Appendix R6: Machine Learning (ML)

Cross-Validation (CV) Testing & Resampling

J. Alberto Espinosa

8/1/2021

Table of Contents

Overview

In machine learning and cross-validation we do a lot of random sampling and re-sampling to train and test models. We train the models with a train sub-sample and we then evaluate the trained model using the test sub-sample. All random sampling is done through random number generators (like the random number tables in the back of statistic books). To draw a random sample you start at an arbitrary number in the table, called the **seed**, and start drawing random numbers in sequence, starting with the first random number starting at the seed location. To set the seed, we use the function `set.seed()`. The `set.seed()` then invokes a random number generator (RNG), which dictates the sequence of random numbers to use. There are more than one RNG methods in R and their differences are mainly related the various rounding methods used by the RNG. If you plan to work alone, this is of absolutely no consequence to you, as any RNG will generate random numbers. But if you plan to work with a team and you want to have comparable and repeatable results, it helps to set the same default RNG by everyone in the team. The default can be established with the `RNGkind()` function as follows:

```
RNGkind(sample.kind="default") # To use the R default RNG
```

After doing this, you may want to set the random seed to any value before any random sampling, if you want to get repeatable results across samples. Do not set the seed if you

want the random sample to change every time you run the script. In the example below I use a seed of 1, but it could be 10, 35, or any number you wish. I'm also setting the scientific penalty to 4, to minimize the display of small numbers in scientific notation.

```
set.seed(1)
options(scipen=4) # Limit scientific notation
```

Cross-Validation (CV) and Resampling Methods

Random Splitting Cross Validation (RSCV)

For RSCV, we select the proportion of the data that we want to use for the train sub-sample and then draw a random sample of that size from the data set. We then use the remaining observations for the test sub-sample. Let's use the **Auto** data set in the **{ISLR}** package for this illustration and take a quick look at the first few observations.

```
library(ISLR) # Contains the Auto data set
head(Auto)
```

```
##   mpg cylinders displacement horsepower weight acceleration year origin
## 1  18         8          307        130   3504         12.0   70      1
## 2  15         8          350        165   3693         11.5   70      1
## 3  18         8          318        150   3436         11.0   70      1
## 4  16         8          304        150   3433         12.0   70      1
## 5  17         8          302        140   3449         10.5   70      1
## 6  15         8          429        198   4341         10.0   70      1
##                       name
## 1 chevrolet chevelle malibu
## 2         buick skylark 320
## 3        plymouth satellite
## 4             amc rebel sst
## 5               ford torino
## 6          ford galaxie 500
```

It is always a good idea to go to the R console and take a look at the documentation for the data set with `?Auto` and become familiarized with the variables. Try it on your own.

Now, let's start by counting the number of records in the **Auto** data set.

```
nrow(Auto) # 392 observations in the data set
```

```
## [1] 392
```

Suppose you want to train your model with 70% of the data and test it with the remaining 30%. We start by generating "index" vector of random numbers named **train**, containing 70% of the 392 row numbers from the Auto data set `nrow(Auto)`. It is always a good idea to set the seed before sampling too.

```
set.seed(10)
train <- sample(392, 0.7*392)
train[1:10] # Check a few of the numbers in the train index vector

##  [1] 137 330 368  72 211 344 271 143 391  24
```

The method used above to generate the train vector is not very good, programmatically. This command will not work well if the data changes. In software programming, it is best to store all the main parameters in variables, as illustrated here:

Store the training sub-sample size in a variable. This will make it easier to change later:

```
tr.size <- 0.7
```

Then, generate the train index using `nrow()`, so it will always work when the data set changes:

```
set.seed(10)
train <- sample(nrow(Auto), tr.size*nrow(Auto))
train[1:10] # Take a look at a few values

##  [1] 137 330 368  72 211 344 271 143 391  24
```

Count the number of values in the index vector just to confirm we are doing things correctly:

```
length(train) # 274 or 70% of 392

## [1] 274
```

You can change **tr.size** to 0.6, 0.8 or any other size to easily change the size of the train sub-sample.

We can now "train" (i.e., fit) the model with the train sub-sample. We can do this in 3 (or more) different ways:

1. With the `subset=index vector` parameter

```
lm.fit.train <- lm(mpg ~ horsepower, data = Auto, subset = train)
```

2. By indexing the Auto data set to select just the train observations

```
lm.fit.train <- lm(mpg ~ horsepower, data = Auto[train,])
```

3. Or with my preferred method, by creating separate train and test subsets, so that you have the subsets ready to use any time, and then training the model:

```
Auto.train <- Auto[train,]
Auto.test <- Auto[-train,]
lm.fit.train <- lm(mpg ~ horsepower, data = Auto.train)
```

Note that we use the train index and then a comma and a blank. This is so because Auto.train is a data frame containing rows and columns. `Auto[train,]` selects a subset that contains all the rows matching the values in the **train** index vector, and the blank after the

comma indicates that we will extract all columns. If you don't need all columns, you can always select columns with the appropriate column index.

