Caret introudction

Koji Mizumura

September 6,2018

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.

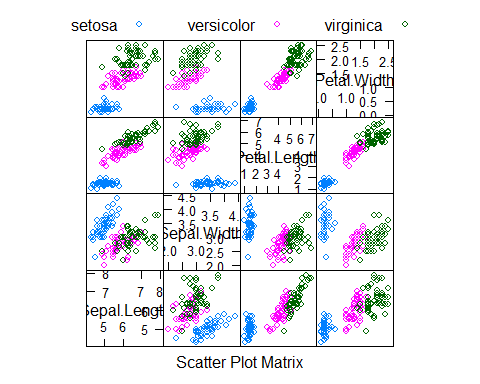
For classification data sets, the iris data are used for illusion.

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 ...

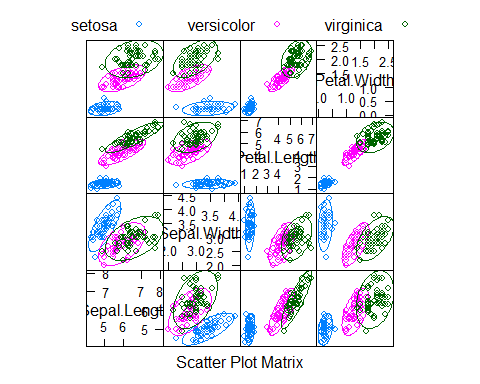
## Scatterplot matrix

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



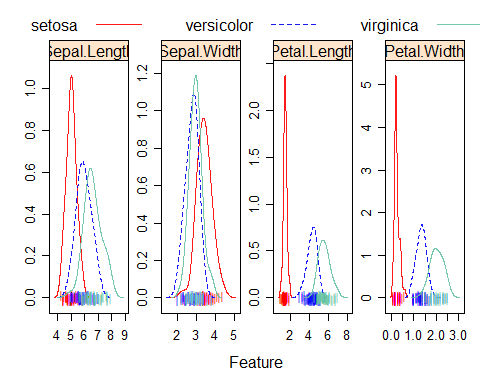
## Scatterplot matrix with ellipses

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



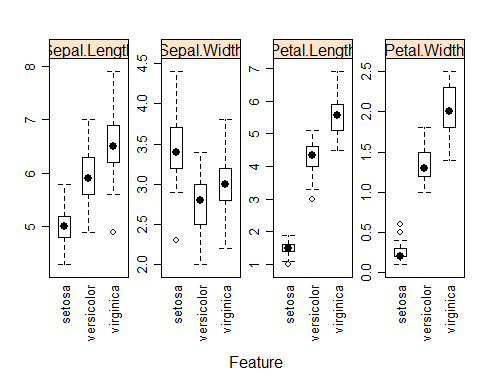
## Overlayed density plots

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)

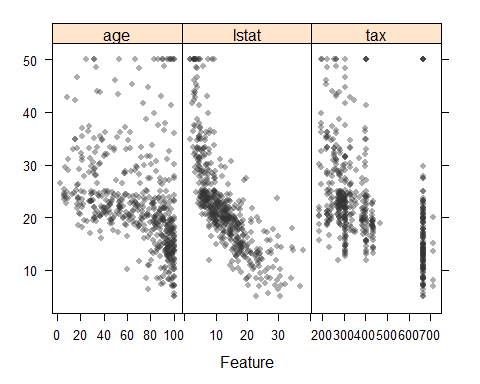
## Warning: package 'mlbench' was built under R version 3.5.1

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 ...

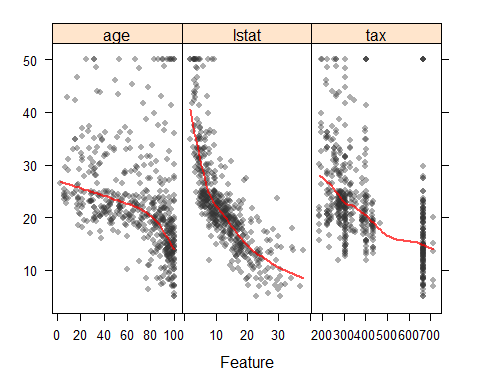
When the predictors are continuous, featurePlot can be used to create scatter plots of each of the predictors with the outcome. For example:

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

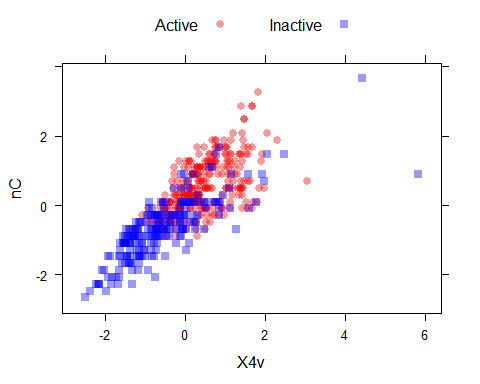
In some cases, there is a need to use **principal component analysis (PCA)** to transform the data to a smaller sub–space where the new variable are uncorrelated with one another. The preProcess class can apply this transformation by including "pca" in the method argument. Doing this will also force 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.

