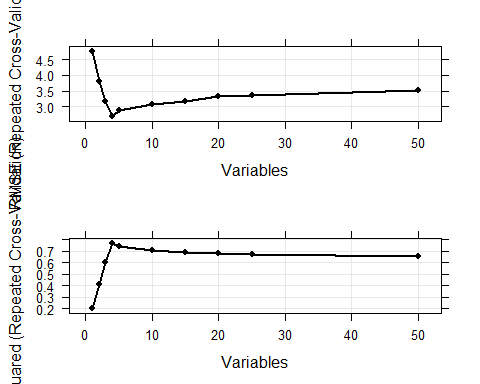
For random forest, we fit the same series of model sizes as the linear model. The option to save all the resampling results across subset sizes was changed for this model and are used to show the lattice plot function capabilities in the figures below.

ctrl$functions <- rfRFE  
ctrl$returnResamp <- "all"  
set.seed(10)  
rfProfile <- rfe(x, y, sizes = subsets, rfeControl = ctrl)  
rfProfile

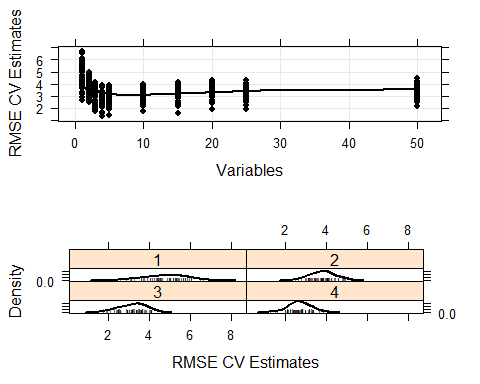
##   
## Recursive feature selection  
##   
## Outer resampling method: Cross-Validated (10 fold, repeated 5 times)   
##   
## Resampling performance over subset size:  
##   
## Variables RMSE Rsquared MAE RMSESD RsquaredSD MAESD Selected  
## 1 4.770 0.2016 4.038 0.9765 0.1806 0.9226   
## 2 3.815 0.4077 3.232 0.5846 0.2102 0.6231   
## 3 3.173 0.6046 2.683 0.6136 0.1907 0.5824   
## 4 2.684 0.7697 2.286 0.5260 0.1141 0.5046 \*  
## 5 2.888 0.7439 2.437 0.5703 0.1244 0.5400   
## 10 3.082 0.7100 2.605 0.5496 0.1448 0.5496   
## 15 3.185 0.6927 2.703 0.5762 0.1725 0.5596   
## 20 3.324 0.6829 2.816 0.5689 0.1686 0.5507   
## 25 3.361 0.6756 2.853 0.5670 0.1739 0.5598   
## 50 3.513 0.6546 3.020 0.5436 0.1865 0.5148   
##   
## The top 4 variables (out of 4):  
## real4, real5, real2, real1

The resampling profile can be visualized along with plots of the individual resampling results:

trellis.par.set(caretTheme())  
plot1 <- plot(rfProfile, type = c("g", "o"))  
plot2 <- plot(rfProfile, type = c("g", "o"), metric = "Rsquared")  
print(plot1, split=c(1,1,1,2), more=TRUE)  
print(plot2, split=c(1,2,1,2))



plot1 <- xyplot(rfProfile,   
 type = c("g", "p", "smooth"),   
 ylab = "RMSE CV Estimates")  
plot2 <- densityplot(rfProfile,   
 subset = Variables < 5,   
 adjust = 1.25,   
 as.table = TRUE,   
 xlab = "RMSE CV Estimates",   
 pch = "|")  
print(plot1, split=c(1,1,1,2), more=TRUE)  
print(plot2, split=c(1,2,1,2))



# Feature selection using genetic algorithms

## Genetic algorithm

enetic algorithms (GAs) mimic Darwinian forces of natural selection to find optimal values of some function (Mitchell, 1998). An initial set of candidate solutions are created and their corresponding fitness values are calculated (where larger values are better). This set of solutions is referred to as a population and each solution as an individual. The individuals with the best fitness values are combined randomly to produce offsprings which make up the next population. To do so, *individual* are selected and undergo cross-over (mimicking genetic reproduction) and also are subject to random mutations. This process is repeated again and again and many generations are produced (i.e. iterations of the search procedure) that should create better and better solutions.

For feature selection, the individuals are subsets of predictors that are encoded as binary; a feature is either included or not in the subset. The fitness values are some measure of model performance, such as the RMSE or classification accuracy. One issue with using GAs for feature selection is that the optimization process can be very aggressive and their is potential for the GA to overfit to the predictors (much like the previous discussion for RFE).

## Internal and external performance estimates

The genetic algorithm code in caret conducts the search of the feature space repeatedly within resampling iterations. First, the training data are split be whatever resampling method was specified in the control function. For example, if 10-fold cross-validation is selected, the entire genetic algorithm is conducted 10 separate times. For the first fold, nine tenths of the data are used in the search while the remaining tenth is used to estimate the external performance since these data points were not used in the search.

During the genetic algorithm, a measure of fitness is needed to guide the search. This is the internal measure of performance. During the search, the data that are available are the instances selected by the top-level resampling (e.g. the nine tenths mentioned above). A common approach is to conduct another resampling procedure. Another option is to use a holdout set of samples to determine the internal estimate of performance (see the holdout argument of the control function). While this is faster, it is more likely to cause overfitting of the features and should only be used when a large amount of training data are available. Yet another idea is to use a penalized metric (such as the AIC statistic) but this may not exist for some metrics (e.g. the area under the ROC curve).

The internal estimates of performance will eventually overfit the subsets to the data. However, since the external estimate is not used by the search, it is able to make better assessments of overfitting. After resampling, this function determines the optimal number of generations for the GA.

Finally, the entire data set is used in the last execution of the genetic algorithm search and the final model is built on the predictor subset that is associated with the optimal number of generations determined by resampling (although the update function can be used to manually set the number of generations).

## Basic syntax

The most basic usage of the function is:

obj <- gafs(x=predictors,  
 y=outcome,  
 iters=100)

where - x: a data frame or matrix of predictors - y: a factor or numeric vector of outcomes - iter: the number of generations for the GA

This isn’t very specific. All of the action is in the control function. That can be used to specify the model to be fit, how predictions are made and summarized as well as the the genetic operations.

