Stat 432 Homework 7

Assigned: Oct 4, 2021; Due: 11:59 PM CT, Oct 12, 2021

Contents

Question 1: Writing your own KNN and Kernel Regression

For this question, you are not allowed to use existing functions that directly calculate KNN or kernel regression, but you can still use any R function to calculate the components you need. For an example of such implementation of NW kernel regression, read the lecture note this week. The end result of this question is to write two functions: `myknn(x0, x, y, k)` and `mynw(x0, x, y, h)` that calculates the prediction of these two models at a target point `x0`, given the data vectors `x` and `y`, and the corresponding tuning parameters `k` and `h`. For `mynw()`, you should use the Gaussian kernel function. An additional requirement is that you **cannot use for-loop** in your code.

a) [25 points] Write your own `myknn()` and `mynw()`

```r
# My KNN function
myknn <- function(x0, x, y, k) {
  # Function to calculate Euclidean distance
  euclidean_distance <- function(a, b) {
    return(sqrt(sum((a - b) ^ 2)))
  }
  # calculate euclidean distance of target point from all x
  dist = unlist(lapply(x, euclidean_distance, x0))
  # sorting distances to find the smallest k for index of  neighbors
  neighbour_ind = which(dist %in% sort(dist)[1:k])
  # mean of the neighboring y values
  ret = mean(y[neighbour_ind])
  return(ret)
}
```

```r
# My NW kernel regression function
mynw <- function(x0, x, y, h) {
  # Calculating weights based on Gaussian distribution
  w = dnorm((x0 - x) / h) / h
  # Calculating fhat using weghted y values
  fhat = sum(w * y) / sum(w)
  return(fhat)
}
```

b) [15 points] Test your code using the artificial data in the lecture note. Try to predict a target point at $x_0 = 2.5$. Compare your results with existing R functions `kknn()` and `locpoly()`, using `k = 10` and `h = 0.5`, respectively, make sure that your implementation is correct. For the `locpoly()` function, you may not be able to directly obtain the prediction at $x_0 = 2.5$. Hence, demonstrate your result in a figure to show that they are correct.

```
# Given parameters to be used
x0 = 2.5
k = 10
h = 0.5

# Generating artificial data
set.seed(1)
x <- runif(40, 0, 2 * pi)
y <- 2 * sin(x) + rnorm(length(x))
```

```
# Finding target using `kknn()
library(kknn)
knn.fit = kknn(
  y ~ x,
  train = data.frame("x" = x, "y" = y),
  test = data.frame("x" = x0),
  k = 10,
  kernel = "rectangular"
)

# Finding target using `myknn()`
fhat_myknn = myknn(x0, x, y, k)

# Comparing results of my_knn() with existing R function `kknn()`
knn.fit$fitted.values-fhat_myknn
```

```
## [1] 0
```

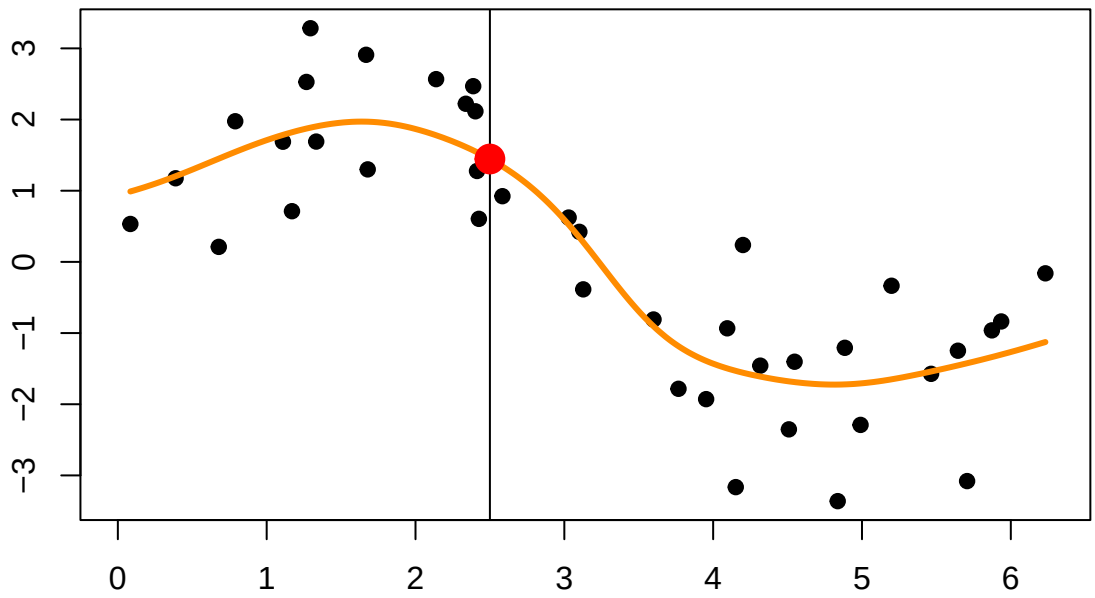
```
# Finding NW estimates using locpoly()
library(KernSmooth)
```

```
## KernSmooth 2.23 loaded
## Copyright M. P. Wand 1997-2009
```

```
NW.fit = locpoly(x,
                 y,
                 degree = 0,
                 bandwidth = 0.5,
                 kernel = "normal")

# Finding target using `mynw()`
fhat_mynw = mynw(x0, x, y, h)
```

```
# Plot to show NW estimates using locpoly()
plot(
```

```
  x,
  y,
  xlim = c(0, 2 * pi),
  xlab = "",
  ylab = "",
  cex.lab = 1.5,
  pch = 19
)

# NW estimated function using locpoly()
lines(NW.fit$x, NW.fit$y, col = "darkorange", lwd = 3)

# estimated using our own code
abline(v=2.5)
points(x0,
       fhat_mynw,
       col = "red",
       pch = 19,
       cex = 2)
```



