STAT 542 / CS 598: Homework 3

Fall 2019, by John Moran (jfmoran2)

Due: Monday, Oct 7 by 11:59 PM Pacific Time

Contents

Question 1 [50 Points] A Simulation Study

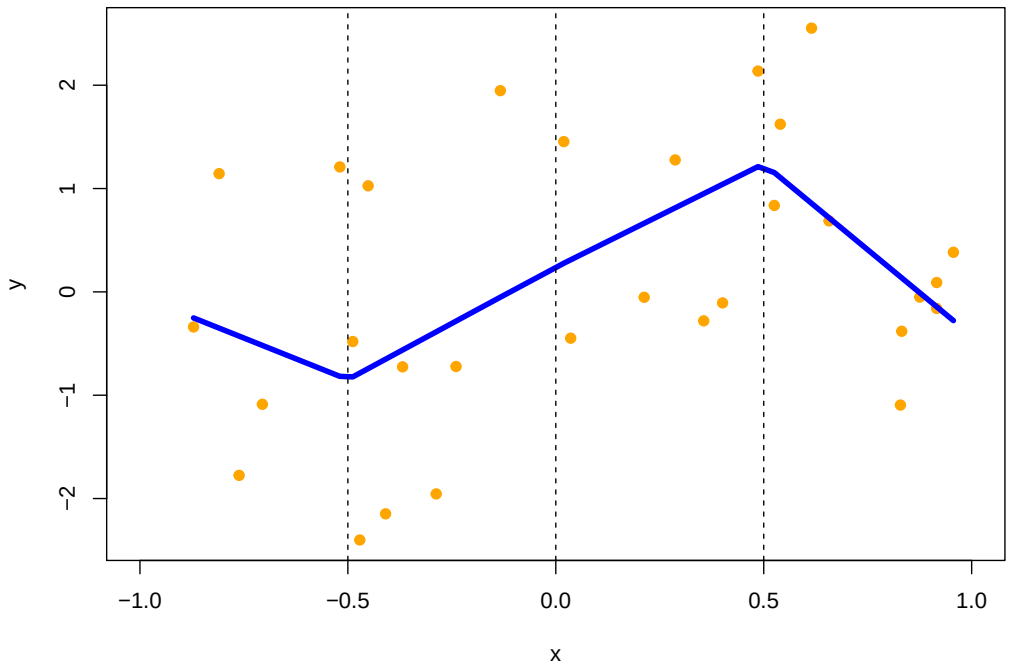
We will perform a simulation study to compare the performance of several different spline methods. Consider the following settings:

- Training data $n = 30$: Generate x from $[-1, 1]$ uniformly, and then generate $y = \sin(\pi x) + \epsilon$, where ϵ's are iid standard normal

```
set.seed(6)
n.train = 30

#sort the x-data since generating randomly
x = sort(runif(n.train, -1, 1))
eps <- rnorm(n.train)
y <- sin(pi * x) + eps
```

