Cross-Validation

We've discussed the idea of supervised learning. In this paradigm we have a **response** or **target** variable we are trying to predict.

- We posit some model for that variable that is a function of our predictor variables.
- We fit that model to data (usually can be seen as minimizing some *loss* function).
- We judge how well our model does at predicting using some **metric.**
    - Usually we evaluate our metric on the test set after doing a training/test split.

Recall our example of fitting an MLR model from the last section of notes. There we first split the data into a training and test set. We fit the model on the training set. Then we looked at the RMSE on the test set to judge how well the models did.

$$RMSE = \sqrt{\frac{1}{n_{test}} \sum_{i=1}^{n_{test}} \left(y_i - \hat{y}_i\right)^2}$$

where y_i is the actual response from the test set and $\hat{y}_i$ is the prediction for that response value using the model.

Note: The terms metric and loss function are often used interchangeably. They are slightly different.

- **Loss function** is usually referred to as the criteria used to optimize or fit the model.
    - For an MLR model we usually fit by minimizing the sum of squared errors (MSE or RMSE equivalently) which is the same as using maximum likelihood under the iid Normal, constant variance, error model.
    - In this case, MSE or RMSE would be our loss function
- Our **metric** is the criteria for evaluating the model. When we have a numeric response, our most common criteria is RMSE! In this case the same as the common loss function for fitting the model.

We could fit our MLR model using a different loss function (say minimizing the absolute deviations). We could also evaluate our model using a different metric (say the average of the absolute deviations).

Cross Validation Video Introduction

An alternative to the training test split is to use cross-validation. Check out the video below for a quick introduction to cross-validation. Please pop this video out

and watch it in the full panopto player!

PDF of these notes.

CV without a Training/Test Set Split

Sometimes we don't have a lot of data so we don't want to split our data into a training and test set. In this setting, CV is often used by itself to select a final model.

Consider our data set about the selling price of motorcycles:

```
library(tidyverse)
bike_data <- read_csv("https://www4.stat.ncsu.edu/~online/datasets/bikeDetai
bike_data <- bike_data |>
  mutate(log_selling_price = log(selling_price),
         log_km_driven = log(km_driven)) |>
  select(log_km_driven, year, log_selling_price, everything())
bike_data
```

```
# A tibble: 1,061 × 9
   log_km_driven  year log_selling_price name    selling_price seller_type
owner
           <dbl> <dbl>             <dbl> <chr>           <dbl> <chr>
<chr>
 1          5.86  2019              12.1  Royal …        175000 Individual
1st …
 2          8.64  2017              10.7  Honda …         45000 Individual
1st …
 3          9.39  2018              11.9  Royal …        150000 Individual
1st …
 4         10.0   2015              11.1  Yamaha…         65000 Individual
1st …
```

```
1st …
 5          9.95  2011              9.90 Yamaha…         20000 Individual
2nd …
 6         11.0   2010              9.80 Honda …         18000 Individual
1st …
 7          9.74  2018             11.3  Honda …         78500 Individual
1st …
 8         10.6   2008             12.1  Royal …        180000 Individual
2nd …
 9         10.4   2010             10.3  Hero H…         30000 Individual
1st …
10         10.6   2016             10.8  Bajaj …         50000 Individual
1st …
# ℹ 1,051 more rows
# ℹ 2 more variables: km_driven <dbl>, ex_showroom_price <dbl>
```

- Let's consider our three linear regression models for the `log_selling_price`
    - Model 1: log_selling_price = intercept + slope*year + Error
    - Model 2: log_selling_price = intercept + slope*log_km_driven + Error
    - Model 3: log_selling_price = intercept + slope*log_km_driven + slope*year + Error
- We can use CV error to choose between these models without doing a training/test split.
- We'll see how to do this within the `tidymodels` framework shortly. For now, let's do it by hand so we can make sure we understand the process CV uses!

Implementing 10 fold Cross-Validation Ourselves

Let's start with dividing the data into separate (distinct) folds.