Suppose that we want to fit a linear regression model. To do this, we can use train as an interface and pass arguments to that function through gafs:

ctrl <- gafsControl(functions = caretGA)  
obj <- gafs(x = predictors,   
 y = outcome,  
 iters = 100,  
 gafsControl = ctrl,  
 ## Now pass options to `train`  
   
 method = "lm")

Other options, such as preProcess can be passed in as well.

Some important options to gafsControl are: - method, number, repeats, index, indexOut, etc: options similar to those for train top control resampling. - metric: this is similar to train’s option but, in this case, the value should be a named vector with values for the internal and external metrics. If none are specified, the first value returned by the summary functions (see details below) are used and a warning is issued. A similar two-element vector for the option maximize is also required. See [the last example](http://topepo.github.io/caret/feature-selection-using-genetic-algorithms.html#example2) here for an illustration. - holdout: this is a number between [0, 1) that can be used to hold out samples for computing the internal fitness value. Note that this is independent of the external resampling step. Suppose 10-fold CV is being used. Within a resampling iteration, holdout can be used to sample an additional proportion of the 90% resampled data to use for estimating fitness. This may not be a good idea unless you have a very large training set and want to avoid an internal resampling procedure to estimate fitness. - allowParallel and genParallel: these are logicals to control where parallel processing should be used (if at all). The former will parallelize the external resampling while the latter parallelizes the fitness calculations within a generation. allowParallel will almost always be more advantageous.

There are a few built-in sets of functions to use with gafs: caretGA, rfGA and treebagGA. The first is a simple interface to train. When using this, as shown above, arguments can be passed to train using the ... structure and the resampling estimates of performance can be used as the internal fitness value. The functions provided by rfGA and treebagGA avoid using train and their internal estimates of fitness come from using the out-of-bag estimates generated from the model.

The GA implementation in caret uses the underlying code from the GA package ([Scrucca, 2013][<https://www.jstatsoft.org/article/view/v053i04>].

## Example

Using example from the previous page, where there are five real predictors and 40 noise predictors.

library(mlbench)  
library(caret)  
  
n <- 100  
p <- 40  
sigma <- 1  
set.seed(1)  
  
sim <- mlbench.friedman1(n,sd=sigma)  
colnames(sim$x) <- c(paste("real", 1:5, sep = ""),  
 paste("bogus", 1:5, sep = ""))  
bogus <- matrix(rnorm(n \* p), nrow = n)  
colnames(bogus) <- paste("bogus", 5+(1:ncol(bogus)), sep = "")  
x <- cbind(sim$x, bogus)  
y <- sim$y  
normalization <- preProcess(x)  
x <- predict(normalization, x)  
x <- as.data.frame(x)

We’ll fit a random forest model and use the out-of-bag RMSE estimate as the internal performance metric and use the same repeated 10-fold cross-validation process used with the search. To do this, we’ll use the built-in rfGA object for this purpose. The default GA operators will be used and conduct 200 generations of the algorithm.

ga\_ctrl <- gafsControl(functions = rfGA,  
 method = "repeatedcv",  
 repeats = 5)  
  
## Use the same random number seed as the RFE process  
## so that the same CV folds are used for the external  
## resampling.   
set.seed(10)  
rf\_ga <- gafs(x = x, y = y,  
 iters = 200,  
 gafsControl = ga\_ctrl)  
rf\_ga

With 5 repeats of 10-fold cross-validation, the GA was executed 50 times. The average external performance is calculated across resamples and these results are used to determine the optimal number of iterations for the final GA to avoid over-fitting. Across the resamples, an average of 8.7 predictors were selected at the end of each of the algorithms.

The plot function is used to monitor the average of the internal out-of-bag RMSE estimates as well as the average of the external performance estimates calculated from the 50 out-of-sample predictions. By default, this function uses ggplot2 package. A black and white theme can be “added” to the output object:

plot(rf\_ga)+theme\_by()

Based on these results, the generation associated with the best external RMSE estimate was 2.83.

Using the entire training set, the final GA is conducted and, at generation 162, there were 12 that were selected: real1, real2, real3, real4, real5, bogus6, bogus7, bogus12, bogus14, bogus17, bogus20, bogus43. The random forest model with these predictors is created using the entire training set is trained and this is the model that is used when predict.gafs is executed.

## Customizing the search

### The fit function

This function builds the model based on a proposed current subset. The arguments for the function must be - x: the current training set of predictor data with the approporate subset of variables - y: the current outcome data (either a numeric or factor vector) - lev: a character vector with the class levels (or NULL for regression problems) - last:a logical that is TRUE when the final GA search is conducted on the entire data set - ...: optional arguments to pass to the fit function in the call to gafs

The function should return a model object that can be used to generate predictions. For random forest, the fit function is simple:

rfGA$fit

## function (x, y, lev = NULL, last = FALSE, ...)   
## {  
## loadNamespace("randomForest")  
## randomForest::randomForest(x, y, ...)  
## }  
## <bytecode: 0x000000002e2da1d8>  
## <environment: namespace:caret>

## The pred function

This function returns a vector of predictions (numeric or factors) from the current model . The input arguments must be

* object: the model generated by the fit function
* x: the current set of predictor set for the held-back samples

For random forests, the function is a simple wrapper for the predict function:

rfGA$pred

## function (object, x)   
## {  
## tmp <- predict(object, x)  
## if (is.factor(object$y)) {  
## out <- cbind(data.frame(pred = tmp), as.data.frame(predict(object,   
## x, type = "prob")))  
## }  
## else out <- tmp  
## out  
## }  
## <bytecode: 0x000000002e2d9d40>  
## <environment: namespace:caret>

For classification, it is probably a good idea to ensure that the resulting factor variables of predictions has the same levels as the input data.

### The fitness\_intern function

The fitness\_intern function takes the fitted model and computes one or more performance metrics. The inputs to this function are: - object: the model generated by the fit function - x: the current set of predictor set. If the option gafsControl$holdout is zero, these values will be from the current resample (i.e. the same data used to fit the model). Otherwise, the predictor values are from the hold-out set created by gafsControl$holdout. - y: outcome values. See the note for the x argument to understand which data are presented to the function. - maximize: a logical from gafsControl that indicates whether the metric should be maximized or minimized - p: the total number of possible predictors

The output should be a **named** numeric vector of performance values.