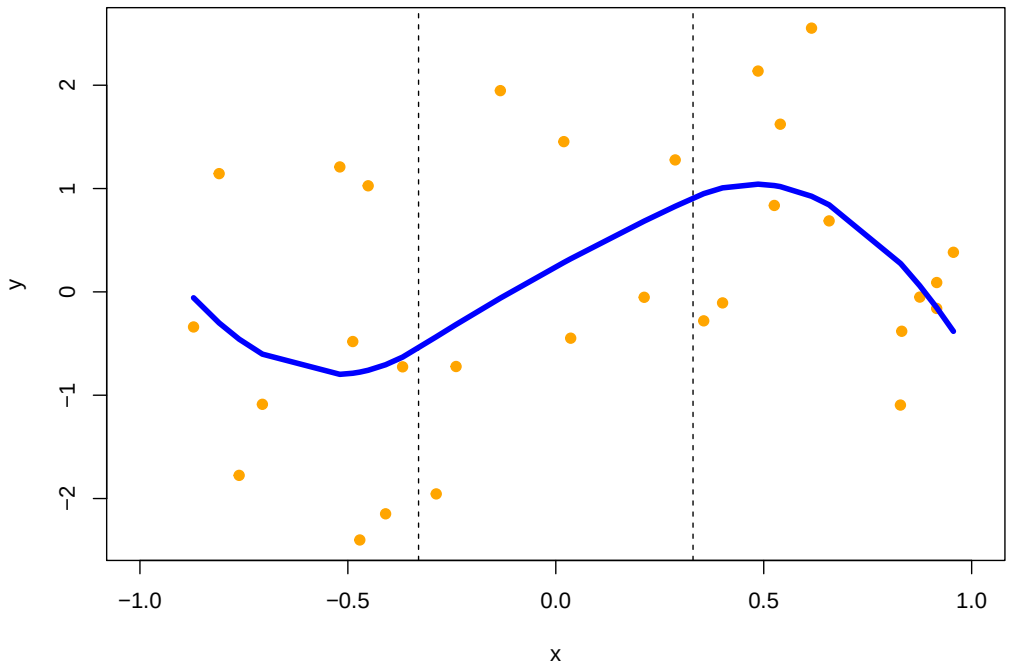
- Consider several different spline methods:
  - Write your own code (you cannot use `bs()` or similar functions) to implement a continuous piecewise linear spline fitting. Choose knots at $(-0.5, 0, 0.5)$

```
myknots.piece = c(-0.5, 0, 0.5)

pos <- function(x) x*(x>0)
mybasis = cbind("int" = 1, "x_1" = x,
                "x_2" = pos(x - myknots.piece[1]),
                "x_3" = pos(x - myknots.piece[2]),
                "x_4" = pos(x - myknots.piece[3]))

lmfit.piece <- lm(y ~ . -1 , data = data.frame(mybasis))

plot(x, y, pch = 19, col = "orange", xlim=c(-1, 1))
lines(x, lmfit.piece$fitted.values, lty = 1, col = "blue", lwd = 4)
abline(v = myknots.piece, lty = 2)
title("Continuous piecewise linear spline")
```



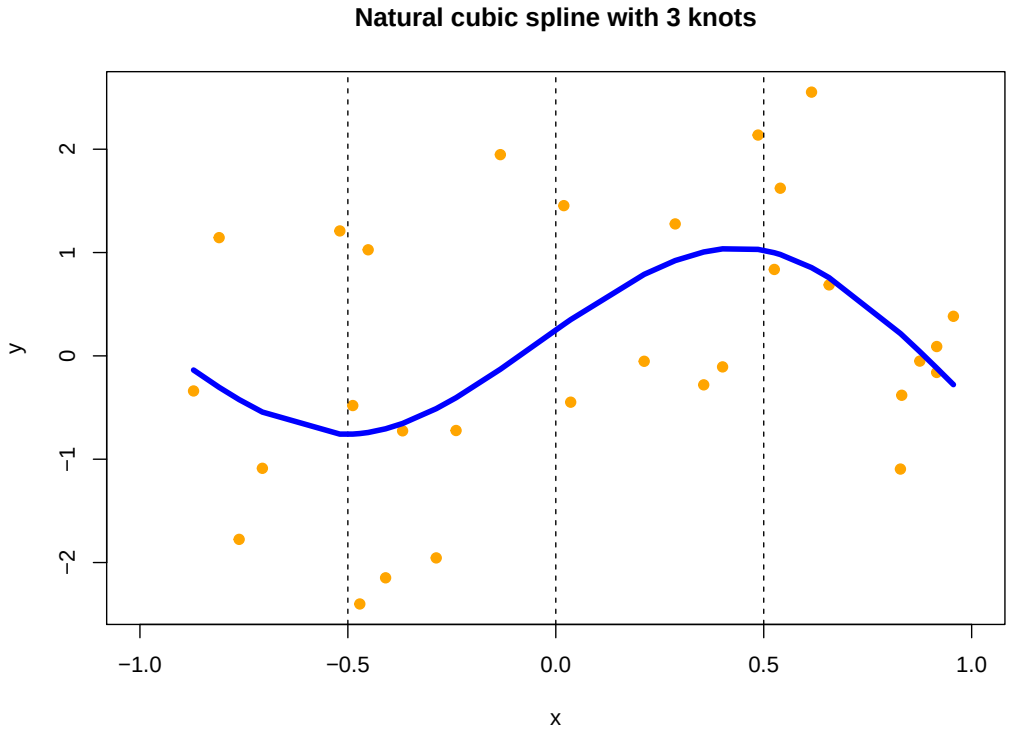
- Use existing functions to implement a quadratic spline 2 knots. Choose your own knots.

```
myknots.quad = c(-0.33, 0.33)
# quadratic spline with 2 knots
lmfit.quad <- lm(y ~ splines::bs(x, degree = 2, knots=myknots.quad,
                                 Boundary.knots = c(-1,1)))

plot(x,y, pch = 19, col = "orange", xlim=c(-1,1))
abline(v = myknots.quad, lty = 2)
lines(x, lmfit.quad$fitted.values, lty=1, col = "blue", lwd = 4)
title("Quadratic spline with 2 knots")
```



