Exercise 6 - Machine Learning and Cross Validation (Solution)

Prof. J. Alberto Espinosa
February 28, 2023

Table of Contents

Preparation

Download the **Ex6_ML_CV_YourLastName.Rmd** R Markdown file to your working directory, **rename** it with your **last name** and follow the instructions below. When you finish, upload onto Blackboard the .Rmd file or the knitted file as a Word, HTML or PDF file.

Knitting and **formatting** are worth up to **3 points** in this and all exercises.

Formatting: Please ensure that all your text narratives are fully visible (if I can't see the text, I can't grade it). Also, please ensure that your **Table of Contents** is visible and properly formatted. Also, please prepare your R Markdown file with a **professional appearance**, as you would for top management or an important client. Please, write all your interpretation narratives in the text area, outside of the R code chunks, with the appropriate formatting and businesslike appearance. **Note:** I write all my interpretation solutions inside of the R code chunk to suppress their display until I print the solution, but don't need to do this. I will read your submission as a report to a client or senior management. Anything unacceptable to that audience is unacceptable to me.

Submission: Submit your knitted homework document in Canvas. There is no need to submit the .Rmd file, just your knitted file.

Important Technical Note about R's **Random Number Generator (RNG)**. R discovered problems with the `sample()` function, caused by some inconsistencies in the `set.seed()` **random number generator (RNG)**. They corrected this with R version 3.6.0. However, we will be doing a lot of random sampling for CV and this may cause your numbers to be somewhat different than mine, depending on what is the default RNG in your R installation. The differences are usually due to rounding differences in randomly generated values. If you are working on your own project, this doesn't affect you because your own results will be consistent. But for your results to match mine, we all need to set the **RNG** to the same default and use the **same seed**. Still, your numbers may not exactly match mine and that is OK, as long as they are not radically different. But let's all **set** the same **RNG default** this way:

```
# Done for you

RNGkind(sample.kind="default")
```

1. Random Splitting (Holdout Sample) Cross Validation (RSCV)

1.1 Train Vector.

Load the **{MASS}** library, which contains the **Boston** data set. Run `?Boston` from the console (not in the script) and inspect its variables. Then use `set.seed(1)` so that you get the same results if you run your cross validation commands multiple times.

Suppose that we want to extract a train sub-sample of 70% of the data. The easiest way to do this would be to find out the number of observations in the Boston data set, which is 506, and then create an index vector containing a random sample of 70% of the numbers between 1 and 506, or 354 numbers:

```
# Done for you
train <- sample(506, 354)
train[1:10] # Check the first 10 values

##  [1] 394 242 462 126 497 373 396 365 137 308
```

This train vector contains 352 random numbers that we can use as an index to create our train subset. However, if the data changes the commands above will have to be edited. It is better to do thing computationally, so that we minimize the changes we need to do as our needs change and as the data changes. Let's do that.

Before we start, load the **{MASS}** library where the **Boston** data set reside and set the seed to `set.seed(1)`.

Then, create a variable named **tr** to store the proportion of observations in the train subset, and store the value **0.7** in it.

Then generate the **train** vector as we did above, but using the **nrow(Boston)** function instead of the fixed number 506, and using `tr * nrow(Boston)` instead of the fixed number 354. This way, if we want to change our training proportion, we just have to change the value in **tr** and if the data changes, the **nrow()** function will catch that.

Store the results in the **train** vector and display the first 10 values as we did above.

```
library(MASS)
set.seed(1)

tr <- 0.7
```

```
train <- sample(nrow(Boston), tr * nrow(Boston))
train[1:10] # Check the first 10 values

##  [1] 505 324 167 129 418 471 299 270 466 187
```

1.2 Train and Test Sub-Samples.

Now that we have our train index vector, let's generate a train subset by selecting the observations in the row numbers matching the train index and all columns of the **Boston** data set (i.e., `Boston[train, ]`. Name this sub-sample **Boston.train**. Just to be certain you did it right, count and display the number of rows of Boston.train using the **nrow()** function. Then create a test subset named **Boston.test** using all the remaining observations not included in the train subset (i.e., `Boston[-train, ]`. Then, count and display the number of rows of Boston.test.

```
Boston.train <- Boston[train, ]
nrow(Boston.train)

## [1] 354

Boston.test <- Boston[-train, ]
nrow(Boston.test)

## [1] 152
```

1.3 Train the Model.

Fit a linear model to predict the median value of houses in Boston counties **medv** using the **train subset**. Use `crim + chas + rm + age + tax + ptratio + lstat` as predictors. Store your resulting model in an object named **fit.train**.

Technical Note: after we fit the fit.train model, we could display the summary results to inspect the outcomes. But we really don't have to because we are only doing this to compute the cross-validation test error. Once we are happy with the results at the end, we will re-fit the model with the entire Boston data set, not just the train subset.

```
fit.train <- lm(medv ~ crim + chas + rm + age + tax + ptratio + lstat,
                data = Boston.train)
```

Now, compute the MSE for the train model with the function `mean(fit.train$residuals ^ 2)`. Save the results in an object named **train.mse**. Then compute RMSE for the train model by taking the `sqrt()` of mse.train, and save it in an object named **train.rmse**.