1. Split the data into 10 separate folds (subsets)
    - We can do this by randomly reordering our observations and taking the first ten percent to be the first fold, 2nd ten percent to be the second fold, and so on.

```
nrow(bike_data)
```

```
[1] 1061
```

```
size_fold <- floor(nrow(bike_data)/10)
```

- Each fold will have 106 observations. There will be an extra observation in this case, we'll lump that into our last fold.
- Let's get our folds by using `sample()` to reorder the indices.
- Then we'll use a for loop to cycle through the pieces and save the folds in a `list`

```
set.seed(8)
random_indices <- sample(1:nrow(bike_data), size = nrow(bike_data), replace
#see the random reordering
```

```
head(random_indices)
```

```
[1] 591  12 631 899 867  86
```

```
#create a list to save our folds in
folds <- list()
#now cycle through our random indices vector and take the appropriate observ
for(i in 1:10){
  if (i < 10) {
    fold_index <- seq(from = (i-1)*size_fold +1, to = i*size_fold, by = 1)
    folds[[i]] <- bike_data[random_indices[fold_index], ]
  } else {
    fold_index <- seq(from = (i-1)*size_fold +1, to = length(random_indices)
    folds[[i]] <- bike_data[random_indices[fold_index], ]
  }
}
folds[[1]]
```

```
# A tibble: 106 × 9
   log_km_driven  year log_selling_price name    selling_price seller_type
owner
           <dbl> <dbl>             <dbl> <chr>           <dbl> <chr>
<chr>
 1         10.1   2014              10.3  Honda …         30000 Individual
1st …
 2          9.21  2016              10.2  Honda …         28000 Individual
2nd …
 3         11.5   2007               9.21 Hero H…         10000 Individual
1st …
 4         10.7   2015              10.3  Honda …         30000 Individual
2nd …
 5          9.40  2014              11.6  Royal …        110000 Individual
1st …
 6         10.8   2009              10.1  Hero H…         25000 Individual
1st …
 7          9.62  2018              10.5  Hero H…         35000 Individual
1st …
 8         10.3   2016              11.2  Bajaj …         70000 Individual
1st …
 9          9.90  2015              10.8  Yamaha…         50000 Individual
1st …
10          9.71  2017              11.8  KTM 25…        135000 Individual
1st …
# ℹ 96 more rows
# ℹ 2 more variables: km_driven <dbl>, ex_showroom_price <dbl>
```

```
folds[[2]]
```

```
# A tibble: 106 × 9
   log_km_driven  year log_selling_price name    selling_price seller_type
owner
           <dbl> <dbl>             <dbl> <chr>           <dbl> <chr>
<chr>
 1         10.7   2013              10.2  Bajaj …         28000 Individual
```

```
1st …
 2          9.21  2018             10.7  Honda …         46000 Individual
1st …
 3          9.68  2016             11.0  Suzuki…         60000 Individual
1st …
 4          9.38  2017             11.6  Honda …        110000 Individual
1st …
 5          8.92  2018             11.1  Hero S…         65000 Individual
1st …
 6          8.99  2018             11.4  Bajaj …         87000 Individual
1st …
 7         11.0   2010              9.90 Bajaj …         20000 Individual
1st …
 8         10.8   2010              9.90 Hero  …         20000 Individual
1st …
 9          9.85  2016             10.5  TVS Ju…         35000 Individual
1st …
10         10.6   2017             11.0  Bajaj …         60000 Individual
2nd …
# ℹ 96 more rows
# ℹ 2 more variables: km_driven <dbl>, ex_showroom_price <dbl>
```

