Homework 5

YutingMei

March 04, 2022

```r
library('MASS') ## for 'mcycle'
library('dplyr')
```

```
##
## Attaching package: 'dplyr'

## The following object is masked from 'package:MASS':
##
##     select

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library('manipulate') ## for 'manipulate'
library(rlang)
library(caret)
```

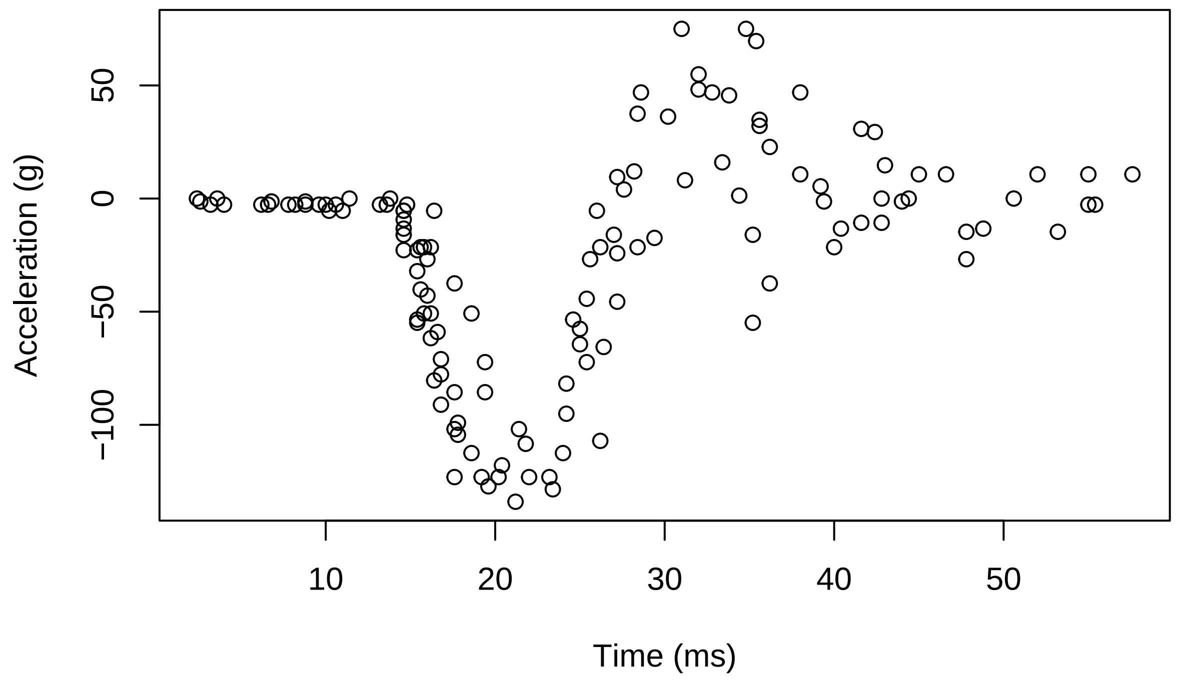
```
## Loading required package: ggplot2

## Loading required package: lattice
```

- Randomly split the mcycle data into training (75%) and validation (25%) subsets.

```r
y <- mcycle$accel
x <- matrix(mcycle$times, length(mcycle$times), 1)
```

```r
plot(x, y, xlab="Time (ms)", ylab="Acceleration (g)")
```



```r
dt = cbind(x,y)
```

```r
train_index = sample.int(nrow(dt), nrow(dt) * .75, replace = F)
index_list = seq(nrow(dt))
valid_index = index_list[!index_list %in% train_index]
train = dt[train_index, ]
valid = dt[valid_index, ]
```