Now that we have trained the model with the train subset, we are ready to use it to CV test it. Note that I don't output the `summary()` of the model, because we will not use the trained model to interpret results. We will do that later with the full data set. For now, we are just testing the predictive accuracy of the model.

Let's now compute the "CV Test MSE" (i.e., deviance) of the model using the **trained model** to predict the outcomes using the **test** sub-sample. We start with the **Auto.test$mpg** vector, which has the actual outcome values in the test subset. We then subtract the predictions, using **lm.fit.train** as the model with the data in **Auto.test**. This will yield the vector of residuals from our test predictions. We then square the residuals and take the mean:

```
mse.test <- mean( (Auto.test$mpg - predict(lm.fit.train, Auto.test) ) ^ 2)
mse.test # Check out the deviance

## [1] 25.0053
```

To recap, in the example above, we trained the model with the train subset. We then used the trained model to make predictions with the test subset. This is the **CV Test MSE**. Let's see what happens if we compute the error using the same data we used to train the model. You would never do this, but I do it here to illustrate a point. Let's call this error the **Train MSE** and display both errors side by side.

```
mse.train <- mean( (Auto.train$mpg - predict(lm.fit.train, Auto.train) ) ^ 2)
cbind("CV Test MSE" = mse.test, "Train MSE" = mse.train)

##      CV Test MSE Train MSE
## [1,]     25.0053  23.60343
```

Notice that the CV Test MSE is larger than the Train MSE. Why is that? It is because when we calculate the model's error using the same data we used to train the model, the train error will generally under-estimate the CV test error. I say generally, but not always, because sometimes this is not the case due to random sampling. This is the reason why we do resampling because the train error will always understimate the test (i.e., real) error over multiple resamples. Let's take a look at RSCV with resampling.

Random Splitting CV with Re-Sampling

You can do a few resampes and average the Test MSE results to avoid issues with lucky or unlucky random samples. The computational power of R makes this relatively easy. Let's resample 10 times with a random seed each time with a loop (you can just replace 10 with any larger number for more resamples):

```
# Store the desired number of resamples, 10 in this case, in a variable.
Also, store the proportion of train observations in a variable, to make it
easier to change as needed.
```

```
res <- 10
tr.size <- 0.7

# Initialize a vector with 10 elements with 0's, which will hold the test
errors in each loop

mse.test <- replicate(res, 0) # res is 10

for (i in 1:res) # Process this loop "res" times.
{
  sd <- sample(1:1000, 1) # Draw a random seed between 1 and 1000
  set.seed(sd) # Set the seed to this random value

  # Draw the train and test sub-samples
  train <- sample(nrow(Auto), tr.size * nrow(Auto))
  Auto.train <- Auto[train,] # Train subset
  Auto.test <- Auto[-train,] # Test subset

  # Train the model
  lm.fit.train <- lm(mpg ~ horsepower, data = Auto.train)

  # Compute the Test MSE for the i-th sub-sample and store in mse.test vector
  mse.test[i] <- mean( (Auto.test$mpg - predict(lm.fit.train, Auto.test) )^2)
}

print(mse.test, digits=4) # This will show all 10 MSE's calculated in the
loop

##  [1] 27.17 24.69 22.17 20.26 26.92 22.02 25.96 23.86 25.95 29.78
```

Then compute the average Test MSE across the 10 resamples:

```
avg.mse.test <- round(mean(mse.test), digits = 3)
paste("Mean CV Test over", res, "resamples = ", avg.mse.test)

## [1] "Mean CV Test over 10 resamples =  24.877"
```

Leave-One-Out Cross-Validation (LOOCV)

The advantage of RSCV is that it is relatively easy to script and can be used for just about any predictive modeling method. The script above can be easily customized as needed. But LOOCV can be very easily applied with **GLM** models using the `cv.glm()` function. Many ML methods like random forest trees and principal components regression contain functionality to perform LOOCV and KFCV testing. In this section, I illustrate the use of the `cv.glm()` function from the **{boot}** package, which works with `glm()` models.

```
library(boot) # Contains the cv.glm() function
library(ISLR) # Contains the Auto data set
```

```
glm.fit <- glm(mpg ~ horsepower + weight + year, data=Auto)
cv.loo <- cv.glm(Auto, glm.fit)
```