In many cases, some resampled measure of performance is used. In the example above using random forest, the OOB error was used. In other cases, the resampled performance from train can be used and if gafsControl$holdout is not zero, a static hold-out set can be used. This depends on the data and problem at hand.

The example function for random forest is:

rfGA$fitness\_intern

## function (object, x, y, maximize, p)   
## rfStats(object)  
## <bytecode: 0x000000002e2d8f40>  
## <environment: namespace:caret>

### The fitness\_extern function

The fitness\_extern function takes the observed and predicted values form the external resampling process and computes one or more performance metrics. The input arguments are:

* data: a data frame or predictions generated by the fit function. For regression, the predicted values in a column called pred. For classification, pred is a factor vector. Class probabilities are usually attached as columns whose names are the class levels (see the random forest example for the fit function above)
* lev: a character vector with the class levels (or NULL for regression problems)

The output should be a **named** numeric vector of performance values.

The example function for random forest is:

rfGA$fitness\_extern

## function (data, lev = NULL, model = NULL)   
## {  
## if (is.character(data$obs))   
## data$obs <- factor(data$obs, levels = lev)  
## postResample(data[, "pred"], data[, "obs"])  
## }  
## <bytecode: 0x000000002f340640>  
## <environment: namespace:caret>

Two functions in caret that can be used as the summary function are defaultSummary and twoClassSummary (for classification problems with two classes).

### The initial Function

This function creates an initial generation. Inputs are:

* vars: the number of possible predictors
* popSize: the population size for each generation
* ...: not currently used

The output should be a binary 0/1 matrix where there are vars columns corresponding to the predictors and popSize rows for the individuals in the population.

The default function populates the rows randomly with subset sizes varying between 10% and 90% of number of possible predictors. For example:

set.seed(128)  
starting <- rfGA$initial(vars=12, popSize=8)  
starting

## [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]  
## [1,] 0 1 0 0 1 0 0 0 0 0 0 0  
## [2,] 0 0 0 0 0 0 1 0 0 0 1 0  
## [3,] 0 0 1 1 1 0 0 0 1 0 1 0  
## [4,] 0 1 0 0 1 0 0 1 0 1 0 1  
## [5,] 0 1 1 1 0 0 1 0 0 1 0 0  
## [6,] 1 1 1 1 1 1 0 1 1 1 1 1  
## [7,] 1 1 1 1 1 0 0 1 0 1 1 1  
## [8,] 1 0 1 1 0 1 1 1 0 1 1 1

apply(starting, 1, mean)

## [1] 0.1666667 0.1666667 0.4166667 0.4166667 0.4166667 0.9166667 0.7500000  
## [8] 0.7500000

gafs has an argument called suggestions that is similar to the one in the ga function where the initial population can be seeded with specific subsets.

### The selection function

This function conducts the genetic selection. Inputs are: - population: the indicators for the current population - fitness: the corresponding fitness values for the population. Note that if the internal performance value is to be minimized, these are the negatives of the actual values - r, q: tuning parameters for specific selection functions. See gafs\_lrSelection and gafs\_nlrSelection - ...: not currently used

The output should be a list with named elements.

* population: the indicators for the selected individuals
* fitness: the fitness values for the selected individuals

The default function is a version of the GA package’s ga\_lrSelection function.

### The crossover function

This function conducts the genetic crossover. Inputs are:

* population: the indicators for the current population
* fitness: the corresponding fitness values for the population. Note that if the internal performance value is to be minimized, these are the negatives of the actual values
* parents: a matrix with two rows containing indicators for the parent individuals.
* ...: not currently used

The default function is a version of the GA package’s ga\_spCrossover function. Another function that is a version of that package’s uniform cross-over function is also available.

* . The output should be a list with named elements.
* children: from ?ga\_spCrossover: “a matrix of dimension 2 times the number of decision variables containing the generated offsprings”"
* fitness: “a vector of length 2 containing the fitness values for the offsprings. A value NA is returned if an offspring is different (which is usually the case) from the two parents.”"

### The mutation function

his function conducts the genetic mutation. Inputs are:

* population: the indicators for the current population
* parents: a vector of indices for where the mutation should occur.
* ...: not currently used

The default function is a version of the GA package’s gabin\_raMutation function.

* . The output should the mutated population.

### The selectIter function

This function determines the optimal number of generations based on the resampling output. Inputs for the function are:

* x: a matrix with columns for the performance metrics averaged over resamples
* metric: a character string of the performance measure to optimize (e.g. RMSE, Accuracy)
* maximize: a single logical for whether the metric should be maximized This function should return an integer corresponding to the optimal subset size.

## The example revisited

The previous GA included some of the non-informative predictors. We can cheat a little and try to bias the search to get the right solution.

We can try to encourage the algorithm to choose a fewer predictors, we can peneralize the RMSE estimate. Normally, a metric like the AIC statistic would be used.

However, with a random forest model, there is no real notion of model degrees of freedom. As an alternative, we can use [disirability functions](https://scholar.google.com/scholar?q=%22desirability+functions) to pneralize the RMSE. To do this, two functions are created that translate the number of predictors and the RMSE values to a measure of “desirability”. For the number of predictors, the most desirable property would be a single predictor and the worst situation would be if the model required all 50 predictors. That desirability function is visualized as;

For the RMSE, the est case would be zero. Many poor models have values around four. To five the RMSE value more weight in the overall desirability calculation, we use a scale parameter value of 2. This desirability function is:

To use the overall desirability to drive the feature selection, the internal function requires replacement. We make a copy of rfGA and add code using the desirability package and the function returns the estimated RMSE and the overall desirability. The gafsControl function also need changes. The metric argument needs to reflect that the overall desirability score should be maximized internally but the RMSE estimate should be minimized externally.

# install.packages("desirability")  
library(desirability)  
rfGA2 <- rfGA  
rfGA2$fitness\_intern <- function (object, x, y, maximize, p) {  
 RMSE <- rfStats(object)[1]  
 d\_RMSE <- dMin(0, 4)  
 d\_Size <- dMin(1, p, 2)  
 overall <- dOverall(d\_RMSE, d\_Size)  
 D <- predict(overall, data.frame(RMSE, ncol(x)))  
 c(D = D, RMSE = as.vector(RMSE))  
 }  
ga\_ctrl\_d <- gafsControl(functions = rfGA2,  
 method = "repeatedcv",  
 repeats = 5,  
 metric = c(internal = "D", external = "RMSE"),  
 maximize = c(internal = TRUE, external = FALSE))  
  
set.seed(10)  
rf\_ga\_d <- gafs(x = x, y = y,  
 iters = 150,  
 gafsControl = ga\_ctrl\_d)  
  
rf\_ga\_d

Here are the RMSE values for this search:

plot(rf\_ga\_d)+theme\_bw()

The final GA found 4 that were selected: real1, real2, real4, real5. During resampling, the average number of predictors selected was 5.1, indicating that the penalty on the number of predictors was effective.

