hw5

Tinglei Wu

3/2/2022

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

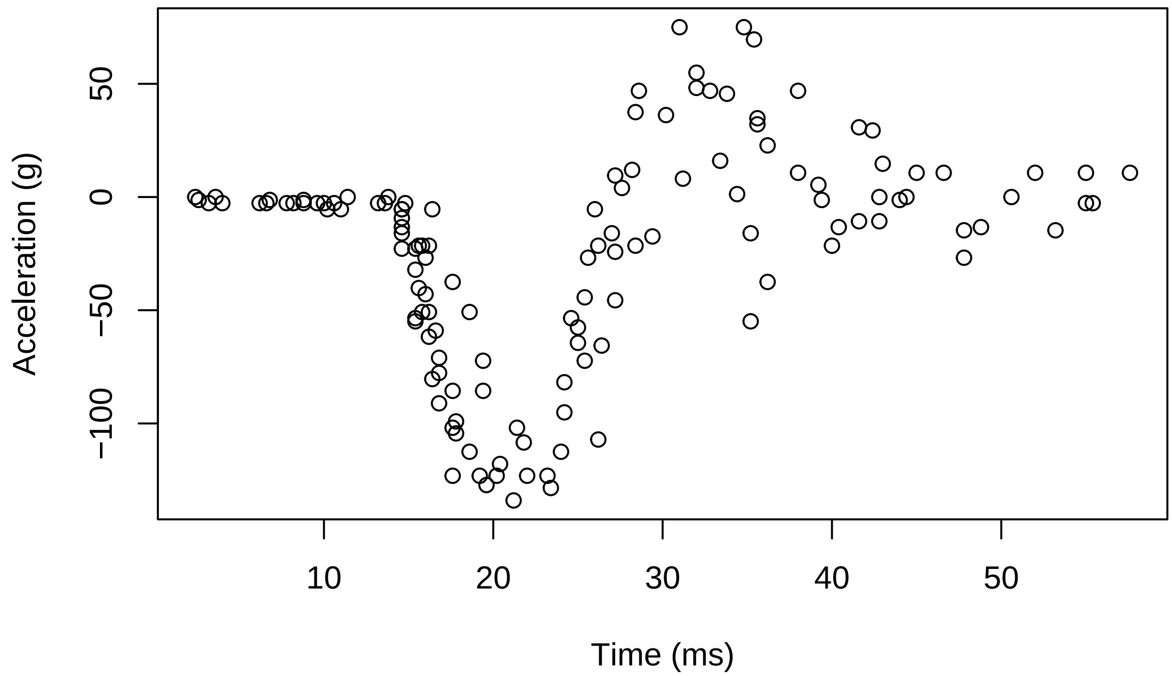
```
##
## Attaching package: 'dplyr'

## 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(caret)
```

```
## Loading required package: ggplot2

## Loading required package: lattice
```

```r
library("knitr")
```

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

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



Quetion 1: Randomly split the mcycle data into training (75%) and validation (25%) subsets

```
train_data<-sample_frac(mcycle, 0.75)
sid<-as.numeric(rownames(train_data))
test_data<-mcycle[-sid,]

sid
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