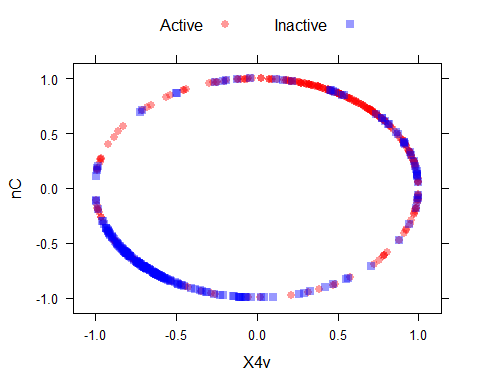
The two figures below show two centered and scaled discritors from the MDRR data before and after the spatial sign transformation. The predictors should be centered and scaled before applyin this transformation.

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.

## 4.4 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

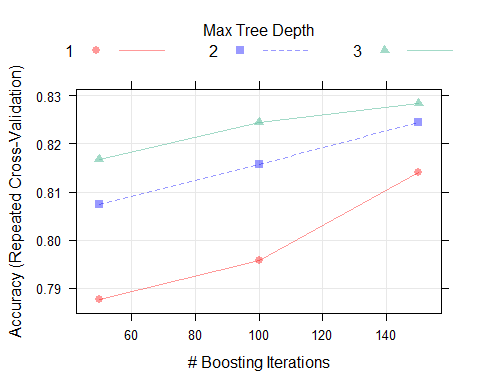
head(training)

## 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.

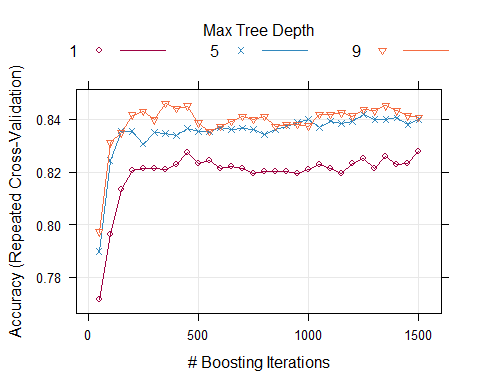
Another option is to use a random sample of possible tuning parameter combinations, i.e. “random search”(pdf). This functionality is described on [this page](http://topepo.github.io/caret/random-hyperparameter-search.html).

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

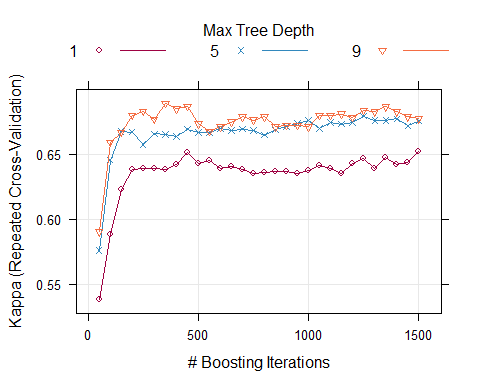
The plot function can be used to examine the relationship between the estimates of performance and the tuninng parameters. For example, a simple invokation of the function shows the results for the first performance measure: 2

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



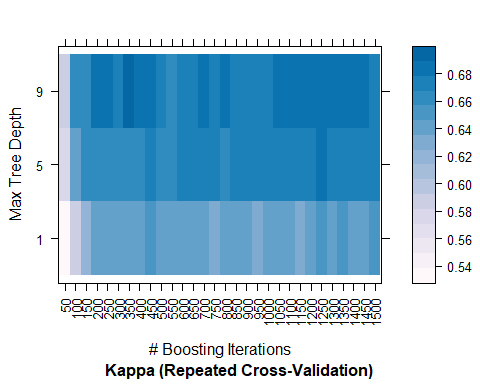
Other performance metrics can be shown using the metric option:

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

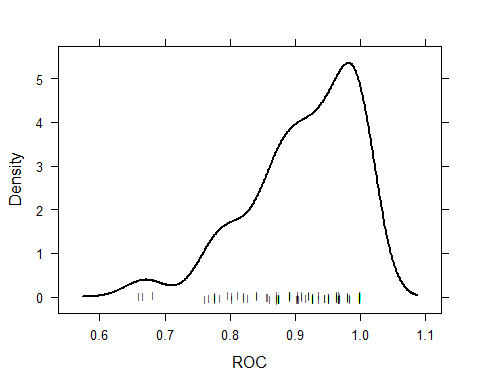
## Exploring and comparing resampling distributions