# Feature selection using simulated annealing

## Simulated annealing

Simulated annealing (SA) is a global search method that makes small random changes (i.e. perturbations) to an initial candidate solution. If the performance value for the perturbed value is better than the previous solution, the new solution is accepted. If not, an acceptance probability is determined based on the difference between the two performance values and the current iteration of the search. From this, a sub-optimal solution can be accepted on the off-change that it may eventually produce a better solution in subsequent iterations. See Kirkpatrick (1984) or Rutenbar (1989) for better descriptions.

In the context of feature selection, a solution is a binary vector that describes the current subset. The subset is perturbed by randomly changing a small number of members in the subset.

## Internal and external performance estimates

Much of the discussion on this subject in the [genetic algorithm page](http://topepo.github.io/caret/feature-selection-using-genetic-algorithms.html#performance) is relevant here, although SA search is less aggressive than GA search. In any case, the implementation here conducts the SA search inside the resampling loops and uses an external performance estimate to choose how many iterations of the search are appropriate.

## Basic syntax

The syntax of this function is very similar to the previous information for genetic algorithm searches. The most basic usage of the function is:

obj <- safs(x=predictors,  
 y=outcome,  
 iters=100)

where, - x: a data fame or matrix of predictor values - y: a factor or numeric vector of outcomes - iters: the number of iterations for the SA

This isn’t very specific. All of the action is in the control function. That can be used to specify the model to be fit, how predictions are made and summarized as well as the genetic operations.

Suppose that we want to fit a linear regression model. To do this, we can use train as an interface and pass arguments to that function throughsafs:

ctrl <- safsControl(functions=caretSA)  
obj <- safs(x=predictors,  
 y=outcome,  
 iters=100,  
 safsControl = ctrl,  
 ## Now pass options to `train`  
 method="lm")

Other options, such as preProcess, can be passed in as well.

Some important options to safsControl are: - method, number, repeats, index, indexOut, etc: options similar to those for train top control resampling.

* metric: this is similar to train’s option but, in this case, the value should be a named vector with values for the internal and external metrics. If none are specified, the first value returned by the summary functions (see details below) are used and a warning is issued. A similar two-element vector for the option maximize is also required. See the last example here for an illustration.
* holdout: this is a number between [0, 1) that can be used to hold out samples for computing the internal fitness value. Note that this is independent of the external resampling step. Suppose 10-fold CV is being used. Within a resampling iteration, holdout can be used to sample an additional proportion of the 90% resampled data to use for estimating fitness. This may not be a good idea unless you have a very large training set and want to avoid an internal resampling procedure to estimate fitness.
* improve: an integer (or infinity) defining how many iterations should pass without an improvement in fitness before the current subset is reset to the last known improvement.
* allowParallel: should the external resampling loop be run in parallel?.

There are a few built-in sets of functions to use with safs: caretSA, rfSA, and treebagSA. The first is a simple interface to train. When using this, as shown above, arguments can be passed to train using the … structure and the resampling estimates of performance can be used as the internal fitness value. The functions provided by rfSA and treebagSA avoid using train and their internal estimates of fitness come from using the out-of-bag estimates generated from the model.

## Example

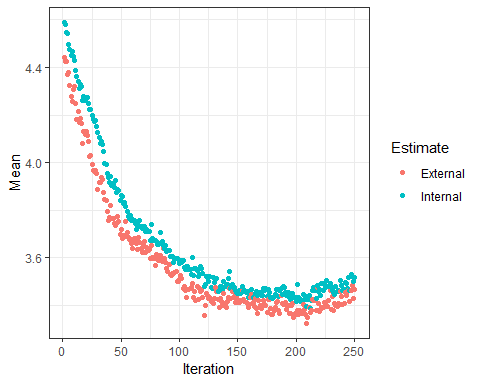
Using the example from the [previous page](http://topepo.github.io/caret/recursive-feature-elimination.html#example) where there are five real predictors and 40 noise predictors.

We will fit a randome forest model and use the out-of-bag RMSE estimate as the internal performance metric and use the same repeated 10-fold cross-validation process used with the search. To do this, we’ll use the built-in rfSA object for this purpose. The default SA operations will be used with 1000 iterations of the algorithm.

sa\_ctrl <- safsControl(functions = rfSA,  
 method="repeatedcv",  
 repeats=5,  
 improve=50)  
  
set.seed(10)  
rf\_sa <- safs(x=x, y=y,  
 iters=250,  
 safsControl=sa\_ctrl)

As with the GA, we can plot the internal and external performance over iterations.

plot(rf\_sa)+theme\_bw()



The performance here isn’t as good as the previous GA or RFE solutions. Based on these results, the iteration associated with the best external RMSE estimate was 216 with a corresponding RMSE estimate of 3.36.

Using the entire training set, the final SA is conducted and, at iteration 216, there were 18 selected: real1, real3, real4, real5, bogus6, bogus7, bogus9, bogus10, bogus12, bogus13, bogus18, bogus28, bogus30, bogus36, bogus39, bogus40, bogus43, bogus44. The random forest model with these predictors is created using the entire training set is trained and this is the model that is used when predict.safs is executed.

## Customizing the search

### the fit function

This function builds the model based on a proposed current subset. The arguments for the function must be:

* x: the current training set of predictor data with the appropriate subset of variables
* y: the current outcome data (either a numeric or factor vector)
* lev: a character vector with the class levels (or NULL for regression problems)
* last: a logical that is TRUE when the final SA search is conducted on the entire data set
* ...: optional arguments to pass to the fit function in the call to safs The function should return a model object that can be used to generate predictions. For random forest, the fit function is simple:

rrSA$fit

### The pred function

This function returns a vector of predictions (numeric or factors) from the current model. The input arguments must be

* object: the model generated by the fit function
* x: the current set of predictor set for the held-back samples

For random forests, the function is a simple wrapper for the predict function:

For classification, it is probably a good idea to ensure that the resulting factor variables of predictions has the same levels as the input data.