The estimated value of y for the point $x0$=2.5 using myknn() when compared with the result of R function kknn()- taking difference between the two values-comes out to be 0 demonstrating that both functions generate similar result.

The estimated value of y for the point $x0$=2.5 using mynw() lies on the line of NW estimated function using locpoly. This demonstrates the fact that mynw() implementation generates similar result as the locpoly() and hence it is correct.

Question 2: The Bias-variance Trade-off

We are going to perform a slightly more complicated simulation analysis of the bias-variance trade-off using prediction errors. Hence, you will need to utilize the functions you wrote in the previous question. We can then vary the tuning parameter k or h to see how they changes. Following the idea of simulation studies in previous HW assignments, setup a simulation study. Complete this question by performing the following steps. Note that you would have a triple-loop to complete this question, the first loop for k and h, the second loop for repeating the simulation `nsim` times, and the third loop for going through all testing points.

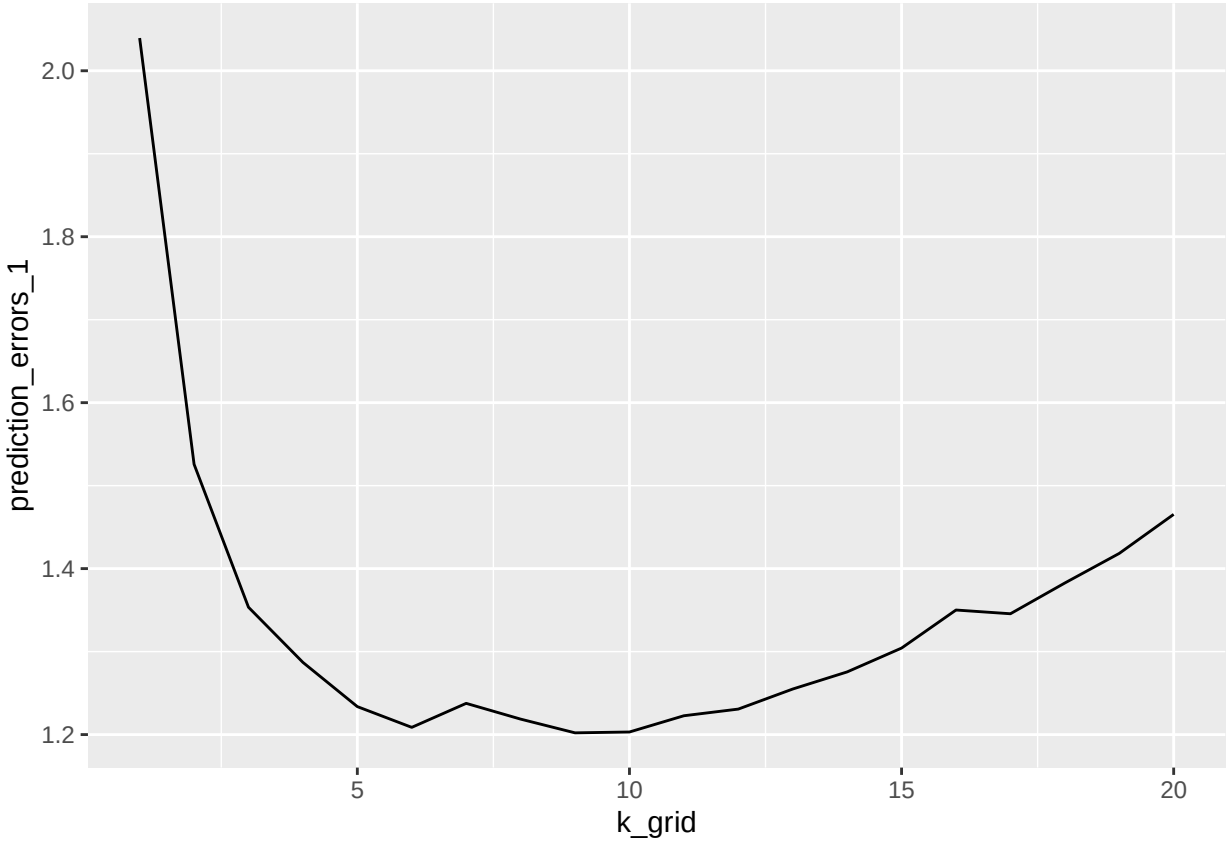
- Generate data using the same model in the previous question
- Generate 100 random testing points uniformly within the range of $[0, 2\pi]$ and also generate their outcomes.
- For each testing point, calculate both the kNN and kernel predictions using your own functions, with a given k and h
- Summarize your model fitting results by calculating the mean squared prediction error
- (the second loop) Run this simulation `nsim` $= 200$ times to obtain the averaged prediction errors.
- (the first loop) Vary your k from 1 to 20, and vary your h in `seq(0.1, 1, length.out = 20)`. Since both have length 20, you could write them in the same loop.

```
# Generating 100 testing points
testx = runif(100, 0, 2 * pi)

# Generating outcomes of testing points
testy = 2 * sin(testx) + rnorm(length(testx))

nsim=200
h_grid = seq(0.1, 1, length.out = 20)
k_grid=seq(1, 20, 1)

# MSE_matrix to store mean squared error of each testing point at different values of k
MSE_matrix_knn = matrix(NA, nsim, 20)
# MSE_matrix to store mean squared error of each testing point at different values of h
MSE_matrix_nw = matrix(NA, nsim, 20)



for (k in 1:20) {
  for (nsim in 1:200) {

    # Generating artificial data
    x <- runif(40, 0, 2 * pi)
    y <- 2 * sin(x) + rnorm(length(x))

    # Initialing se1=0 for every simulation
    se1 = rep(NA,100)
    se2= rep(NA,100)

    for (t in 1:100) {
      # Estimate using myknn() and its squared error
      y_hat1 = myknn(testx[t], x, y, k)
      se1[t] = (testy[t] - y_hat1) ^ 2
      # Estimate using mynw() and its squared error
      y_hat2 = mynw(testx[t], x, y, h_grid[k])
      se2[t] = (testy[t] - y_hat2) ^ 2
```