### 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.

### 5.8.2 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

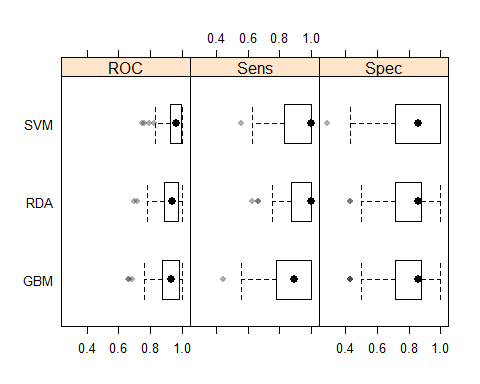
summary(resamps)

##   
## 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

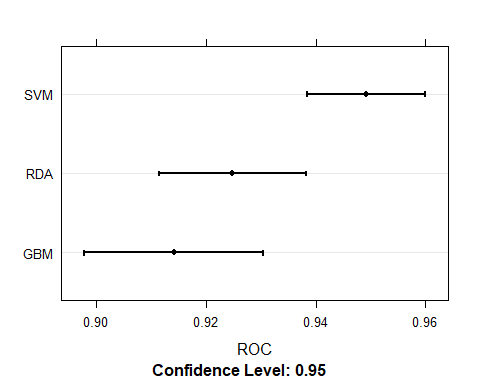
Note that, in this case, the option resamples = "final" should be user-defined in the control objects.

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:

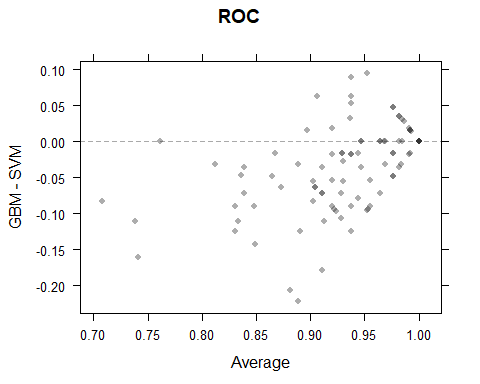
theme1 <- trellis.par.get()  
theme1$plot.symbol$col=rgb(.2, .2, .2, .4)  
theme1$plot.symbol$pch = 16  
theme1$plot.line$col = rgb(1, 0, 0, .7)  
theme1$plot.line$lwd <- 2  
trellis.par.set(theme1)  
  
bwplot(resamps, layout = c(3, 1))



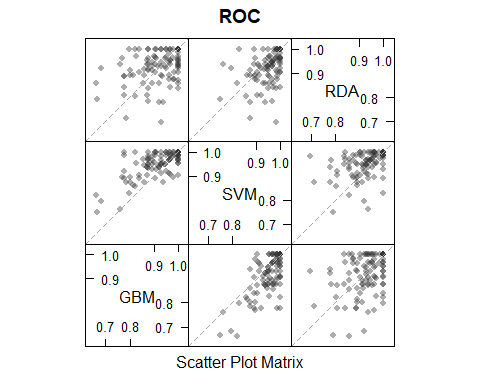
trellis.par.set(caretTheme())  
dotplot(resamps,metric="ROC")



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.

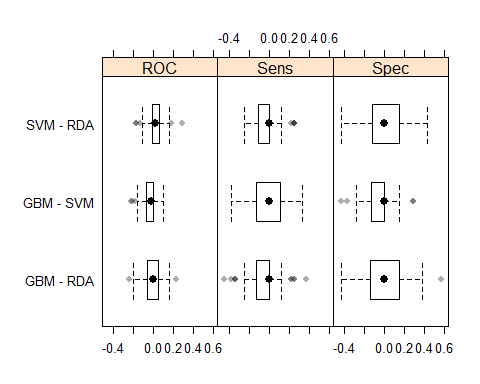
difValues <- diff(resamps)  
difValues

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

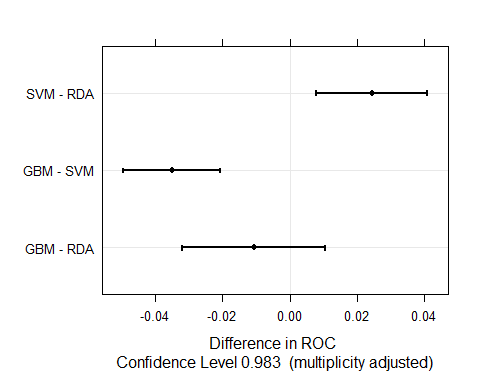
summary(difValues)

##   
## 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)



## 5.9 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, 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.