### The fitness\_intern function Thefitness\_intern` function takes the fitted model and computes one or more performance metrics. The inputs to this function are:

* object: the model generated by the fit function
* x: the current set of predictor set. If the option safsControl$holdout is zero, these values will be from the current resample (i.e. the same data used to fit the model). Otherwise, the predictor values are from the hold-out set created by safsControl$holdout.
* y: outcome values. See the note for the x argument to understand which data are presented to the function.
* maximize: a logical from safsControl that indicates whether the metric should be maximized or minimized
* p: the total number of possible predictors

The output should be a **named** numeric vector of performance values.

In many cases, some resampled measure of performance is used. In the example above using random forest, the OOB error was used. In other cases, the resampled performance from train can be used and, if safsControl$holdout is not zero, a static hold-out set can be used. This depends on the data and problem at hand. If left

The example function for random forest is:`

rfSA$fitness\_intern()

### The fitness\_extern function

The fitness\_extern function takes the observed and predicted values form the external resampling process and computes one or more performance metrics. The input arguments are:

* data: a data frame or predictions generated by the fit function. For regression, the predicted values in a column called pred. For classification, pred is a factor vector. Class probabilities are usually attached as columns whose names are the class levels (see the random forest example for the fit function above)
* lev: a character vector with the class levels (or NULL for regression problems) The output should be a **named** numeric vector of performance values.

The example function for random forest is:

rfSA$fitness\_extern

Two functions in caret that can be used as the summary function are defaultSummary and twoClassSummary (for classification problems with two classes).

### The initial function

This function creates an initial subset. Inputs are:

* vars: the number of possible predictors
* prob: the probability that a feature is in the subset
* ...: not currently used

The output should be a vector of integers indicating which predictors are in the initial subset.

Alternatively, instead of a function, a vector of integers can be used in this slot.

### The perturb function

This function perturbs the subset. Inputs are:

* x: the integers defining the current subset
* vars: the number of possible predictors
* number: the number of predictors to randomly change
* ...: not currently used The output should be a vector of integers indicating which predictors are in the new subset.

### The prob function

This function computes the acceptance probability. Inputs are:

* old: the fitness value for the current subset
* new: the fitness value for the new subset
* iteration: the current iteration number or, if the improveargument of safsControl is used, the number of iterations since the last restart
* ...: not currently used The output should be a numeric value between zero and one.

One of the biggest difficulties in using simulated annealing is the specification of the acceptance probability calculation. There are many references on different methods for doing this but the general consensus is that 1) the probability should decrease as the difference between the current and new solution increases and 2) the probability should decrease over iterations. One issue is that the difference in fitness values can be scale-dependent. In this package, the default probability calculations uses the percent difference, i.e. (current - new)/current to normalize the difference. The basic form of the probability simply takes the difference, multiplies by the iteration number and exponentiates this product:

prob=exp[(current - new)/current\*iteration]

To demonstrate this, the plot below shows the probability profile for different fitness values of the current subset and different (absolute) differences. For the example data that were simulated, the RMSE values ranged between values greater than 4 to just under 3. In the plot below, the red curve in the right-hand panel shows how the probability changes over time when comparing a current value of 4 with a new values of 4.5 (smaller values being better). While this difference would likely be accepted in the first few iterations, it is unlikely to be accepted after 30 or 40. Also, larger differences are uniformly disfavored relative to smaller differences.

grid <- expand.grid(old = c(4, 3.5),  
 new = c(4.5, 4, 3.5) + 1,  
 iter = 1:40)  
grid <- subset(grid, old < new)  
  
grid$prob <- apply(grid, 1,   
 function(x)   
 safs\_prob(new = x["new"],   
 old= x["old"],   
 iteration = x["iter"]))  
  
grid$Difference <- factor(grid$new - grid$old)  
grid$Group <- factor(paste("Current Value", grid$old))  
  
ggplot(grid, aes(x = iter, y = prob, color = Difference)) +   
 geom\_line() + facet\_wrap(~Group) + theme\_bw() +  
 ylab("Probability") + xlab("Iteration")

# Data sets

There are a few data sets included in caret. The first four are computational chemistry problems where the object is to relate the molecular structure of compounds (via molecular descriptors) to some property of interest (Clark and Pickett (2000)). Similar data sets can be found in the QSARdata R pacakge.

Other R packages with data are:

* mlbench,
* SMCRM and
* AppliedPredictiveModeling.

## Blood-Brain Barrier Data

Mente and Lombardo (2005) developed models to predict the log of the ratio of the concentration of a compound in the brain and the concentration in blood. For each compound, they computed three sets of molecular descriptors: MOE 2D, rule-of-five and Charge Polar Surface Area (CPSA). In all, 134 descriptors were calculated. Included in this package are 208 non-proprietary literature compounds. The vector logBBB contains the log concentration ratio and the data fame bbbDescr contains the descriptor values.

## COX-2 Activity Data

From Sutherland, O’Brien, and Weaver (2003): A set of 467 cyclooxygenase-2 (COX-2) inhibitors has been assembled from the published work of a single research group, with in vitro activities against human recombinant enzyme expressed as IC50 values ranging from 1 nM to >100 uM (53 compounds have indeterminate IC50 values).

A set of 255 descriptors (MOE2D and QikProp) were generated. To classify the data, we used a cutoff of 2^{2.5} to determine activity.

Using data(cox2) exposes three R objects: cox2Descr is a data frame with the descriptor data, cox2IC50 is a numeric vector of IC50 assay values and cox2Class is a factor vector with the activity results.

## DHFR Inhibition

Sutherland and Weaver (2004) discuss QSAR models for dihydrofolate reductase (DHFR) inhibition. This data set contains values for 325 compounds. For each compound, 228 molecular descriptors have been calculated. Additionally, each samples is designated as “active” or “inactive”.

The data frame dhfr contains a column called Y with the outcome classification. The remainder of the columns are molecular descriptor values.

## Tecator NIR Data

These data can be found in the datasets section of StatLib. The data consist of 100 near infrared absorbance spectra used to predict the moisture, fat and protein values of chopped meat.

From StatLib: > These data are recorded on a Tecator Infratec Food and Feed Analyzer working in the wavelength range 850 - 1050 nm by the Near Infrared Transmission (NIT) principle. Each sample contains finely chopped pure meat with different moisture, fat and protein contents. If results from these data are used in a publication we want you to mention the instrument and company name (Tecator) in the publication. In addition, please send a preprint of your article to: Karin Thente, Tecator AB, Box 70, S-263 21 Hoganas, Sweden.