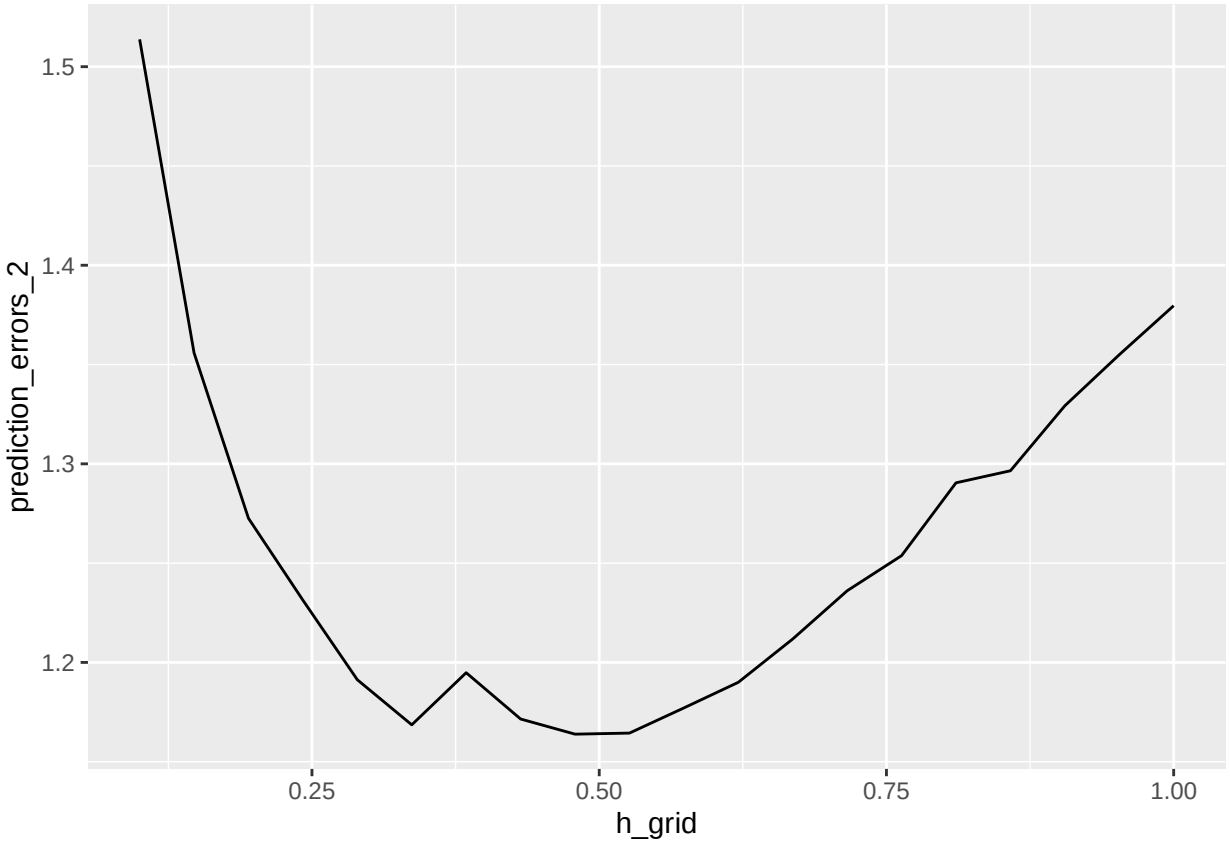
```
    }

    # Mean squared error value stored for each simulation at a particular k value
    MSE_matrix_knn[nsim, k] = mean(se1)
    # Mean squared error value stored for each simulation at a particular h value
    MSE_matrix_nw[nsim, k] = mean(se2)
  }
}
```

After obtaining the simulation results, provide a figure to demonstrate the bias-variance trade-off. What are the optimal k and h values based on your simulation? What kind of values of k or h would have large bias and small variance? And what kind of values would give small bias and large variance?

```
# Plot of prediction errors with changing values of k
prediction_errors_1=colMeans(MSE_matrix_knn)
df1 = data.frame(k_grid, prediction_errors_1)
library(ggplot2)
ggplot(df1, aes(x=k_grid)) +
  geom_line(aes(y = prediction_errors_1))
```




```
# Plot of prediction errors with changing values of h
prediction_errors_2=colMeans(MSE_matrix_nw)
df2 = data.frame(h_grid, prediction_errors_2)
ggplot(df2, aes(x=h_grid)) +
  geom_line(aes(y = prediction_errors_2))
```



Plot-1 shows MSE variations with different values of k.

Plot-2 shows MSE variations with different values of h.

```
# Best value of k
k_optimal=k_grid[which.min(prediction_errors_1)]
# Best value of h
h_optimal=h_grid[which.min(prediction_errors_2)]

k_optimal
```

```
## [1] 9
```