- Using the mcycle data, consider predicting the mean acceleration as a function of time. Use the Nadaraya-Watson method with the k-NN kernel function to create a series of prediction models by varying the tuning parameter over a sequence of values. (hint: the script already implements this)

```r
## k-NN kernel function
## x  - n x p matrix of training inputs
## x0 - 1 x p input where to make prediction
## k  - number of nearest neighbors
kernel_k_nearest_neighbors <- function(x, x0, k) {
  ## compute distance betwen each x and x0
  z <- t(t(x) - x0)
  d <- sqrt(rowSums(z*z))

  ## initialize kernel weights to zero
  w <- rep(0, length(d))

  ## set weight to 1 for k nearest neighbors
  w[order(d)[1:k]] <- 1

  return(w)
}
```

```r
## Make predictions using the NW method
## y  - n x 1 vector of training outputs
## x  - n x p matrix of training inputs
## x0 - m x p matrix where to make predictions
## kern  - kernel function to use
## ... - arguments to pass to kernel function
nadaraya_watson <- function(y, x, x0, kern, ...) {
  k <- t(apply(x0, 1, function(x0_) {
    k_ <- kern(x, x0_, ...)
    k_/sum(k_)
  }))
  yhat <- drop(k %*% y)
  attr(yhat, 'k') <- k
  return(yhat)
}
```

```r
## Compute effective df using NW method
## y  - n x 1 vector of training outputs
## x  - n x p matrix of training inputs
## kern  - kernel function to use
## ... - arguments to pass to kernel function
effective_df <- function(y, x, kern, ...) {
  y_hat <- nadaraya_watson(y, x, x,
    kern=kern, ...)
  sum(diag(attr(y_hat, 'k')))
}
```

```r
## loss function
## y    - train/test y
## yhat - predictions at train/test x
loss_squared_error <- function(y, yhat)
  (y - yhat)^2

## test/train error
## y    - train/test y
## yhat - predictions at train/test x
## loss - loss function
error <- function(y, yhat, loss=loss_squared_error)
  mean(loss(y, yhat))

## AIC
## y    - training y
## yhat - predictions at training x
## d    - effective degrees of freedom
aic <- function(y, yhat, d)
  error(y, yhat) + 2/length(y)*d

## BIC
## y    - training y
## yhat - predictions at training x
## d    - effective degrees of freedom
bic <- function(y, yhat, d)
  error(y, yhat) + log(length(y))/length(y)*d
```

```r
options(warn=-1)
## how does k affect shape of predictor and eff. df using k-nn kernel ?
# manipulate({
#   ## make predictions using NW method at training inputs
#   y_hat <- nadaraya_watson(y, x, x,
#     kern=kernel_k_nearest_neighbors, k=k_slider)
#   edf <- effective_df(y, x,
#     kern=kernel_k_nearest_neighbors, k=k_slider)
#   aic_ <- aic(y, y_hat, edf)
#   bic_ <- bic(y, y_hat, edf)
#   y_hat_plot <- nadaraya_watson(y, x, x_plot,
#     kern=kernel_k_nearest_neighbors, k=k_slider)
#   plot(x, y, xlab="Time (ms)", ylab="Acceleration (g)")
#   legend('topright', legend = c(
#     paste0('eff. df = ', round(edf,1)),
#     paste0('aic = ', round(aic_, 1)),
#     paste0('bic = ', round(bic_, 1))),
#     bty='n')
#   lines(x_plot, y_hat_plot, col="#882255", lwd=2)
# }, k_slider=slider(1, 15, initial=3, step=1))
```