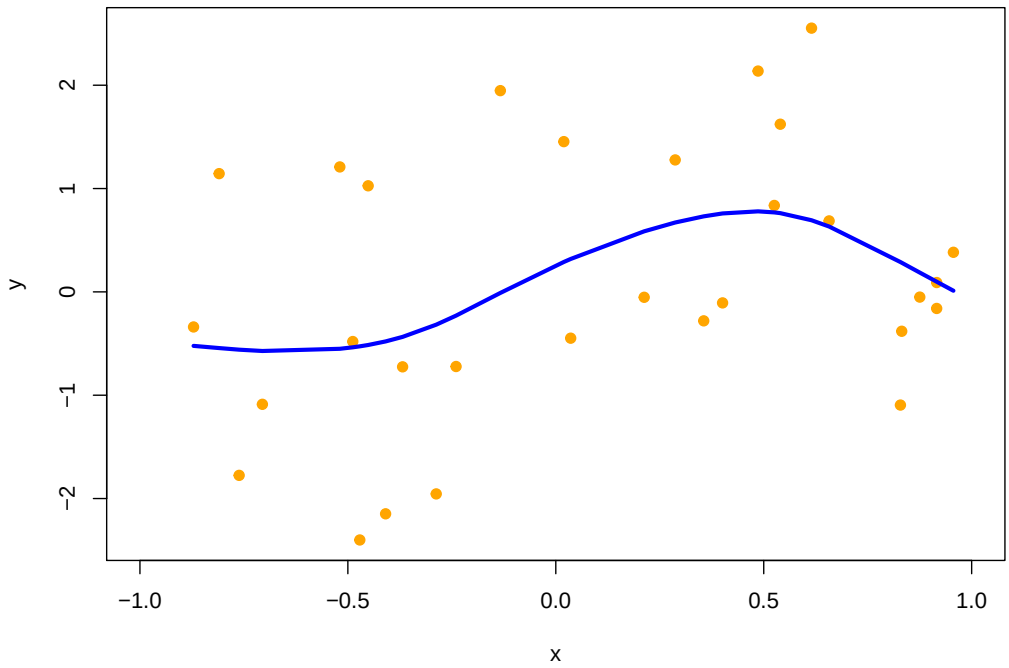
- Use existing functions to implement a natural cubic spline with 3 knots. Choose your own knots.

```
myknots.ncubic = c(-0.5, 0, 0.5)
lmfit.ncubic <- lm(y ~ splines::ns(x, knots = myknots.ncubic, Boundary.knots = c(-1,1)))

plot(x,y, pch = 19, col = "orange", xlim=c(-1,1))
abline(v = myknots.ncubic, lty = 2)
lines(x, lmfit.ncubic$fitted.values, lty=1, col = "blue", lwd = 4)
title("Natural cubic spline with 3 knots")
```



- Use existing functions to implement a smoothing spline. Use the built-in ordinary leave-one-out cross-validation to select the best tuning parameter.