```
h_optimal
```

```
## [1] 0.4789474
```

The optimal k and h values based on the simulation are shown above in order. At these values of k (and h) there is minimum mean squared error.

As h increase, we are using a "larger" neighborhood of the target point x0 for the weighted average. Hence, the estimation is more stable, i.e., smaller variance. But this would introduce a larger bias because the neighborhood can already be far away. We would get larger variance and smaller bias as we decrease the value of h.

As k increases, we have a more stable model, i.e., smaller variance. However, the bias is also increased. As k decreases, the bias decreases, but the variance increases and hence, model is less stable.

```
y_hat1_k_2=rep(NA,length(testy))
y_hat1_k_20=rep(NA,length(testy))
y_hat1_k_optimal=rep(NA,length(testy))

for(i in 1:length(testy)){
  y_hat1_k_2[i]=myknn(testx[i],x,y,2)
  y_hat1_k_20[i]=myknn(testx[i],x,y,20)
  y_hat1_k_optimal[i]=myknn(testx[i],x,y,k_optimal)
}

plot(x,y,xlim=c(0,2*pi),pch=19)
lines(testx,y_hat1_k_2, type="s",col="darkorange",lwd=3)
lines(testx,y_hat1_k_20, type="s",col="red",lwd=3)
lines(testx,y_hat1_k_optimal, type="s",col="blue",lwd=3)
legend("topright",c("k=2","k=20","k=optimal"),col=c("darkorange","red","blue"),lty=1,lwd=2,cex=1.5)
title(main=paste("knn results: Bias-Variance trade-off"),cex.main=1.5)
```