```
test_data
```

```
##     times accel
## 101  35.2 -16.0
## 102  35.2 -54.9
## 103  35.4  69.6
## 104  35.6  34.8
## 105  35.6  32.1
## 106  36.2 -37.5
## 107  36.2  22.8
## 108  38.0  46.9
## 109  38.0  10.7
## 110  39.2   5.4
```

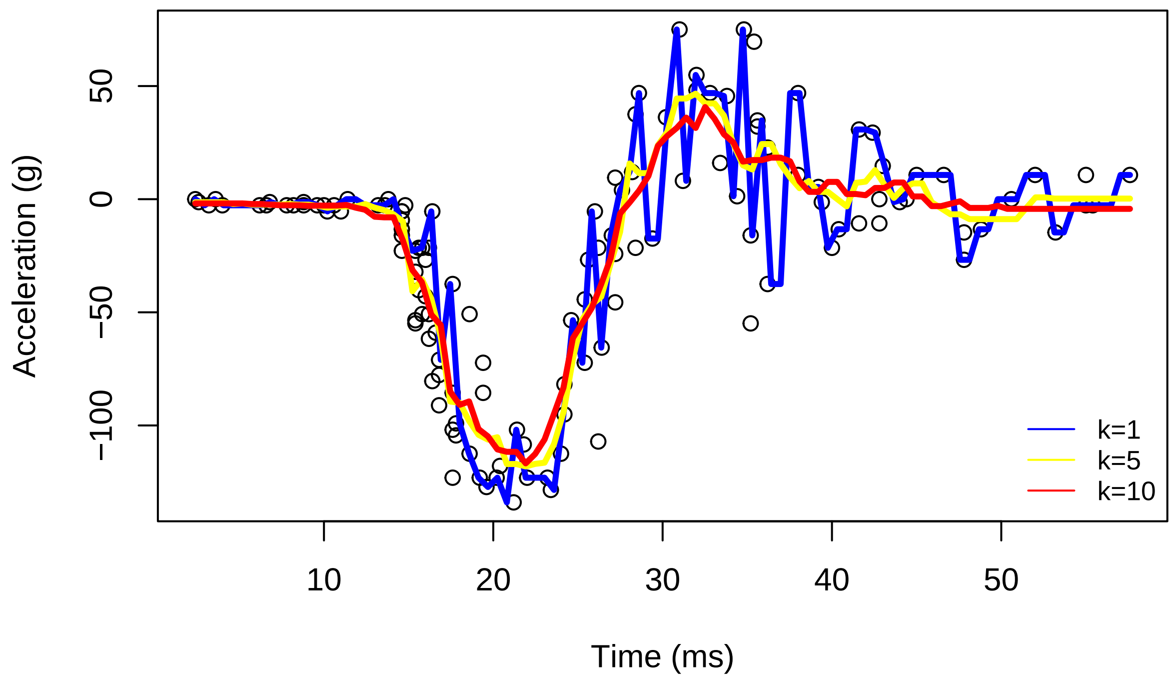
```
## 111  39.4  -1.3
## 112  40.0 -21.5
## 113  40.4 -13.3
## 114  41.6  30.8
## 115  41.6 -10.7
## 116  42.4  29.4
## 117  42.8   0.0
## 118  42.8 -10.7
## 119  43.0  14.7
## 120  44.0  -1.3
## 121  44.4   0.0
## 122  45.0  10.7
## 123  46.6  10.7
## 124  47.8 -26.8
## 125  47.8 -14.7
## 126  48.8 -13.3
## 127  50.6   0.0
## 128  52.0  10.7
## 129  53.2 -14.7
## 130  55.0  -2.7
## 131  55.0  10.7
## 132  55.4  -2.7
## 133  57.6  10.7
```

Question 2: Using the mcycle data, consider predicting the mean acceleration as a function of time

```r
## Epanechnikov kernel function
## x  - n x p matrix of training inputs
## x0 - 1 x p input where to make prediction
## lambda - bandwidth (neighborhood size)
kernel_epanechnikov <- function(x, x0, lambda=1) {
  d <- function(t)
    ifelse(t <= 1, 3/4*(1-t^2), 0)
  z <- t(t(x) - x0)
  d(sqrt(rowSums(z*z))/lambda)
}

## 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=1) {
  ## 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
```

```r
  return(w)
}

## 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)
}
## Helper function to view kernel (smoother) matrix
matrix_image <- function(x) {
  rot <- function(x) t(apply(x, 2, rev))
  cls <- rev(gray.colors(20, end=1))
  image(rot(x), col=cls, axes=FALSE)
  xlb <- pretty(1:ncol(x))
  xat <- (xlb-0.5)/ncol(x)
  ylb <- pretty(1:nrow(x))
  yat <- (ylb-0.5)/nrow(x)
  axis(3, at=xat, labels=xlb)
  axis(2, at=yat, labels=ylb)
  mtext('Rows', 2, 3)
  mtext('Columns', 3, 3)
}