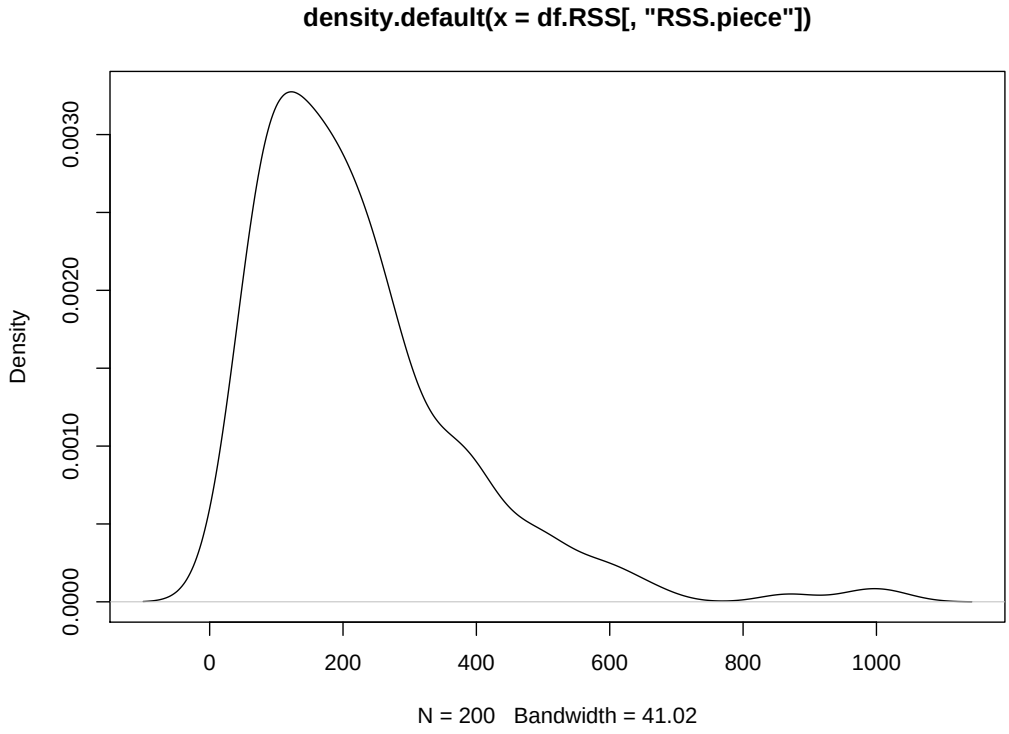
```
#smoothing spline, cv = TRUE means use ordinary leave one out cross validation
fit.smooth = smooth.spline(x, y, cv=TRUE)

plot(x, y, pch = 19, col = "orange", xlim=c(-1,1))
lines(x, fit.smooth$y, col="blue", lty=1, lwd = 3)
title("Smoothing spline")
```