- With the squared-error loss function, compute and plot the training error, AIC, BIC, and validation error (using the validation data) as functions of the tuning parameter.

```r
error_combine = function(k_seq, y, x){
  y = matrix(y)
  x = matrix(x)
  aic_ = c()
  bic_ = c()
  error_ = c()
  for (i in k_seq){
  y_hat <- nadaraya_watson(y, x, x,
    kern=kernel_k_nearest_neighbors, k=i)
  edf <- effective_df(y, x,
    kern=kernel_k_nearest_neighbors, k=i)
  error_ = append(error_, error(y, y_hat))
  aic_ <- append(aic_, aic(y, y_hat, edf))
  bic_ <- append(bic_, bic(y, y_hat, edf))
  }
  data.frame(k = k_seq, aic = aic_, bic = bic_, error = error_)
}
```

```r
# train error
error_combine(seq(1,20), train[,2], train[,1])
```

```
##     k      aic      bic    error
## 1   1 293.5860 295.5520 292.0708
## 2   2 302.9697 304.2017 302.0202
## 3   3 322.2885 323.1448 321.6286
## 4   4 369.2359 369.8847 368.7359
## 5   5 363.1398 363.6588 362.7398
## 6   6 415.7653 416.1978 415.4319
## 7   7 419.2733 419.6440 418.9876
## 8   8 427.5155 427.8399 427.2655
## 9   9 470.2718 470.5602 470.0496
## 10 10 475.0152 475.2747 474.8152
## 11 11 502.3939 502.6298 502.2121
## 12 12 488.7541 488.9703 488.5874
## 13 13 513.8828 514.0824 513.7289
## 14 14 545.0598 545.2452 544.9169
## 15 15 563.8486 564.0216 563.7153
## 16 16 584.6833 584.8454 584.5583
## 17 17 575.6361 575.7888 575.5185
## 18 18 591.1786 591.3228 591.0675
## 19 19 608.1407 608.2773 608.0355
## 20 20 603.2211 603.3509 603.1211
```

```
# test error
error_combine(seq(1,20), valid[,2], valid[,1])
```

```
##     k         aic         bic     error
## 1   1    7.532353    8.968928    5.6500
## 2   2  403.262426  404.025607  402.2624
## 3   3  407.027353  407.536140  406.3607
## 4   4  409.259577  409.641167  408.7596
## 5   5  470.807412  471.112684  470.4074
## 6   6  564.883693  565.138086  564.5504
## 7   7  698.587743  698.805795  698.3020
## 8   8  770.311861  770.502656  770.0619
## 9   9  909.573500  909.743096  909.3513
## 10 10  979.040324  979.192960  978.8403
## 11 11  960.626490  960.765250  960.4447
## 12 12 1050.036315 1050.163512 1049.8696
## 13 13 1107.749539 1107.866951 1107.5957
## 14 14 1248.844110 1248.953136 1248.7013
## 15 15 1515.481401 1515.583159 1515.3481
## 16 16 1760.386490 1760.481888 1760.2615
## 17 17 2006.523808 2006.613594 2006.4062
## 18 18 2246.982703 2247.067501 2246.8716
## 19 19 2422.974197 2423.054531 2422.8689
## 20 20 2630.564568 2630.640886 2630.4646
```

- For each value of the tuning parameter, Perform 5-fold cross-validation using the combined training and validation data. This results in 5 estimates of test error per tuning parameter value.