## 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')))
}

## create a grid of inputs
x_plot <- matrix(seq(min(x),max(x),length.out=100),100,1)

## make predictions using NW method for k=1, k=10 and k=20
y_hat_plot1 <- nadaraya_watson(y, x, x_plot, kern=kernel_k_nearest_neighbors, k=1)
y_hat_plot2 <- nadaraya_watson(y, x, x_plot, kern=kernel_k_nearest_neighbors, k=5)
y_hat_plot3<- nadaraya_watson(y, x, x_plot, kern=kernel_k_nearest_neighbors, k=10)

## plot predictions
```

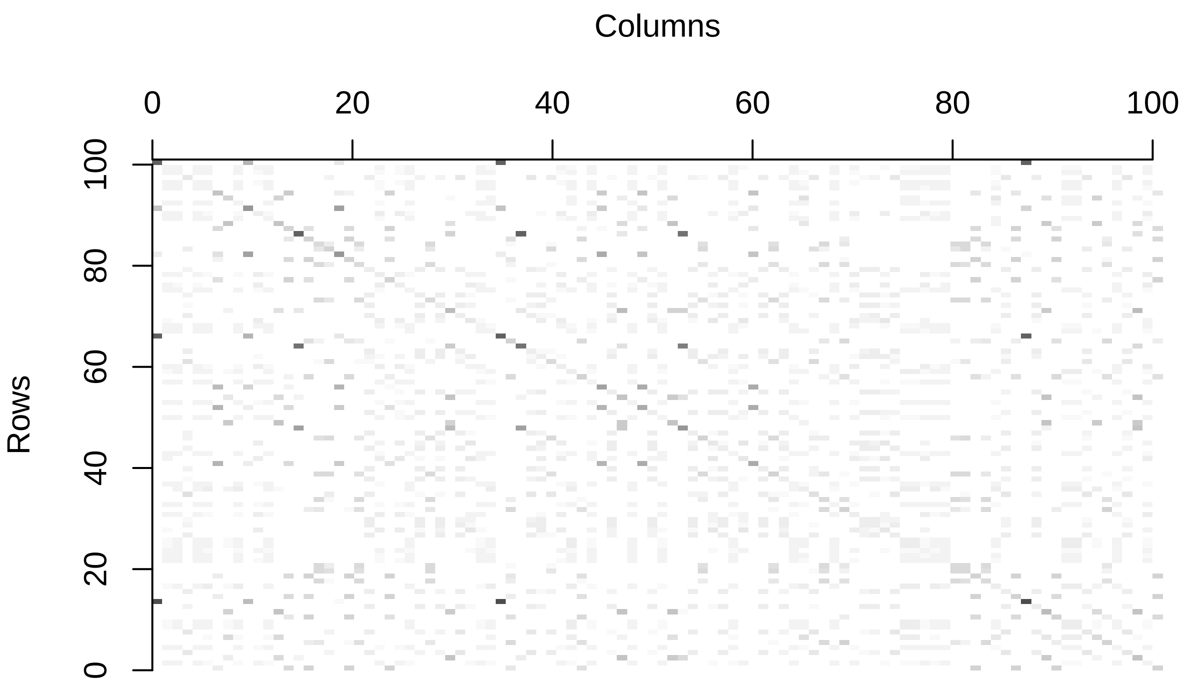
```r
plot(x, y, xlab="Time (ms)", ylab="Acceleration (g)")
lines(x_plot, y_hat_plot1, col="blue", lwd=3)
lines(x_plot, y_hat_plot2, col="yellow", lwd=3)
lines(x_plot, y_hat_plot3, col="red", lwd=3)
legend('bottomright', c('k=1', 'k=5', 'k=10'), cex=0.8, col=c('blue', 'yellow', 'red'), bty='n', lty=1)
```



Question 3: With the squared-error loss function, compute and plot the training error, AIC, BIC, and validation error

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

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

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

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

y <- train_data$accel
x <- matrix(train_data$times, length(train_data$times), 1)

## make predictions using NW method at training inputs
y_hat <- nadaraya_watson(y, x, x,
  kernel_epanechnikov, lambda=5)

## view kernel (smoother) matrix
matrix_image(attr(y_hat, 'k'))
```




```
## compute effective degrees of freedom
edf <- effective_df(y, x, kernel_epanechnikov, lambda=5)


## create a grid of inputs
x_plot <- matrix(seq(min(x),max(x),length.out=100),100,1)

## make predictions using NW method at each of grid points
y_hat_plot <- nadaraya_watson(y, x, x_plot,
  kernel_epanechnikov, lambda=1)
```

```
# Training Error
error(y, y_hat)
```

```
## [1] 743.8526
```