Then bind the two results into a data frame by binding them as columns using the **cbind()** function and label the respective columns **MSE** and **RMSE**. Store the result in an object named **train.error**. Then display the **train.error**.

Technical Note: You may be wondering why I ask you to create and name these objects. Saving your results in named objects allows you to use the objects later on in the program.

Important note: the train MSE and RMSE are computed with the same data used to train the model, so it will **underestimate** the test model error. For this reason, we would not use the train MSE or train RMSE. We are only doing this to illustrate the differences with the test MSE and RMSE a bit later.

```
train.mse <- mean(fit.train$residuals ^ 2)
train.rmse <- sqrt(train.mse)

train.error <- cbind("MSE" = train.mse,
                     "RMSE" = train.rmse)
train.error

##           MSE     RMSE
## [1,] 24.34824 4.934393
```

1.4 Test MSE and RMSE

Compute the (Random Split Cross-Validation) **RSCV Test MSE** by making predictions with the trained model **fit.train**, but using the **Boston.test** subset. Store the resulting MSE in an object named **test.mse.rs** and display the result.

To help you understand how to do this, let's build the **test.mse.rs** formula in steps:

First let's make the predictions using the trained model, but with the test subset data, with the function `predict(fit.train, Boston.test)` and store it in an object named **pred.test**. Check the first 6 predictions with the **head()** function.

```
pred.test <- predict(fit.train, Boston.test)
head(pred.test)

##        5        6        7        8        9       10
## 31.03927 27.17429 23.90289 21.66391 13.37194 21.67977
```

Then, subtract these predictions from the actual test outcomes `Boston.test$medv`, square the differences and take the mean. Store the results in a vector named **mse.test.rscv**. Then compute the **RSCV Test RMSE** by taking the `sqrt()` of **mse.test.rscv**.

Then, use the `cbind()` function to bind the two columns into one data frame, with the respective column labels **"MSE"** and **"RMSE"** and name it **test.error.rscv**. Then display this **test.error.rscv** data frame.

```
test.mse.rscv <- mean( (Boston.test$medv - pred.test) ^ 2 )
test.rmse.rscv <- sqrt(test.mse.rscv)

test.error.rscv <- cbind("MSE" = test.mse.rscv,
                         "RMSE" = test.rmse.rscv)

test.error.rscv

##          MSE     RMSE
## [1,] 31.5938 5.620836
```

2. Leave One Out Cross-Validation (LOOCV)

We saw above how to do RSCV, which is essentially hard-coding the CV testing. But a single random subset sample could be deceiving if we get a lucky or unlucky train subset sample. This is why re-sampling multiple times is recommended. A Test MSE based on the average of multiple re-samples is more representative of the true error of the model. But this means writing a loop to do the predictions and calculations over and over. This is not so difficult, but there are functions already developed to do that and many modeling packages include CV functions. LOOCV and KFCV are two very popular re-sampling CV methods.

2.1 Fit a GLM Model.

The function we feature in 2.2 requires a **glm()** object. If you are fitting an OLS model, you can use the **lm()** function to get OLS fit statistics, but you need to use the **glm()** function to do LOOCV or KFCV.

Using the full **Boston** data set, fit a **GLM** model to predict **medv** using the same predictors `crim + chas + rm + age + tax + ptratio + lstat`. Again, we don't need to display the `summary()` results because we are just evaluating the predictive accuracy of the model at this point. Store the results in an object named **glm.fit**. Just to verify results, display the model's **2LL** or `glm.fit$deviance`, null deviance or `glm.fit$null.deviance` and the proportion of deviance reduction of the model `(glm.fit$null.deviance - glm.fit$deviance) / glm.fit$null.deviance`, which is equivalent to the R squared.

```
glm.fit <- glm(medv ~ crim + chas + rm + age + tax + ptratio + lstat,
               data = Boston)

glm.fit$deviance

## [1] 13178.72

glm.fit$null.deviance

## [1] 42716.3

(glm.fit$null.deviance - glm.fit$deviance) / glm.fit$null.deviance

## [1] 0.6914825
```

2.2 Leave One Out (LOOCV)

Let's compute the Test MSE and RMSE using LOOCV. Load the **{boot}** library and use the `cv.glm()` function and the **glm.fit** object above to compute the **LOOCV Test MSE**. You need to feed `Boston, glm.fit` to the `cv.glm()` function. Since **LOOCV** is the **default** CV for `cv.glm()`, we don't need to specify any additional parameters. The actual MSE result is contained in the first element of the attribute `cv.glm(Boston, glm.fit)$delta[1]`.