One reference for these data is Borggaard and Thodberg (1992).

Using data(tecator) loads a 215 x 100 matrix of absorbance spectra and a 215 x 3 matrix of outcomes.

## Fatty Acid Composition Data

Brodnjak-Voncina et al. (2005) describe a set of data where seven fatty acid compositions were used to classify commercial oils as either pumpkin (labeled A), sunflower (B), peanut (C), olive (D), soybean (E), rapeseed (F) and corn (G). There were 96 data points contained in their Table 1 with known results. The breakdown of the classes is given in below:

data(oil)  
dim(fattyAcids)

## [1] 96 7

## [1] 96 7  
table(oilType)

## oilType  
## A B C D E F G   
## 37 26 3 7 11 10 2

## oilType  
## A B C D E F G   
## 37 26 3 7 11 10 2

As a note, the paper states on page 32 that there are 37 unknown samples while the table on pages 33 and 34 shows that there are 34 unknowns.

## German Credit Data

Data from Dr. Hans Hofmann of the University of Hamburg and stored at the UC Irvine Machine Learning Repository.

These data have two classes for the credit worthiness: good or bad. There are predictors related to attributes, such as: checking account status, duration, credit history, purpose of the loan, amount of the loan, savings accounts or bonds, employment duration, Installment rate in percentage of disposable income, personal information, other debtors/guarantors, residence duration, property, age, other installment plans, housing, number of existing credits, job information, Number of people being liable to provide maintenance for, telephone, and foreign worker status.

Many of these predictors are discrete and have been expanded into several 0/1 indicator variables

library(caret)  
data(GermanCredit)  
## Show the first 10 columns  
str(GermanCredit[, 1:10])

## 'data.frame': 1000 obs. of 10 variables:  
## $ Duration : int 6 48 12 42 24 36 24 36 12 30 ...  
## $ Amount : int 1169 5951 2096 7882 4870 9055 2835 6948 3059 5234 ...  
## $ InstallmentRatePercentage: int 4 2 2 2 3 2 3 2 2 4 ...  
## $ ResidenceDuration : int 4 2 3 4 4 4 4 2 4 2 ...  
## $ Age : int 67 22 49 45 53 35 53 35 61 28 ...  
## $ NumberExistingCredits : int 2 1 1 1 2 1 1 1 1 2 ...  
## $ NumberPeopleMaintenance : int 1 1 2 2 2 2 1 1 1 1 ...  
## $ Telephone : num 0 1 1 1 1 0 1 0 1 1 ...  
## $ ForeignWorker : num 1 1 1 1 1 1 1 1 1 1 ...  
## $ Class : Factor w/ 2 levels "Bad","Good": 2 1 2 2 1 2 2 2 2 1 ...

## 'data.frame': 1000 obs. of 10 variables:  
## $ status : Factor w/ 4 levels "... < 100 DM",..: 1 2 4 1 1 4 4 2 4 2 ...  
## $ duration : num 6 48 12 42 24 36 24 36 12 30 ...  
## $ credit\_history : Factor w/ 5 levels "no credits taken/all credits paid back duly",..: 5 3 5 3 4 3 3 3 3 5 ...  
## $ purpose : Factor w/ 10 levels "car (new)","car (used)",..: 5 5 8 4 1 8 4 2 5 1 ...  
## $ amount : num 1169 5951 2096 7882 4870 ...  
## $ savings : Factor w/ 5 levels "... < 100 DM",..: 5 1 1 1 1 5 3 1 4 1 ...  
## $ employment\_duration: Ord.factor w/ 5 levels "unemployed"<"... < 1 year"<..: 5 3 4 4 3 3 5 3 4 1 ...  
## $ installment\_rate : num 4 2 2 2 3 2 3 2 2 4 ...  
## $ personal\_status\_sex: Factor w/ 5 levels "male : divorced/separated",..: 3 2 3 3 3 3 3 3 1 4 ...  
## $ other\_debtors : Factor w/ 3 levels "none","co-applicant",..: 1 1 1 3 1 1 1 1 1 1 ...

## Kelly Blue Book

Resale data for 2005 model year GM cars Kuiper (2008) collected data on Kelly Blue Book resale data for 804 GM cars (2005 model year).

cars is data frame of the suggested retail price (column Price) and various characteristics of each car (columns Mileage, Cylinder, Doors, Cruise, Sound, Leather, Buick, Cadillac, Chevy, Pontiac, Saab, Saturn, convertible, coupe, hatchback, sedan and wagon)

## Cell Body Segmentation Data

Hill, LaPan, Li and Haney (2007) develop models to predict which cells in a high content screen were well segmented. The data consists of 119 imaging measurements on 2019. The original analysis used 1009 for training and 1010 as a test set (see the column called Case).

The outcome class is contained in a factor variable called Class with levels PS for poorly segmented and WS for well segmented.

data("segmentationData")  
str(segmentationData[,1:10])

## 'data.frame': 2019 obs. of 10 variables:  
## $ Cell : int 207827637 207932307 207932463 207932470 207932455 207827656 207827659 207827661 207932479 207932480 ...  
## $ Case : Factor w/ 2 levels "Test","Train": 1 2 2 2 1 1 1 1 1 1 ...  
## $ Class : Factor w/ 2 levels "PS","WS": 1 1 2 1 1 2 2 1 2 2 ...  
## $ AngleCh1 : num 143.25 133.75 106.65 69.15 2.89 ...  
## $ AreaCh1 : int 185 819 431 298 285 172 177 251 495 384 ...  
## $ AvgIntenCh1 : num 15.7 31.9 28 19.5 24.3 ...  
## $ AvgIntenCh2 : num 4.95 206.88 116.32 102.29 112.42 ...  
## $ AvgIntenCh3 : num 9.55 69.92 63.94 28.22 20.47 ...  
## $ AvgIntenCh4 : num 2.21 164.15 106.7 31.03 40.58 ...  
## $ ConvexHullAreaRatioCh1: num 1.12 1.26 1.05 1.2 1.11 ...