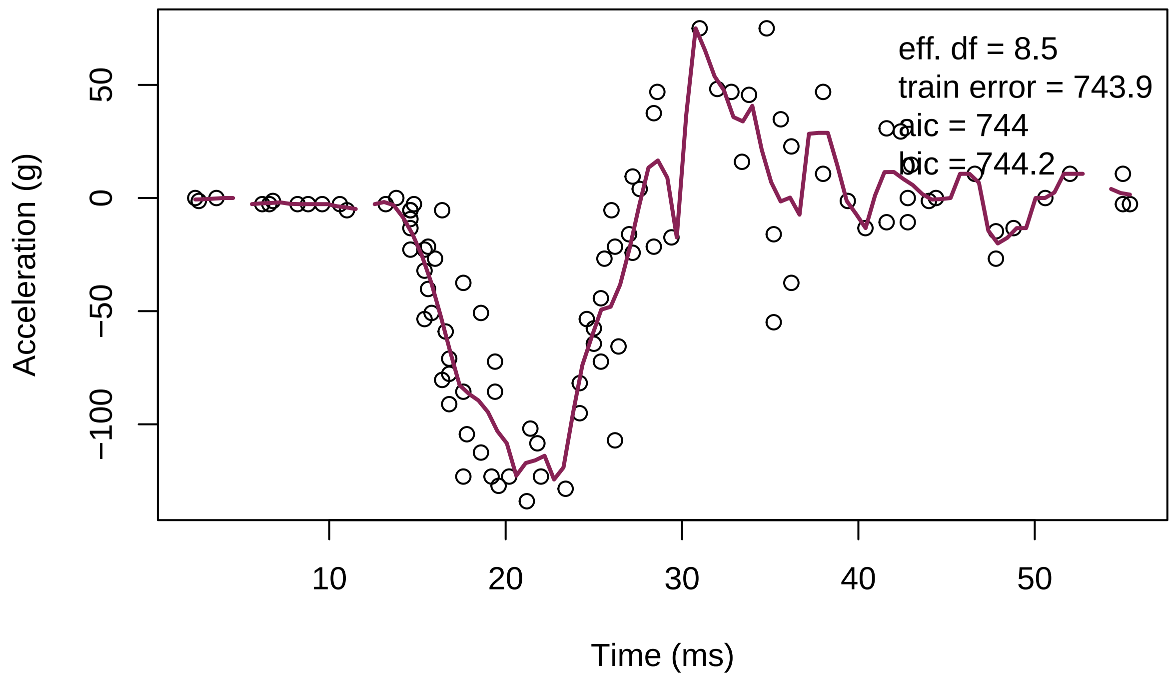
```r
# AIC
aic(y, y_hat, edf)
```

```
## [1] 744.0228
```

```r
# BIC
bic(y, y_hat, edf)
```