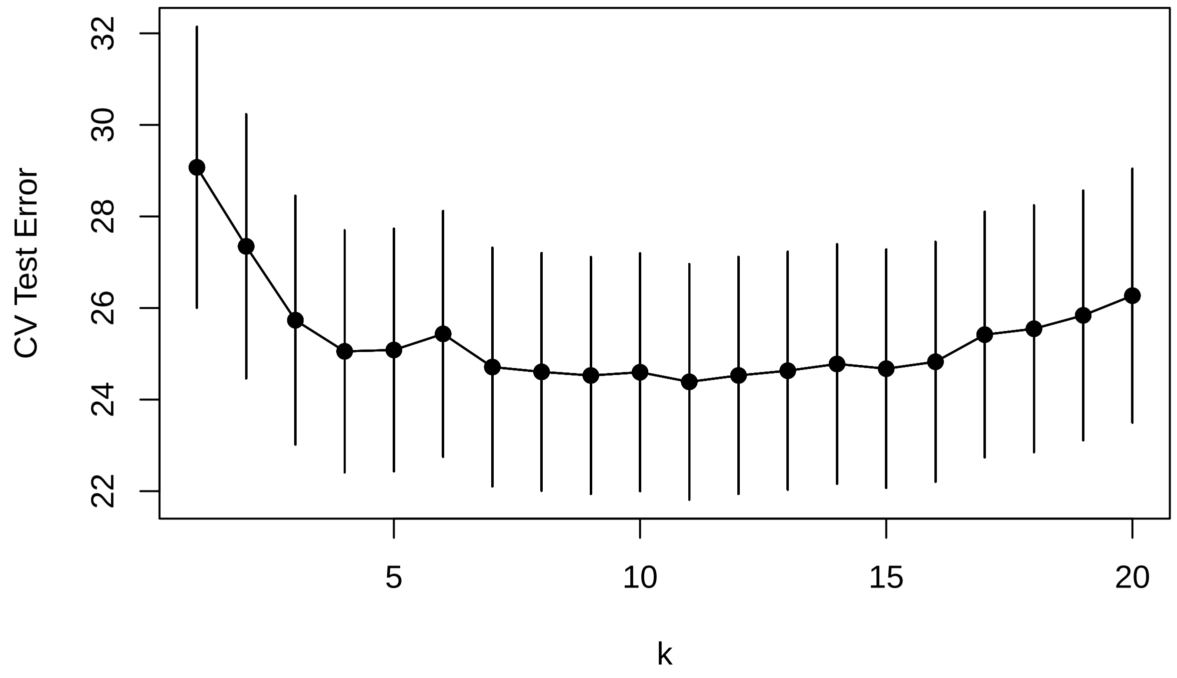
```
train.control <- trainControl(method = "cv", number = 5)
train_error_cv = train(y ~ .,
      method = 'knn',
      tuneGrid = expand.grid(k = 1:20),
      trControl = train.control,
      data = rbind(train, valid) %>% data.frame())
train_error_cv
```

```
## k-Nearest Neighbors
##
## 133 samples
##   1 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 106, 106, 107, 106, 107
## Resampling results across tuning parameters:
##
##   k   RMSE      Rsquared   MAE
##    1  29.07268  0.6566387  20.58189
##    2  27.34744  0.6904227  19.28653
##    3  25.73375  0.7171636  17.67392
##    4  25.05367  0.7306719  17.79038
##    5  25.08336  0.7306442  18.17646
##    6  25.43539  0.7225952  18.74183
##    7  24.71088  0.7368381  18.18798
##    8  24.60541  0.7427416  18.23554
##    9  24.52671  0.7440385  18.22645
##   10  24.59730  0.7412267  18.54510
##   11  24.38768  0.7479871  18.28943
##   12  24.52765  0.7474382  18.45783
##   13  24.63070  0.7470389  18.50467
##   14  24.77817  0.7443419  18.64150
##   15  24.67521  0.7464947  18.59702
##   16  24.82594  0.7456683  18.81034
##   17  25.41827  0.7328213  19.27880
##   18  25.54719  0.7348632  19.74214
##   19  25.83972  0.7307408  20.05332
##   20  26.27027  0.7235733  20.59050
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was k = 11.
```

- Plot the CV-estimated test error (average of the five estimates from each fold) as a function of the tuning parameter. Add vertical line segments to the figure (using the segments function in R) that represent one "standard error" of the CV-estimated test error (standard deviation of the five estimates from each fold).

```
cv_all = train_error_cv$results %>%
  mutate(rmse_low = RMSE - 2*(RMSE / sqrt(train_error_cv$control$number)),
         rmse_high = RMSE - 2*(RMSE / sqrt(train_error_cv$control$number)))
```

```
plot(x=range(as.matrix((cv_all$k))),
     y=range(range(cv_all$RMSE - cv_all$rmse_low, cv_all$RMSE + cv_all$rmse_high)),
     type='n',
     xlab='k',
     ylab='CV Test Error')
for(i in 1:nrow(cv_all)) {
  points(x=cv_all$k, y=cv_all$RMSE, pch=19, col='#00000055')
  lines(x=cv_all$k, y=cv_all$RMSE, col='#00000055')
  segments(x0 = cv_all$k, y0 = cv_all$RMSE - cv_all$rmse_low, x1 = cv_all$k, y1 = cv_all$rmse_high + cv_
}
```



- Interpret the resulting figures and select a suitable value for the tuning parameter.
- When the k increases, the error become smaller for k from 1 to 15, but when k stil increases, the test error become greater again. k = 3 or 4 is suitable by one-standard-error rule.