- Let's put this process into our own function for splitting our data up!
- This function will take in a data set and a number of folds (`num_folds`) and returns a list with the folds of the data.

```
get_cv_splits <- function(data, num_folds){
  #get fold size
  size_fold <- floor(nrow(data)/num_folds)
  #get random indices to subset the data with
  random_indices <- sample(1:nrow(data), size = nrow(data), replace = FALSE)
  #create a list to save our folds in
  folds <- list()
  #now cycle through our random indices vector and take the appropriate obse
  for(i in 1:num_folds){
    if (i < num_folds) {
      fold_index <- seq(from = (i-1)*size_fold +1, to = i*size_fold, by = 1)
      folds[[i]] <- data[random_indices[fold_index], ]
    } else {
      fold_index <- seq(from = (i-1)*size_fold +1, to = length(random_indice
      folds[[i]] <- data[random_indices[fold_index], ]
    }
  }
  return(folds)
}
folds <- get_cv_splits(bike_data, 10)
```

2. Now we can fit the data on nine of the folds and test on the tenth. Then switch which fold is left out and repeat the process. We'll use MSE rather than RMSE as our metric for now.

```
#set the test fold number
test_number <- 10
```

```
#pull out the testing fold
test <- folds[[test_number]]
test
```

```
# A tibble: 107 × 9
   log_km_driven  year log_selling_price name    selling_price seller_type
owner
           <dbl> <dbl>             <dbl> <chr>           <dbl> <chr>
<chr>
 1          10.3  2015              10.1 Honda …         25000 Individual
1st …
 2          10.4  2016              10.5 Hero H…         35000 Individual
1st …
 3          10.1  2018              11.0 Yamaha…         60000 Individual
1st …
 4          10.4  2013              10.6 Bajaj …         40000 Individual
1st …
 5           7.57 2017              12.0 Mahind…        165000 Individual
1st …
 6           9.90 2019              11.4 TVS Ap…         90000 Individual
1st …
 7           8.52 2017              11.1 Bajaj …         65000 Individual
1st …
 8           9.17 2018              11.5 TVS Ap…         95000 Individual
1st …
 9           8.61 2016              10.5 Hero G…         38000 Dealer
1st …
10          10.1  2015              10.7 Yamaha…         45000 Individual
1st …
# ℹ 97 more rows
# ℹ 2 more variables: km_driven <dbl>, ex_showroom_price <dbl>
```

```
#remove the test fold from the list
folds[[test_number]] <- NULL
#note that folds is only length 9 now
length(folds)
```

```
[1] 9
```

- Recall the `purrr:reduce()` function. This allows us to iteratively apply a function across an object. Here we'll use `reduce()` with `rbind()` to combine the tibbles saved in the different folds.

```
#combine the other folds into train set
train <- purrr::reduce(folds, rbind)
#the other nine folds are now in train
train
```

```
# A tibble: 954 × 9
   log_km_driven  year log_selling_price name    selling_price seller_type
owner
           <dbl> <dbl>             <dbl> <chr>           <dbl> <chr>
<chr>
 1          11.2  2012              10.7 Hero H…         45000 Individual
```

```
1st …
 2         10.4   2014              10.5 Hero I…         35000 Individual
1st …
 3          9.62  2017              11.8 Bajaj …        138000 Individual
1st …
 4          8.85  2017              11.9 Royal …        150000 Individual
1st …
 5         10.6   2010              10.5 Bajaj …         35000 Individual
1st …
 6          8.85  2018              11.2 Honda …         70000 Individual
1st …
 7          8.07  2019              11.4 Hero X…         87000 Individual
1st …
 8          9.80  2017              11.2 Yamaha…         75000 Individual
1st …
 9          9.85  2014              11.5 Honda …        100000 Individual
1st …
10          9.96  2018              11.3 Bajaj …         80000 Individual
1st …
# ℹ 944 more rows
# ℹ 2 more variables: km_driven <dbl>, ex_showroom_price <dbl>
```

- Now we can fit our model and check the metric on the test set

```
fit <- lm(log_selling_price ~ log_km_driven, data = train)
#get test error (MSE - square RMSE)
(yardstick::rmse_vec(test[["log_selling_price"]],
                     predict(fit, newdata = test)))^2
```

```
[1] 0.3318796
```