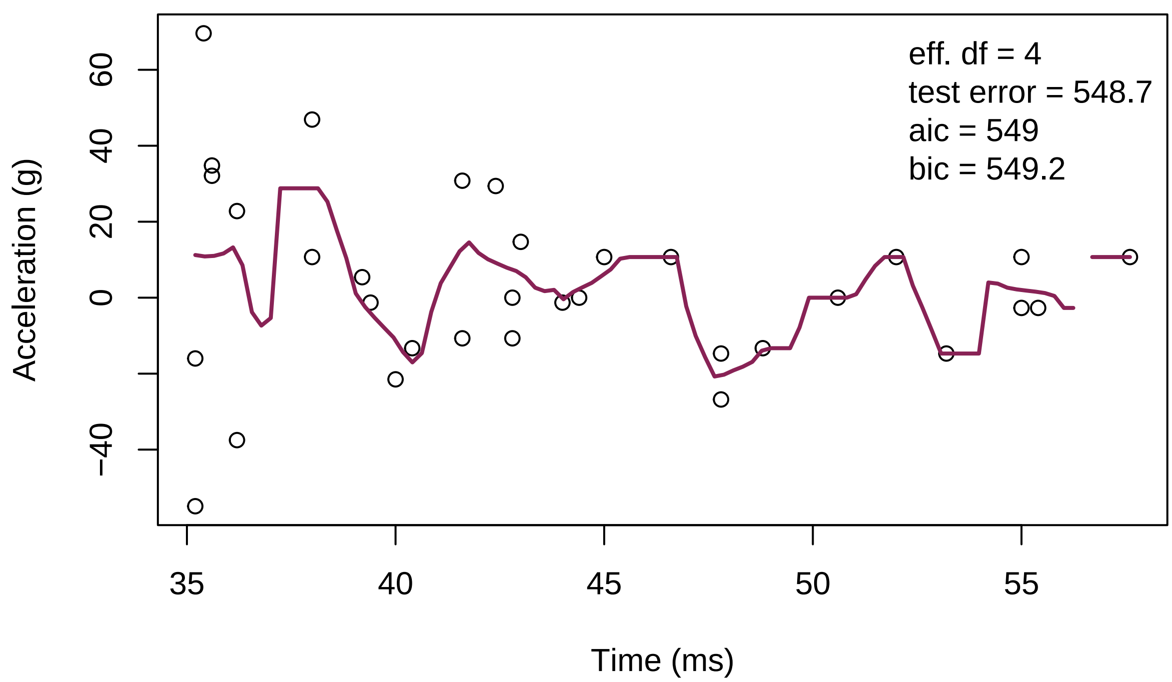
```
## [1] 744.2445
```

```r
err<-error(y, y_hat)
aic_ <- aic(y, y_hat, edf)
bic_ <- bic(y, y_hat, edf)
plot(x, y, xlab="Time (ms)", ylab="Acceleration (g)")
   legend('topright', legend = c(
     paste0('eff. df = ', round(edf,1)),
     paste0('train error = ', round(err, 1)),
     paste0('aic = ', round(aic_, 1)),
     paste0('bic = ', round(bic_, 1))),
     bty='n')
lines(x_plot, y_hat_plot, col="#882255", lwd=2)
```




```r
ky <- test_data$accel
kx <- matrix(test_data$times, length(test_data$times), 1)

## make predictions using NW method at testing inputs
ky_hat <- nadaraya_watson(ky, kx, kx,
  kernel_epanechnikov, lambda=5)
```

```r
## compute effective degrees of freedom
# tedf <- effective_df(ty, tx, kernel_epanechnikov, lambda=5)


## create a grid of inputs
kx_plot <- matrix(seq(min(kx),max(kx),length.out=100),100,1)

## make predictions using NW method at each of grid points
ky_hat_plot <- nadaraya_watson(ky, kx, kx_plot,
  kernel_epanechnikov, lambda=1)

# Validation Error
err1<-error(ky, ky_hat)
kedf <- effective_df(ky, kx, kernel_epanechnikov, lambda=5)
# AIC
aic_<-aic(ky, ky_hat, kedf)
# BIC
bic_<-bic(ky, ky_hat, kedf)
## plot predictions
plot(kx, ky, xlab="Time (ms)", ylab="Acceleration (g)")
lines(kx_plot, ky_hat_plot, col="#882255", lwd=2)
legend('topright', legend = c(
    paste0('eff. df = ', round(kedf,1)),
     paste0('test error = ', round(err1, 1)),
     paste0('aic = ', round(aic_, 1)),
     paste0('bic = ', round(bic_, 1))),
     bty='n')
```



