module4_part1

Andrew Estes

4/6/2022

2)

Create a data set using the same simulation code that you used in Module 1, Part 1, Question 1.

```
#function to create simulated dataframe
sim.data <- function(seed, h=3, s=3){
  set.seed(seed)
  x <- runif(200, min=0, max=6)
  y <- 8 + (x-4)^2 + 4*sin(h*x) + rnorm(200, mean=0, sd=s)
  return(data.frame(x=x, y=y))
}

#creating the dataframe using the above function
df <- sim.data(1234, 5, 3)
```

1)

Review the function you wrote for the Module 1, Part 1, Question 2. Rewrite the code using foreach/%do% loops for both the inner and outer loops.

Since foreach loops return a vector of values by default, you won't have to worry about indexing in the same way you had to do it in Module 1.

Also, note that since you're always running the same number of cross-validations in the inner loop, averaging RMSE's that are returned from the inner loop, then averaging the averages after completing the outer loop should give the same result that you got by doing a single average of all RMSE's in Module 1.

```
#proper foreach utilization
x <- function(method=loess, degree=1, span=0.50, k=10, m=5, ...){
  set.seed(1234)
  foreach(i=1:m, .combine= c) %do% {
    folds <- rep(1:k, each=ceiling(nrow(df)/k))
    folds <- sample(folds, nrow(df))

    foreach(j=1:k, .combine= c) %do% {

      fit <- method(y ~ x, data=df[folds != j, ], degree=1, span=span)

      RMSE(df$y[folds==j], predict(fit, newdata=df[folds==j, ]), na.rm=TRUE)
    }
```

```
  }
}
mean(x(loess, span=.5))
```

```
## [1] 3.820319
```

3)

For a single span value, run your repeated cross-validation code from Module 1, along with your new code from this homework. Report the RMSE from both. If you set the seed to be the same before each run, both functions (old and new) should produce the same RMSE.

```
#Module 1 Code and Results
rmse.df <- function(method=loess, degree=1, span=0.50, k=10, m=5, ...){
  set.seed(1234)
  rmse.vec <- numeric(k*m)
  for(i in 1:m) {
    folds <- rep(1:k, each=ceiling(nrow(df)/k))
    folds <- sample(folds, nrow(df))

    for(j in 1:k) {
      fit <- method(y ~ x, data=df[folds != j, ], degree=1, span=span)

      rmse.vec[j+(k*i)-k] <- RMSE(df$y[folds==j],
                                  predict(fit, newdata=df[folds==j, ]),
                                  na.rm=TRUE)
    }
  }
  return(rmse.vec)
}

mean(rmse.df(loess, span=.5))
```

```
## [1] 3.820319
```

```
#Module 4 Code and Results
x <- function(method=loess, degree=1, span=0.50, k=10, m=5, ...){
  set.seed(1234)
  foreach(i=1:m, .combine= c) %do% {
    folds <- rep(1:k, each=ceiling(nrow(df)/k))
    folds <- sample(folds, nrow(df))

    foreach(j=1:k, .combine= c) %do% {

      fit <- method(y ~ x, data=df[folds != j, ], degree=1, span=span)

      RMSE(df$y[folds==j], predict(fit, newdata=df[folds==j, ]), na.rm=TRUE)
    }
  }
}
mean(x(loess, span=.5))
```

```
## [1] 3.820319
```

4)

Now that you've verified that old and new function do the same thing, write timing code to create a measurement of "elapsed time" for the model tuning (the search for the optimal span) you did in Module 1, Part 1, Question 3. Measure timing for each scenario with the system.time function:

```
n.cores <- detectCores()-1
```

A) Using your "Module 1" repeated cross-validation function

```
a <-
  system.time({
    rmse.df <- function(method=loess, degree=1, span=0.50, k=10, m=5, ...){
      set.seed(1234)
      rmse.vec <- numeric(k*m)
      for(i in 1:m) {
        folds <- rep(1:k, each=ceiling(nrow(df)/k))
        folds <- sample(folds, nrow(df))

        for(j in 1:k) {
          fit <- method(y ~ x, data=df[folds != j, ], degree=1, span=span)

          rmse.vec[j+(k*i)-k] <- RMSE(df$y[folds==j],
                                      predict(fit, newdata=df[folds==j, ]),
                                      na.rm=TRUE)
        }
      }
      return(rmse.vec)
    }

    mean(rmse.df(loess, span=.11))
  })

a
```

```
##    user  system elapsed
##    0.06    0.00    0.06
```

B) using the new foreach/%do% repeated cross-validation function from this homework assignment

```
b <-
  system.time({
    registerDoParallel(n.cores)
    cl <- makeCluster(n.cores)
    x <- function(method=loess, degree=1, span=0.50, k=10, m=5, ...){
      set.seed(1234)
      foreach(i=1:m, .combine= c) %do% {
        folds <- rep(1:k, each=ceiling(nrow(df)/k))
        folds <- sample(folds, nrow(df))

        foreach(j=1:k, .combine= c) %do% {

          fit <- method(y ~ x, data=df[folds != j, ], degree=1, span=span)

          RMSE(df$y[folds==j], predict(fit, newdata=df[folds==j, ]), na.rm=TRUE)
        }
      }
    }
    mean(x(loess, span=.5))
    stopCluster(cl)
  })

b
```

```
##    user  system elapsed
##    0.11    0.14    0.87
```