- Ok, let's wrap that into a function for ease! Let `y` be a string for the response, `x` a character vector of predictors, and `test_number` the fold to test on.

```
find_test_metric <- function(y, x, folds, test_number){
  #pull out the testing fold
  test <- folds[[test_number]]
  #remove the test fold from the list
  folds[[test_number]] <- NULL
  #combine the other folds into train set
  train <- purrr::reduce(folds, rbind)
  #fit our model. reformulate allows us to take strings and turn that into a
  fit <- lm(reformulate(termlabels = x, response = y), data = train)
  #get test error
  (yardstick::rmse_vec(test[[y]], predict(fit, newdata = test)))^2
}
```

- We can use the `purrr::map()` function to apply this function to `1:10`. This will use each fold as the test set once.

```
folds <- get_cv_splits(bike_data, 10)
purrr::map(1:10, find_test_metric,
```

```
            y = "log_selling_price",
            x = "log_km_driven",
            folds = folds)
```

```
[[1]]
[1] 0.2829525

[[2]]
[1] 0.3260931

[[3]]
[1] 0.4689752

[[4]]
[1] 0.3097491

[[5]]
[1] 0.3006277

[[6]]
[1] 0.3152691

[[7]]
[1] 0.4214135

[[8]]
[1] 0.2991941

[[9]]
[1] 0.3496436

[[10]]
[1] 0.4744565
```

3. Now we combine these into one value! We'll average the MSE values across the folds and then take the square root of that to obtain the RMSE (averaging square roots makes less sense, which is why we found MSE first)
    - We can again use `reduce()` to combine all of these values into one number! We average it by dividing by the number of folds and finally find the square root to make it the RMSE!

```
sum_mse <- purrr::map(1:10,
                      find_test_metric,
                      y = "log_selling_price",
                      x = "log_km_driven",
                      folds = folds) |>
  reduce(.f = sum)
sqrt(sum_mse/10)
```

```
[1] 0.5956823
```

    - Let's put it into a function and make it generic for the number of folds we have

have.

```
find_cv <- function(y, x, folds, num_folds){
  sum_mse <- purrr::map(1:num_folds,
                        find_test_metric,
                        y = y,
                        x =x,
                        folds = folds) |>
    reduce(.f = sum)
  return(sqrt(sum_mse/num_folds))
}
find_cv("log_selling_price", "log_km_driven", folds, num_folds = 10)
```

```
[1] 0.5956823
```

Compare Different Models Using Our CV Functions

Now let's compare our three separate models via 10 fold CV! We can use the same `folds` with different models.

1. First our model with just `year` as a predictor:

```
folds <- get_cv_splits(bike_data, 10)
cv_reg_1 <- find_cv("log_selling_price", "year", folds, num_folds = 10)
```

2. Now for our model with just `log_km_driven`:

```
cv_reg_2 <- find_cv("log_selling_price", "log_km_driven", folds, num_folds =
```

3. And for the model with both of these (main) effects:

```
cv_reg_3 <- find_cv("log_selling_price", c("year", "log_km_driven"), folds,
```

We can then use these CV scores to compare across models and choose the 'best' one!

```
c("year" = cv_reg_1, "log_km_driven" = cv_reg_2, "both" = cv_reg_3)
```

```
      year log_km_driven          both
 0.5508152     0.5956502     0.5135431
```

We see that the model that has both predictors has the lowest cross-validation RMSE!

We would now fit this 'best' model on the full data set!

```
best_fit <- lm(log_selling_price ~ year + log_km_driven, data = bike_data)
best_fit
```

```
Call:
lm(formula = log_selling_price ~ year + log_km_driven, data = bike_data)
```

```
Coefficients:
  (Intercept)           year  log_km_driven
   -148.79329        0.08034       -0.22686
```

CV Recap

Nice! We've done a comparison of models on how well they predict on data they are not trained on without splitting our data into a train and test set! This is very useful when we don't have a lot of data.

Again, we would choose our best model (model 3 here) and refit the model on the full data set! We'll see how to do this in the `tidymodels` framework shortly