The resulting `cv.glm()` object stores the CV Test MSE results in a list named *delta* **. *The* ** ***deta*** list contains 2 values, which are almost identical. The first delta value is the actual raw CV Test MSE. The second one is some bias-corrected value. Let's just focus on the raw CV test value:

```
test.mse.loo <- round(cv.loo$delta[1], digits = 3) # Round and store the test
MSE
paste("LOOCV Test MSE = ", test.mse.loo)

## [1] "LOOCV Test MSE =  11.899"
```

k-Fold Cross-Validation (KFCV)

KFCV is computed in the exact same way as LOOCV, except that you need to include a parameter **K** to set the number of folds. For example, for 10FCV, you need to use `K = 10`. In essence, the LOOCV method is a special case of KFCV in which K = N, where N is the number of observations in the data. In fact, `K = N` is the default value in the `cv.glm()` function, which is why I did not include a K parameter in the LOOCV calculations.

```
cv.10K <- cv.glm(Auto, glm.fit, K=10)
test.mse.10K <- round(cv.10K$delta[1], digits = 3) # Round and store the test
MSE
paste("10FCV Test MSE = ", test.mse.10K)

## [1] "10FCV Test MSE =  11.867"
```

Let's look at LOOCV and 10KCV together and notice that the results are almost identical, except for rounding.

```
c("MSE LOOCV" = test.mse.loo, "MSE 10FCV" = test.mse.10K)

## MSE LOOCV MSE 10FCV
##    11.899    11.867
```

Bootstrap Regression

Bootstrapping is used for many statistical methods and it is based on re-sampling the data with replacement many times. For CV testing, the bootstrap samples are used to train the model and the observations left out are used for CV testing. There are many R packages to do apply bootstrapping in various statistical computations, like **{boot}**. But for CV testing, the **{caret}** package provides a simple way to apply it. In fact, bootstrap is the default method for CV testing in this package. I discuss the {caret} package next.

The {caret} Package

The {caret} package is a companion to the book *Applied Predictive Modeling* book by Kuhn and Johnson (see: http://appliedpredictivemodeling.com/)

It contains several useful functions for estimating and testing machine learning models. The `train()` function is particularly useful to do CV testing with various models and methods. You can easily test various models using multiple CV resampling method by changing a few parameters. The documentation for this package can be found at: http://topepo.github.io/caret/index.html.

You can use many modeling methods with the **{caret}** package. See: http://topepo.github.io/caret/train-models-by-tag.html.

Let's illustrate how to use of the `train()` function of the **{caret}** package:

```
library(caret) # Load the package
library(ISLR) # Contains the Auto data set
set.seed(1) # Set the seed
```

Let's start by fitting and testing an OLS model with the lm() function. The {caret} package does not have its own functions to train models. Instead, it uses the `train()` function in which the `method=` parameter invokes the actual modeling method, `lm()` in this case. This is an excellent feature because the **{caret}** package will yield the exact same results than the `lm()` function, or which ever model function you use, because it will call the actual `lm()` function to fit the model. The default CV testing is done with bootstrap with 25 resamples.

```
lm.fit.caret <- train(mpg ~ horsepower + weight + year,
                      data = Auto,
                      method = "lm")

lm.fit.caret # Reports RMSE and R squared
```

```
## Linear Regression
##
## 392 samples
##   3 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 392, 392, 392, 392, 392, 392, ...
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   3.419317  0.8095377  2.625425
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

You can print specific results by invoking the train() object properties:

```
lm.fit.caret$results$RMSE # This is where RMSE is stored

## [1] 3.419317

lm.fit.caret$results$RMSE ^ 2 # To get the MSE

## [1] 11.69173

summary(lm.fit.caret) # Same as lm() results

##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -8.7911 -2.3220 -0.1753  2.0595 14.3527
##
## Coefficients:
##               Estimate  Std. Error t value Pr(>|t|)
## (Intercept) -13.7193600   4.1817582  -3.281  0.00113 **
## horsepower   -0.0049998   0.0094391  -0.530  0.59663
## weight       -0.0064477   0.0004089 -15.768  < 2e-16 ***
## year          0.7487051   0.0521200  14.365  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.43 on 388 degrees of freedom
## Multiple R-squared:  0.8083, Adjusted R-squared:  0.8068
## F-statistic: 545.4 on 3 and 388 DF,  p-value: < 2.2e-16
```

Here are more results:

```
lm.fit.caret$resample # Display fit stats for each sample

##        RMSE  Rsquared      MAE   Resample
## 1  3.376097 0.8056360 2.635579 Resample01
## 2  3.490104 0.8074854 2.679650 Resample02
## 3  3.264611 0.8168003 2.514205 Resample03
## 4  3.312251 0.8160117 2.460108 Resample04
## 5  3.204453 0.8254676 2.553712 Resample05
## 6  3.468692 0.8194017 2.593418 Resample06
## 7  3.281412 0.8241793 2.616985 Resample07
## 8  3.344366 0.7995870 2.609919 Resample08
## 9  3.638589 0.8131311 2.688768 Resample09
## 10 3.225821 0.8185690 2.602811 Resample10
## 11 3.509129 0.8059482 2.672373 Resample11
## 12 3.136018 0.8317275 2.479244 Resample12
## 13 3.459333 0.7796983 2.454693 Resample13
## 14 3.725826 0.8075100 2.787716 Resample14
## 15 3.741200 0.8039969 2.751579 Resample15
## 16 3.407245 0.8256503 2.772444 Resample16
```