Technical Note: The **$delta** attribute in the **cv.glm()** object is a vector with 2 elements. The first element is the raw CV test result. The second element is a bias-adjusted version of the CV test result, which we don't need to use for our purposes. So we use the index [1] to extract the first element, which is the raw CV test value.

Store this value in an object named **test.mse.loo**. Then compute the corresponding LOOCV Test RMSE with `sqrt(test.mse.loo)` and store it in an object named **test.rmse.loo**

Then, use the `cbind()` function to create a new data frame with the two columns labeled "MSE" and "RMSE", respectively, and name this data frame **test.error.loocv**. Then display this data frame.

```
library(boot)

test.mse.loo <- cv.glm(Boston, glm.fit)$delta[1]
test.rmse.loo <- sqrt(test.mse.loo)

test.error.loocv <- cbind("MSE" = test.mse.loo,
                          "RMSE" = test.rmse.loo)

test.error.loocv

##           MSE     RMSE
## [1,] 27.67968 5.261148
```

3. K-Fold Validation (KFCV)

3.1 10-Fold Validation (10FCV)

Using the same `cv.glm()` function and **glm.fit** model object, compute and display the **10-Fold** cross validation MSE for this model. This time you need to add the parameter `K = 10` in the `cv.glm()` function. Store the result contained in `$delta[1]` in an object named **test.mse.10f** and also store the square root of this value in an object named **test.rmse.10f**.

Then, use the `cbind()` function to create a new data frame with the two columns labeled "MSE" and "RMSE", respectively, and name this data frame **test.error.10fcv**. Then display this data frame.

```
test.mse.10f <- cv.glm(Boston,glm.fit, K=10)$delta[1]
test.rmse.10f <- sqrt(test.mse.10f)

test.error.10fcv <- cbind("MSE" = test.mse.10f,
                          "RMSE" = test.rmse.10f)

test.error.10fcv

##         MSE     RMSE
## [1,] 28.022 5.293581
```

3.2 Compare Results

Now, to make it easier to compare errors, let's bind the 4 type of errors computed above. Use the `rbind()` function to row-bind the **train.error**, **test.error.rscv**, **test.error.loocv** and **test.error.10fcv**. Then use the `rownames()` function to name the respective rows "Train", "RSCV Test", "LOOCV Test" and "10FCV Test", using the concatenate function `c()`. Name this new data frame **errors.all** and then display it, but using the `kable()` function to get a nicely formatted table. Use `knitr::kable(errors.all, format = "simple", digits = 2, caption = "CV Result Summary")`.

```
errors.all <- rbind(train.error,
                    test.error.rscv,
                    test.error.loocv,
                    test.error.10fcv)

rownames(errors.all) <- c("Train", "RSCV Test", "LOOCV Test", "10FCV Test")

knitr::kable(errors.all,
             format = "simple",
             digits = 2,
             caption = "CV Result Summary")
```

CV Result Summary

	MSE	RMSE
Train	24.35	4.93
RSCV Test	31.59	5.62
LOOCV Test	27.68	5.26
10FCV Test	28.02	5.29

4. Commentary

Provide a brief commentary of the results above. Is there a meaning to the difference between these 3 MSE Cross-Validation result? Briefly explain

```
# My comments to you:

# These are just 3 different ways of doing cross-validation testing. Some
students may be tempted to say that one CV method is better that others
because it yields a lower MSE, but this is incorrect. It would be like saying
that a judge at an Olympic competition is better than another because she
always gives tougher scores. These are just different metrics of the CV test
errors.

# Commentary:

# We should never use the Train MSE because it underestimates the true MSE of
the model. The difference in the results across the other 3 methods doesn't
really matter. The 3 values are just different measures of the Test MSE of
```

```
the same model, using different re-sampling strategies for cross-validation.
The LOOCV is generally a stronger test because it uses all the data, except
one point, to fit the model and then test with the point left out, and does
this until all data points have been tested. But this can be computationally
heavy. Experts concur based on many simulation studies that a 10FCV is
sufficient and a pretty stable Test MSE measure. Since these cross-validation
tests are NOT generally used to evaluate a single model, but to make
comparisons across many models, all 3 methods will generally yield similar
ranking of the methods being evaluated. However, the RSCV is based on a
single sample, so it is not as accurate in measuring CV test error. RSCV with
multiple re-samples would work better, but LOOCV and 10FCV are still
preferred.
```

Technical Note

The MSE for any CV test is difficult to interpret because it involves the squared value of the errors. It is an excellent deviance statistic, but doesn't have an intuitive explanation for a management audience. We computed the RSCV Test **RMSE**. for this reason, which is the average error. Roughly, the margin of error of a predictive model, with approximately 95% confidence is +/- (2 * Test RMSE). For example, using the LOOCV method:

```
# Done for you

paste("Based on LOOCV, the prediction margin of error = +/- $",
      1000 * 2 * round(test.rmse.loo, digits = 3))

## [1] "Based on LOOCV, the prediction margin of error = +/- $ 10522"
```