Question 4: 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

```
set.seed(15)
mcycle_flds  <- createFolds(mcycle$accel, k=5)
print(mcycle_flds)
```

```
## $Fold1
##  [1]   5   7   8  10  24  25  26  34  37  40  41  44  56  60  64  67  85  88  89
## [20]  92  96 101 102 106 126 130 131
##
## $Fold2
##  [1]   4  13  15  16  21  27  30  33  39  46  49  54  63  70  72  73  76  86  95
## [20]  97  98 104 115 119 120 125
##
## $Fold3
##  [1]   9  12  17  20  22  29  31  32  42  47  58  61  62  68  79  80  81  83  91
## [20]  94  99 103 111 122 123 129 132
##
## $Fold4
##  [1]   1  18  19  23  35  36  52  53  55  59  69  71  74  75  78 107 108 110 112
## [20] 114 118 121 124 127 128 133
##
## $Fold5
##  [1]   2   3   6  11  14  28  38  43  45  48  50  51  57  65  66  77  82  84  87
## [20]  90  93 100 105 109 113 116 117
```

```
sapply(mcycle_flds, length)
```

```
## Fold1 Fold2 Fold3 Fold4 Fold5
##    27    26    27    26    27
```

```
cvknnreg_mcycle <- function(kNN = 10, flds=mcycle_flds) {
  cverr <- rep(NA, length(flds))
  for(tst_idx in 1:length(flds)) { ## for each fold

    ## get training and testing data
    mcycle_trn <- mcycle[-flds[[tst_idx]],]
    mcycle_tst <- mcycle[ flds[[tst_idx]],]

    ## fit kNN model to training data
    knn_fit <- knnreg(accel ~ times,
                      k=kNN, data=mcycle_trn)

    ## compute test error on testing data
    pre_tst <- predict(knn_fit, mcycle_tst)
    cverr[tst_idx] <- mean((mcycle_tst$accel - pre_tst)^2)
  }
  return(cverr)
}
```

Question 5: 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

```r
cverrs <- sapply(1:40, cvknnreg_mcycle)
print(cverrs)
```

