Cross-Validation

Tucker Morgan - tlm2152

3/20/2022

```
library(tidyverse)
```

Here, I will try to create an algorithm for data partitioning and cross-validation.

```
set.seed(100)
ex_data <-
  data.frame(x = c(rnorm(1000, mean = 0, sd = 10)),
             y = c(rnorm(1000, mean = 5, sd = 1)),
             z = c(rnorm(1000, mean = -5, sd = 10)),
             error = c(rnorm(1000, mean = 0, sd = 10))) %>%
  as_tibble() %>%
  mutate(outcome = x + y + z + error) %>%
  select(-error)
```

```
partition <- function(p, data){
  set.seed(100)
  # generating a probability value
  part_p = runif(nrow(data), min = 0, max = 1)
  # assigning partition id based on probability value
  # parameter p sets proportion of train vs test
  part_id = ifelse(part_p <= p, "train", "test")
  # appending to data set
  data_new = cbind(data, part_id)

  return(data_new)
}
# here training proportion is set to 0.8
part_data <- partition(p = 0.8, data = ex_data)
```

Let's see if this worked properly.

```
part_data %>%
  filter(part_id == "train") %>%
  nrow() / nrow(part_data) # percentage in training
```

```
## [1] 0.781
```

```
part_data %>%
  filter(part_id == "test") %>%
  nrow() / nrow(part_data) # percentage in test
```

```
## [1] 0.219
```

Looks pretty good. There might be a way to better set this up so that we get more consistent splits aligned with our denoted `p`.

Now let's see if we can set up a cross-validation algorithm for the training data. Here, we will want to identify 5 different folds and iterate over the folds.

```
trn_data <-
  part_data %>%
  filter(part_id == "train") %>%
  select(-part_id)

tst_data <- # this will come back into play for test error
  part_data %>%
  filter(part_id == "test") %>%
  select(-part_id)
```

```
set.seed(100)
cv_sets <- function(k = 5, training){
  # again generating a probability value
  fold_p = runif(nrow(training), min = 0, max = 1)

  split_list = list()
  # looping to establish cutoff values for each fold
  for (i in 1:k){
    split_list[[i]] = data.frame("cutoff" = i/k, "fold" = i)
  }
  split_df = as.data.frame(do.call(rbind, split_list))
  # this is some rather manual code to assign each observation to a fold
  # if this function needed to handle varying values of k, this would need to change
  fold_id = case_when(fold_p <= split_df[1,1] ~ split_df[1,2],
                      fold_p <= split_df[2,1] & fold_p > split_df[1,1] ~ split_df[2,2],
                      fold_p <= split_df[3,1] & fold_p > split_df[2,1] ~ split_df[3,2],
                      fold_p <= split_df[4,1] & fold_p > split_df[3,1] ~ split_df[4,2],
                      fold_p <= split_df[5,1] & fold_p > split_df[4,1] ~ split_df[5,2])

  data_new = cbind(training, fold_p, fold_id)

  return(data_new)
}

trn_folds <- cv_sets(training = trn_data)
```

Again, I'll run a few checks to see if this is behaving as intended.

```
folds_check <- list()

for (i in 1:5){
folds_check[[i]] <-
  data.frame(
    "fold" = i,
    "fold_prop" =
      trn_folds %>%
      filter(fold_id == i) %>%
      nrow() / nrow(trn_data))
}

as.data.frame(do.call(rbind, folds_check))
```

```
##   fold fold_prop
## 1    1 0.1613316
## 2    2 0.2343150
## 3    3 0.1779770
## 4    4 0.2202305
## 5    5 0.2061460
```

With a goal of $P(\text{fold}_i) = 0.2$, our results here show some discrepancies. This might be okay, but maybe we could improve somehow?

Next, let's create a cross-validation function that will iterate a target function over each fold.

```
set.seed(100)
# creating a simple example function for testing
ex_func <- function(data){
  lm(outcome ~ ., data = data)
}

ex_func(data = trn_data) # just for example, not stored
```

```
##
## Call:
## lm(formula = outcome ~ ., data = data)
##
## Coefficients:
## (Intercept)            x            y            z
##      3.7535       0.9489       0.2029       0.9835
```

```
cv_function <- function(k = 5, training, func){

  error_list = list()

  for (i in 1:k){
    # this will identify the training set as not i
    trn_set =
      training %>%
      filter(fold_id != i) %>%
      select(-fold_p, -fold_id)
    # and this assigns i to be the test set
    tst_set =
      training %>%
      filter(fold_id == i) %>%
      select(-fold_p, fold_id)
    # fitting our function based on training set
    trn_fit = ex_func(trn_set)
```

```r
    # making predictions based on test set
    cv_pred = predict(trn_fit,
                      newdata = tst_set)
    # creating a data frame for the actuals and predictions
    error_df = data.frame("actual" = tst_set$outcome,
                          "prediction" = cv_pred)
    # calculating RMSE
    cv_rmse = sqrt(mean(error_df$actual - error_df$prediction)^2)

    error_list[[i]] = cv_rmse
  }

  return(error_list)
}

cv_function(training = trn_folds, func = ex_func)
```

```
## [[1]]
## [1] 0.6275944
##
## [[2]]
## [1] 1.821316
##
## [[3]]
## [1] 0.1842442
##
## [[4]]
## [1] 0.2673691
##
## [[5]]
## [1] 1.091111
```

When this function is used with our lasso algorithm, each of the five models will have one lambda value associated with it. There's no tuning parameter for linear regression here. I believe the optimal model, and therefore optimal lambda value, will come from the instance with the lowest RMSE. So we will want to update this algorithm to return the lambda value as well. I think this can be achieved through similar means as in the `cv_sets()` function using `do.call()` and the like.

To run Monte Carlo CV, I believe we can build a function like the one below. The current result will need to be cleaned further, but I think this should perform as intended.

```r
monte_carlo <- function(n, data){
  mc_res <- list()
  for (i in 1:n){
    set.seed(100 * i)

    folded_data = cv_sets(training = data)

    mc_res[[i]] = cv_function(training = folded_data, func = ex_func)
  }

  return(mc_res)
}
```

After getting these results, we choose our optimal lambda and make predictions based on the previously set aside `tst_data` and assess our test error from there.