```
## 17 3.663826 0.7765565 2.796614 Resample17
## 18 3.403578 0.8006071 2.680102 Resample18
## 19 3.051953 0.8197103 2.377129 Resample19
## 20 3.581487 0.7808526 2.763787 Resample20
## 21 3.817122 0.7788553 2.849153 Resample21
## 22 3.198878 0.8217708 2.426788 Resample22
## 23 3.385999 0.8159974 2.606749 Resample23
## 24 3.248830 0.8340675 2.576248 Resample24
## 25 3.546117 0.8092249 2.685858 Resample25

lm.fit.caret$results # Display all bootstrap results

##   intercept     RMSE  Rsquared      MAE    RMSESD RsquaredSD     MAESD
## 1      TRUE 3.419317 0.8095377 2.625425 0.2003173 0.01631914 0.1257239
```

Also, the `trControl` parameter and the `trainControl()` function invoked within the `train()` function, allows you to use other re-sampling methods, such as **boot** (default), **LOOCV**, **CV**, etc.). For example, to change the default number of bootstrap samples from 25 to, for example 100:

```
lm.fit.caret.100 <- train(mpg ~ horsepower + weight + year,
                          data = Auto, method = "lm",
                          trControl = trainControl(number = 100))

lm.fit.caret.100$results # Display bootstrap results for 100 samples

##   intercept     RMSE  Rsquared      MAE    RMSESD RsquaredSD     MAESD
## 1      TRUE 3.462784 0.8093771 2.646649 0.2565325 0.01982924 0.1599223
```

To change the CV method to 10FCV change the `method=` parameter to `"cv"`:

```
lm.fit.caret.10FCV <- train(mpg ~ horsepower + weight,
                            data = Auto, method = "lm",
                            trControl = trainControl(method="cv", number=10))

lm.fit.caret.10FCV # Check the RMSE

## Linear Regression
##
## 392 samples
##   2 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 355, 353, 353, 353, 353, 353, ...
## Resampling results:
##
##   RMSE     Rsquared   MAE
##   4.22225  0.7155102  3.253505
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

To change the CV method to LOOFCV change the `method=` parameter to `"loocv"`:

```
lm.fit.caret.LOOCV <- train(mpg ~ horsepower + weight,
                            data=Auto, method="lm",
                            trControl = trainControl(method="loocv"))

lm.fit.caret.LOOCV # Check the RMSE
```

```
## Linear Regression
##
## 392 samples
##   2 predictor
##
## No pre-processing
## Resampling: Leave-One-Out Cross-Validation
## Summary of sample sizes: 391, 391, 391, 391, 391, 391, ...
## Resampling results:
##
##   RMSE    Rsquared  MAE
##   3.2609  NaN       3.2609
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

Now let's quickly change the model method to a Random Forest tree model with 10FCV using the `method = "rf"` parameter:

```
rf.fit.caret.10FCV <- train(mpg ~ horsepower + weight + year,
                            data=Auto, method="rf",
                            trControl = trainControl(method="cv", number=10))
```

```
## note: only 2 unique complexity parameters in default grid. Truncating the
grid to 2 .
```

```
rf.fit.caret.10FCV # Check the RMSE
```

```
## Random Forest
##
## 392 samples
##   3 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 352, 352, 352, 353, 353, 353, ...
## Resampling results across tuning parameters:
##
##   mtry  RMSE      Rsquared   MAE
##   2     2.827029  0.8735483  2.067045
##   3     2.852559  0.8711088  2.073289
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was mtry = 2.
```

Let's fit a **Neural Network** with the `method = "neuralnet"` parameter:

```
nn.fit <- train(mpg ~ horsepower + weight, data=Auto, method="neuralnet")
nn.fit # Check the RMSE
```

```
## Neural Network
##
## 392 samples
##   2 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 392, 392, 392, 392, 392, 392, ...
## Resampling results across tuning parameters:
##
##   layer1  RMSE      Rsquared     MAE
##   1       7.740227          NaN  6.498222
##   3       7.740228          NaN  6.498223
##   5       7.740228  0.009666173  6.498223
##
## Tuning parameter 'layer2' was held constant at a value of 0
## Tuning
##  parameter 'layer3' was held constant at a value of 0
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were layer1 = 1, layer2 = 0 and layer3
= 0.
```