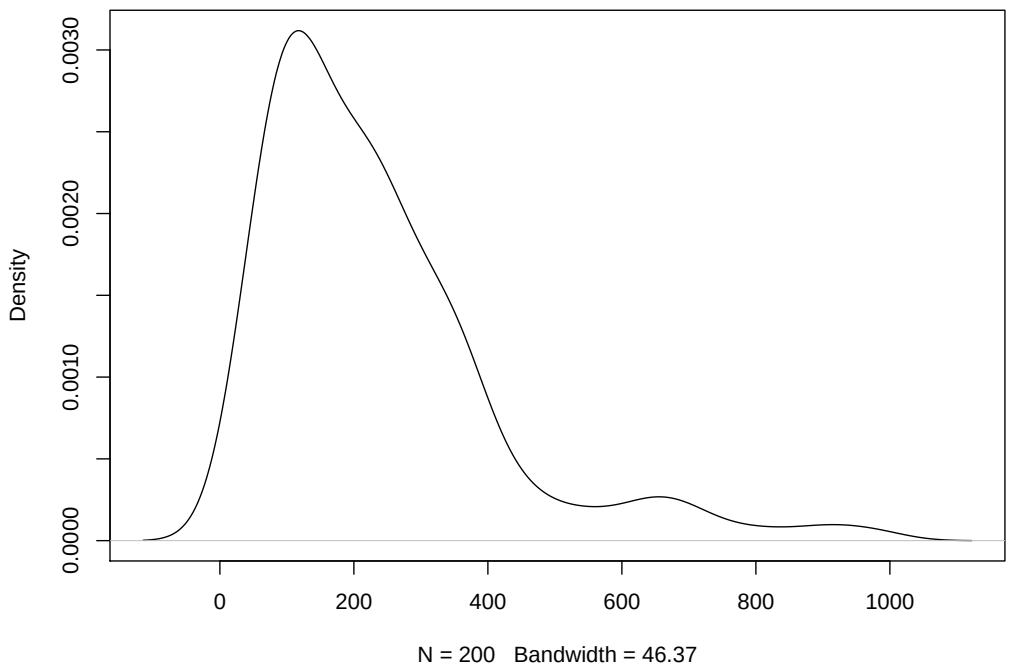
- After fitting these models, evaluate their performances by comparing the fitted functions with the true function value on an equispaced grid of 1000 points on $[-1, 1]$. Use the squared distance as the metric.
- Repeat the entire process 200 times. Record and report the mean, median, and standard deviation of the errors for each method. Also, provide an informative boxplot that displays the error distribution for all models side-by-side.

```
loop.max = 200
n.true = 1000

RSS <- function(x1, x2) {
  rss <- sum((x1-x2)^2)
  return (rss)
}

#set up a data frame to save the error for each method for each iteration
df.RSS <- data.frame(matrix(ncol = 4, nrow = loop.max))
colnames(df.RSS) <- c("RSS.piece","RSS.quad", "RSS.ncubic", "RSS.smooth")

#calculate x.true and y.true outside of the loop since they will never
#change
x.true <- seq(-1, 1, length.out = n.true)
x.true.df <- as.data.frame(x.true)
colnames(x.true.df) <- "x"

y.true <- sin(pi * x.true)
y.true.df <- as.data.frame(seq(-1, 1, length.out = n.true))
colnames(y.true.df) <- "y"

#calculate true basis based on x.true positions outside of the loop
mybasis.true = cbind("int" = 1, "x_1" = x.true,
                "x_2" = pos(x.true - myknots.piece[1]),
```