## Sacramento house price data

This data frame contains house and sale price data for 932 homes in Sacramento CA. The original data were obtained from the website for the SpatialKey software. From their website: “The Sacramento real estate transactions file is a list of 985 real estate transactions in the Sacramento area reported over a five-day period, as reported by the Sacramento Bee.” Google was used to fill in missing/incorrect data.

data(Sacramento)  
str(Sacramento)

## 'data.frame': 932 obs. of 9 variables:  
## $ city : Factor w/ 37 levels "ANTELOPE","AUBURN",..: 34 34 34 34 34 34 34 34 29 31 ...  
## $ zip : Factor w/ 68 levels "z95603","z95608",..: 64 52 44 44 53 65 66 49 24 25 ...  
## $ beds : int 2 3 2 2 2 3 3 3 2 3 ...  
## $ baths : num 1 1 1 1 1 1 2 1 2 2 ...  
## $ sqft : int 836 1167 796 852 797 1122 1104 1177 941 1146 ...  
## $ type : Factor w/ 3 levels "Condo","Multi\_Family",..: 3 3 3 3 3 1 3 3 1 3 ...  
## $ price : int 59222 68212 68880 69307 81900 89921 90895 91002 94905 98937 ...  
## $ latitude : num 38.6 38.5 38.6 38.6 38.5 ...  
## $ longitude: num -121 -121 -121 -121 -121 ...

## Animal Scat Data

Reid (2105) collected data on animal feses in coastal California. The data consist of DNA verified species designations as well as fields related to the time and place of the collection and the scat itself. The data frame scat\_orig contains while scat contains data on the three main species.

data(scat)  
str(scat)

## 'data.frame': 110 obs. of 19 variables:  
## $ Species : Factor w/ 3 levels "bobcat","coyote",..: 2 2 1 2 2 2 1 1 1 1 ...  
## $ Month : Factor w/ 9 levels "April","August",..: 4 4 4 4 4 4 4 4 4 4 ...  
## $ Year : int 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 ...  
## $ Site : Factor w/ 2 levels "ANNU","YOLA": 2 2 2 2 2 2 1 1 1 1 ...  
## $ Location : Factor w/ 3 levels "edge","middle",..: 1 1 2 2 1 1 3 3 3 2 ...  
## $ Age : int 5 3 3 5 5 5 1 3 5 5 ...  
## $ Number : int 2 2 2 2 4 3 5 7 2 1 ...  
## $ Length : num 9.5 14 9 8.5 8 9 6 5.5 11 20.5 ...  
## $ Diameter : num 25.7 25.4 18.8 18.1 20.7 21.2 15.7 21.9 17.5 18 ...  
## $ Taper : num 41.9 37.1 16.5 24.7 20.1 28.5 8.2 19.3 29.1 21.4 ...  
## $ TI : num 1.63 1.46 0.88 1.36 0.97 1.34 0.52 0.88 1.66 1.19 ...  
## $ Mass : num 15.9 17.6 8.4 7.4 25.4 ...  
## $ d13C : num -26.9 -29.6 -28.7 -20.1 -23.2 ...  
## $ d15N : num 6.94 9.87 8.52 5.79 7.01 8.28 4.2 3.89 7.34 6.06 ...  
## $ CN : num 8.5 11.3 8.1 11.5 10.6 9 5.4 5.6 5.8 7.7 ...  
## $ ropey : int 0 0 1 1 0 1 1 0 0 1 ...  
## $ segmented: int 0 0 1 0 1 0 1 1 1 1 ...  
## $ flat : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ scrape : int 0 0 1 0 0 0 1 0 0 0 ...