C) using your new function, but replacing the outer foreach/%do% with foreach/%dopar% (in other words parallelizing only the outer loop)

```
registerDoParallel(7)

c <- function(df, method=loess, degree=1, span=0.50, k=10, m=5, ...){
  set.seed(1234)
  foreach(i=1:m, .combine= c, .packages="foreach") %dopar% {
    folds <- rep(1:k, each=ceiling(nrow(df)/k))
    folds <- sample(folds, nrow(df))

    foreach(j=1:k, .combine= c, .packages="foreach") %do% {
      fit <- method(y ~ x, data=df[folds != j, ], degree=1, span=span)
      library(Metrics)
      rmse(df$y[folds==j], predict(fit, newdata=df[folds==j, ]))
    }
  }
}

c <- system.time({
  mean(c(df=df, loess, span=.5, na.rm=TRUE), na.rm=TRUE)
})
```

```
## error calling combine function:
## <simpleError in {    folds <- rep(1:k, each = ceiling(nrow(df)/k))    folds <- sample(folds, nrow(df)

## Warning in mean.default(c(df = df, loess, span = 0.5, na.rm = TRUE), na.rm =
## TRUE): argument is not numeric or logical: returning NA
```

```
c
```

```
##    user  system elapsed
##    0.03    0.00    0.14
```

D) using your new function, but replacing the inner foreach/%do% with foreach/%dopar% (in other words parallelizing only the inner loop)

```
registerDoParallel(7)

d <- function(df, method=loess, degree=1, span=0.50, k=10, m=5, ...){
  set.seed(1234)
  foreach(i=1:m, .combine= c, .packages="foreach") %do% {
    folds <- rep(1:k, each=ceiling(nrow(df)/k))
    folds <- sample(folds, nrow(df))

    foreach(j=1:k, .combine= c, .packages="foreach") %dopar% {
      fit <- method(y ~ x, data=df[folds != j, ], degree=1, span=span)
      library(Metrics)
      rmse(df$y[folds==j], predict(fit, newdata=df[folds==j, ]))
    }
  }
}

d <- system.time({
  mean(d(df=df, loess, span=.5, na.rm=TRUE), na.rm=TRUE)
})

d
```

```
##    user  system elapsed
##    0.05    0.02    0.20
```

E) using your new function, but replacing the both inner and outer foreach/%do% with foreach/%dopar% (in other words parallelizing both loops)

```
registerDoParallel(7)

e <- function(df, method=loess, degree=1, span=0.50, k=10, m=5, ...){
  set.seed(1234)
  foreach(i=1:m, .combine= c, .packages="foreach") %dopar% {
    folds <- rep(1:k, each=ceiling(nrow(df)/k))
    folds <- sample(folds, nrow(df))

    foreach(j=1:k, .combine= c, .packages="foreach") %dopar% {
      fit <- method(y ~ x, data=df[folds != j, ], degree=1, span=span)
      library(Metrics)
      rmse(df$y[folds==j], predict(fit, newdata=df[folds==j, ]))
    }
  }
}

e <- system.time({
  mean(e(df=df, loess, span=.5, na.rm=TRUE), na.rm=TRUE)
})
e
```

```
##    user  system elapsed
##    0.02    0.03    0.17
```

Note: Rather than edit the same code to change the %do% to %dopar%, create five different functions, one for each part above, so that I can see all the code you used in each. Also, don't forget to use the registerDoParallel command before running your %dopar% versions, so that they actually will use multiple cores.

5)

Make a table of the timing for each version you did above. Then, write a short explanation of what you saw in the timings. Does the pattern in timings make sense, given what you know about the overhead involved in initiating a multi-processor calculation? [Hint: Think about the structure of the loops and how many time a multi-processor computation is initiated in each case.]

```
a.table <- matrix(a)
b.table <- matrix(b)
c.table <- matrix(c)
d.table <- matrix(d)
e.table <- matrix(e)


combined.table <- cbind(a.table, b.table, c.table, d.table, e.table)
kable(combined.table)
```

0.06	0.11	0.03	0.05	0.02
0.00	0.14	0.00	0.02	0.03
0.06	0.87	0.14	0.20	0.17
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA

Yes, the extra speed gained by utilizing parallelization is offset by the time cost to initiate the multi-processor setup. The only instance where it comes comparatively close to the initial function is when both loops are using %dopar%. I agree with the github post below where it makes more sense to optimize the code than to try to parallelize the functions. https://privefl.github.io/blog/a-guide-to-parallelism-in-r/