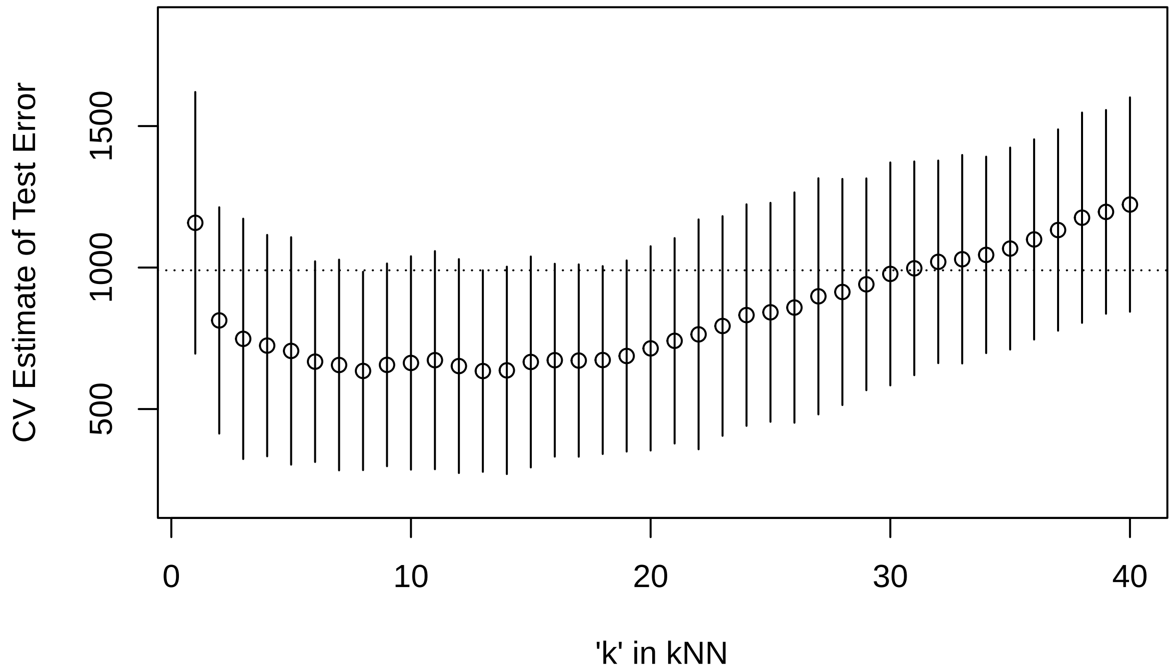
```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 1714.2741 1225.5741 1194.6940 1288.9915 1276.1731 1171.3820 1184.8180
## [2,]  519.1238  269.7850  185.5025  268.1531  260.6804  233.1877  202.9172
## [3,] 1110.6328  686.9212  556.4522  522.7368  439.1764  467.2601  467.2058
## [4,]  971.6179  691.7369  655.7638  628.8749  631.2647  656.4100  594.0081
## [5,] 1474.8920 1192.2785 1148.2494  912.7228  920.2319  809.7640  829.7366
##           [,8]      [,9]     [,10]     [,11]     [,12]     [,13]     [,14]
## [1,] 1128.8391 1179.7252 1247.7534 1279.9892 1258.2631 1193.1744 1218.2526
## [2,]  182.1323  217.6144  243.7948  244.6061  234.0310  220.4289  215.2597
## [3,]  479.3272  485.8238  463.7354  478.1474  484.8418  487.1394  500.6296
## [4,]  615.3839  614.8452  606.3356  635.2262  618.5401  605.0913  588.8028
## [5,]  767.6451  782.9823  753.1773  725.9601  663.8392  665.5412  660.8071
##          [,15]     [,16]     [,17]     [,18]     [,19]     [,20]     [,21]
## [1,] 1255.8161 1212.6865 1209.2418 1203.7611 1232.5108 1296.6235 1328.4156
## [2,]  238.0875  282.9021  274.1647  291.1334  301.3908  301.0971  328.9239
## [3,]  532.4641  537.3451  551.2119  562.6586  600.5497  626.4130  648.1937
## [4,]  596.4829  629.4134  654.5002  641.6799  652.3628  660.9103  690.6735
## [5,]  709.5163  701.0754  668.0268  666.7256  651.1609  687.0429  709.9031
##          [,22]     [,23]     [,24]     [,25]     [,26]     [,27]     [,28]
## [1,] 1444.9842 1447.9111 1502.9871 1507.2693 1566.1316 1613.5059 1587.4674
## [2,]  355.1384  410.8127  473.6967  493.8207  520.0545  529.1936  522.9638
## [3,]  624.9934  663.8973  717.2165  711.4466  706.8416  759.5581  792.9546
## [4,]  676.5652  699.8368  704.9470  731.6596  730.1856  731.0626  787.2829
## [5,]  718.4453  745.1866  761.6988  765.3411  770.3431  858.9445  877.8209
##          [,29]     [,30]     [,31]     [,32]     [,33]     [,34]     [,35]
## [1,] 1573.1530 1650.8496 1638.8546 1627.6508 1655.7542 1629.5808 1676.0561
## [2,]  577.2849  620.6617  640.5730  689.7478  689.2437  722.9051  774.2112
## [3,]  810.5431  825.8785  868.3449  890.5762  936.0847  956.0453  987.5866
## [4,]  841.6003  853.8440  882.1799  889.1085  871.2455  876.1727  852.2868
## [5,]  901.5220  936.6488  956.6900 1003.9637  994.9708 1039.3533 1045.6052
##          [,36]    [,37]     [,38]     [,39]     [,40]
## [1,] 1701.0613 1726.500 1792.8060 1806.3105 1853.1714
## [2,]  796.6132  807.597  841.9857  916.4554  931.1676
## [3,] 1044.1188 1087.715 1162.1688 1159.4262 1180.7908
## [4,]  892.8485  918.675  932.1466  940.5811  922.9947
## [5,] 1062.4498 1122.771 1151.9358 1161.3120 1225.4667
```

```r
cverrs_mean <- apply(cverrs, 2, mean)
cverrs_sd   <- apply(cverrs, 2, sd)

plot(x=1:40, y=cverrs_mean,
     ylim=range(cverrs),
     xlab="'k' in kNN", ylab="CV Estimate of Test Error")
segments(x0=1:40, x1=1:40,
```

```r
         y0=cverrs_mean-cverrs_sd,
         y1=cverrs_mean+cverrs_sd)
best_idx <- which.min(cverrs_mean)
points(x=best_idx, y=cverrs_mean[best_idx], pch=30)
```

```
## Warning in plot.xy(xy.coords(x, y), type = type, ...): unimplemented pch value
## '30'
```

```r
abline(h=cverrs_mean[best_idx] + cverrs_sd[best_idx], lty=3)
```



Question 6: Interpret the resulting figures and select a suitable value for the tuning parameter.

It is reasonable for the tuning parameter to be 30, k=30