# Session information

This documentation was created on Sat May 26 2018 with the following R packages

## ─ Session info ──────────────────────────────────────────────────────────  
## setting value   
## version R version 3.5.0 (2018-04-23)  
## os macOS High Sierra 10.13.4   
## system x86\_64, darwin15.6.0   
## ui X11   
## language (EN)   
## collate en\_US.UTF-8   
## tz America/New\_York   
## date 2018-05-26   
##   
## ─ Packages ──────────────────────────────────────────────────────────────  
## package \* version date   
## abind 1.4-5 2016-07-21  
## acepack 1.4.1 2016-10-29  
## AppliedPredictiveModeling \* 1.1-6 2014-07-25  
## assertthat 0.2.0 2017-04-11  
## backports 1.1.2 2017-12-13  
## base64enc 0.1-3 2015-07-28  
## bindr 0.1.1 2018-03-13  
## bindrcpp \* 0.2.2 2018-03-29  
## bitops 1.0-6 2013-08-17  
## bookdown \* 0.7.11 2018-05-23  
## broom \* 0.4.4 2018-03-29  
## C50 \* 0.1.2 2018-05-22  
## caret \* 6.0-80 2018-05-26  
## caTools \* 1.17.1 2014-09-10  
## checkmate 1.8.5 2017-10-24  
## class 7.3-14 2015-08-30  
## clisymbols 1.2.0 2017-05-21  
## cluster 2.0.7-1 2018-04-09  
## codetools 0.2-15 2016-10-05  
## coin 1.2-2 2017-11-28  
## colorspace 1.3-2 2016-12-14  
## combinat 0.0-8 2012-10-29  
## CORElearn 1.52.1 2018-04-02  
## crosstalk 1.0.0 2016-12-21  
## Cubist 0.2.2 2018-05-21  
## curl 3.2 2018-03-28  
## CVST 0.2-1 2013-12-10  
## data.table 1.11.2 2018-05-08  
## ddalpha 1.3.3 2018-04-30  
## dendextend 1.7.0 2018-02-11  
## DEoptimR 1.0-8 2016-11-19  
## desirability \* 2.1 2016-09-22  
## digest 0.6.15 2018-01-28  
## dimRed 0.1.0.9001 2018-05-24  
## diptest 0.75-7 2016-12-05  
## DMwR \* 0.4.1 2013-08-08  
## doMC \* 1.3.5 2017-12-12  
## doParallel \* 1.0.11 2017-09-28  
## dplyr \* 0.7.5 2018-05-19  
## DRR 0.0.3 2018-01-06  
## DT \* 0.4 2018-01-30  
## e1071 \* 1.6-8 2017-02-02  
## earth \* 4.6.2 2018-03-21  
## ellipse 0.4.1 2018-01-05  
## evaluate 0.10.1 2017-06-24  
## flexmix 2.3-14 2017-04-28  
## foreach \* 1.4.4 2017-12-12  
## foreign 0.8-70 2017-11-28  
## Formula \* 1.2-2 2017-07-10  
## fpc 2.1-11 2018-01-13  
## gam 1.15 2018-02-25  
## gbm \* 2.1.3 2017-03-21  
## gclus 1.3.1 2012-06-25  
## gdata 2.18.0 2017-06-06  
## geometry 0.3-6 2015-09-09  
## ggplot2 \* 2.2.1 2016-12-30  
## ggthemes \* 3.4.2 2018-04-03  
## glue 1.2.0 2017-10-29  
## gower 0.1.2 2017-02-23  
## gplots 3.0.1 2016-03-30  
## gridExtra 2.3 2017-09-09  
## gtable 0.2.0 2016-02-26  
## gtools 3.5.0 2015-05-29  
## heatmaply \* 0.14.1 2018-02-01  
## highr 0.6 2016-05-09  
## Hmisc \* 4.1-1 2018-01-03  
## htmlTable 1.11.2 2018-01-20  
## htmltools 0.3.6 2017-04-28  
## htmlwidgets 1.2 2018-04-19  
## httpuv 1.4.1 2018-04-21  
## httr 1.3.1 2017-08-20  
## igraph 1.2.1 2018-03-10  
## inum 1.0-0 2017-12-12  
## ipred \* 0.9-6 2017-03-01  
## iterators \* 1.0.9 2017-12-12  
## jpeg 0.1-8 2014-01-23  
## jsonlite 1.5 2017-06-01  
## kernlab \* 0.9-25 2016-10-03  
## KernSmooth 2.23-15 2015-06-29  
## klaR \* 0.6-14 2018-03-19  
## knitr \* 1.20 2018-02-20  
## labeling 0.3 2014-08-23  
## later 0.7.1 2018-03-07  
## lattice \* 0.20-35 2017-03-25  
## latticeExtra \* 0.6-28 2016-02-09  
## lava 1.6.1 2018-03-28  
## lazyeval 0.2.1 2017-10-29  
## libcoin 1.0-1 2017-12-13  
## lubridate 1.7.4 2018-04-11  
## magic 1.5-8 2018-01-26  
## magrittr 1.5 2014-11-22  
## MASS \* 7.3-49 2018-02-23  
## Matrix 1.2-14 2018-04-09  
## mboost \* 2.8-1 2017-07-23  
## mclust 5.4 2017-11-22  
## mda 0.4-10 2017-11-02  
## mime 0.5 2016-07-07  
## miniUI 0.1.1 2016-01-15  
## mlbench \* 2.1-1 2012-07-10  
## MLmetrics 1.1.1 2016-05-13  
## mnormt 1.5-5 2016-10-15  
## ModelMetrics 1.1.0 2016-08-26  
## modeltools \* 0.2-21 2013-09-02  
## multcomp 1.4-8 2017-11-08  
## munsell 0.4.3 2016-02-13  
## mvtnorm \* 1.0-7 2018-01-26  
## networkD3 \* 0.4 2017-03-18  
## nlme \* 3.1-137 2018-04-07  
## nnet 7.3-12 2016-02-02  
## nnls 1.4 2012-03-19  
## party \* 1.3-0 2018-04-20  
## partykit 1.2-1 2018-04-20  
## pillar 1.2.1 2018-02-27  
## pkgconfig 2.0.1 2017-03-21  
## plotly \* 4.7.1 2017-07-29  
## plotmo \* 3.3.6 2018-03-21  
## plotrix \* 3.7 2017-12-07  
## pls \* 2.6-0 2016-12-18  
## plyr \* 1.8.4 2016-06-08  
## prabclus 2.2-6 2015-01-14  
## pROC \* 1.11.0 2018-03-25  
## prodlim 2018.04.18 2018-04-18  
## promises 1.0.1 2018-04-13  
## proxy \* 0.4-22 2018-04-08  
## psych 1.8.3.3 2018-03-30  
## purrr 0.2.4 2017-10-18  
## QSARdata \* 1.3 2013-07-16  
## quadprog 1.5-5 2013-04-17  
## quantmod 0.4-13 2018-04-13  
## questionr 0.6.2 2017-11-01  
## R6 2.2.2 2017-06-17  
## randomForest \* 4.6-14 2018-03-25  
## RColorBrewer \* 1.1-2 2014-12-07  
## Rcpp 0.12.17 2018-05-18  
## RcppRoll 0.2.2 2015-04-05  
## recipes \* 0.1.2.9001 2018-05-24  
## registry 0.5 2017-12-03  
## reshape2 \* 1.4.3 2017-12-11  
## rlang 0.2.0 2018-02-20  
## rmarkdown 1.9 2018-03-01  
## robustbase 0.93-0 2018-04-24  
## ROCR 1.0-7 2015-03-26  
## ROSE \* 0.0-3 2014-07-15  
## rpart 4.1-13 2018-02-23  
## rprojroot 1.3-2 2018-01-03  
## rstudioapi 0.7 2017-09-07  
## sandwich \* 2.4-0 2017-07-26  
## scales 0.5.0.9000 2018-05-09  
## seriation 1.2-3 2018-02-05  
## sessioninfo \* 1.0.0 2017-06-21  
## sfsmisc 1.1-2 2018-03-05  
## shiny 1.0.5 2017-08-23  
## stabs \* 0.6-3 2017-07-19  
## stringi 1.2.2 2018-05-02  
## stringr 1.3.0 2018-02-19  
## strucchange \* 1.5-1 2015-06-06  
## survival \* 2.42-3 2018-04-16  
## TeachingDemos \* 2.10 2016-02-12  
## TH.data 1.0-8 2017-01-23  
## tibble 1.4.2 2018-01-22  
## tidyr 0.8.1 2018-05-18  
## tidyselect 0.2.4 2018-02-26  
## timeDate 3043.102 2018-02-21  
## trimcluster 0.1-2 2012-10-29  
## TSP 1.1-5 2017-02-22  
## TTR 0.23-3 2018-01-24  
## viridis \* 0.5.1 2018-03-29  
## viridisLite \* 0.3.0 2018-02-01  
## webshot 0.5.0 2017-11-29  
## whisker 0.3-2 2013-04-28  
## withr 2.1.2 2018-03-15  
## xfun 0.1 2018-01-22  
## xtable 1.8-2 2016-02-05  
## xts 0.10-2 2018-03-14  
## yaml 2.1.19 2018-05-01  
## zoo \* 1.8-1 2018-01-08