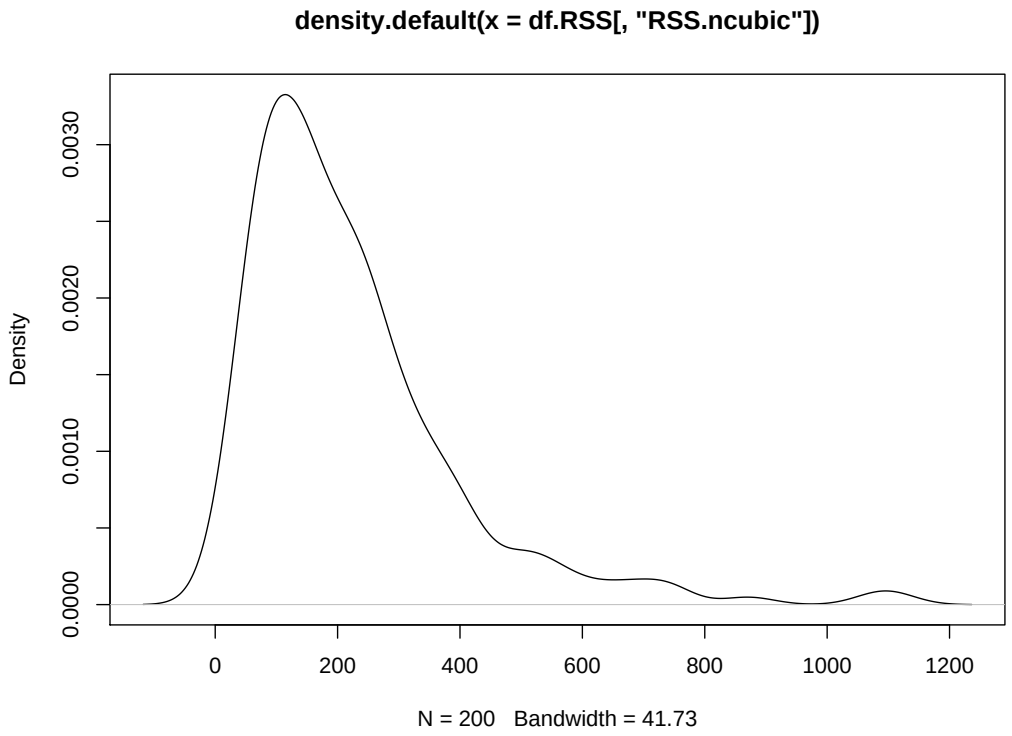
```
                "x_3" = pos(x.true - myknots.piece[2]),
                "x_4" = pos(x.true - myknots.piece[3]))

for (j in 1:loop.max) {

  x = sort(runif(n.train, -1, 1))

  eps <- rnorm(n.train)
  y <- sin(pi * x) + eps

  # piecewise linear
  mybasis = cbind("int" = 1, "x_1" = x,
                        "x_2" = pos(x - myknots.piece[1]),
                        "x_3" = pos(x - myknots.piece[2]),
                        "x_4" = pos(x - myknots.piece[3]))
  lmfit.piece <- lm(y ~ . -1 , data = data.frame(mybasis))
  pred.piece <- predict(lmfit.piece, data.frame(mybasis.true))
  RSS.piece <- RSS(pred.piece, y.true)
  df.RSS[j, "RSS.piece"] <- RSS.piece

  # quadratic spline with 2 knots
  lmfit.quad <- lm(y ~ splines::bs(x, degree = 2, knots=myknots.quad,
                                   Boundary.knots = c(-1,1)))
  pred.quad <- predict(lmfit.quad, newdata = x.true.df)
  RSS.quad <- RSS(pred.quad, y.true)
  df.RSS[j, "RSS.quad"] <- RSS.quad

  # natural cubic spline with 3 knots
  lmfit.ncubic <- lm(y ~ splines::ns(x, knots = myknots.ncubic, Boundary.knots = c(-1,1)))
  pred.ncubic <- predict(lmfit.ncubic, newdata = x.true.df)
  RSS.ncubic <- RSS(pred.ncubic, y.true)
  df.RSS[j, "RSS.ncubic"] <- RSS.ncubic

  #smoothing spline, cv = TRUE means use ordinary leave one out cross validation
  fit.smooth = smooth.spline(x, y, cv=TRUE)

  # don't use newdata in call to predict here
  pred.smooth <- predict(fit.smooth, x.true.df)$y
  RSS.smooth <- RSS(pred.smooth, y.true)
  df.RSS[j, "RSS.smooth"] <- RSS.smooth

}

d.piece <- density(df.RSS[,"RSS.piece"])
plot(d.piece)
```



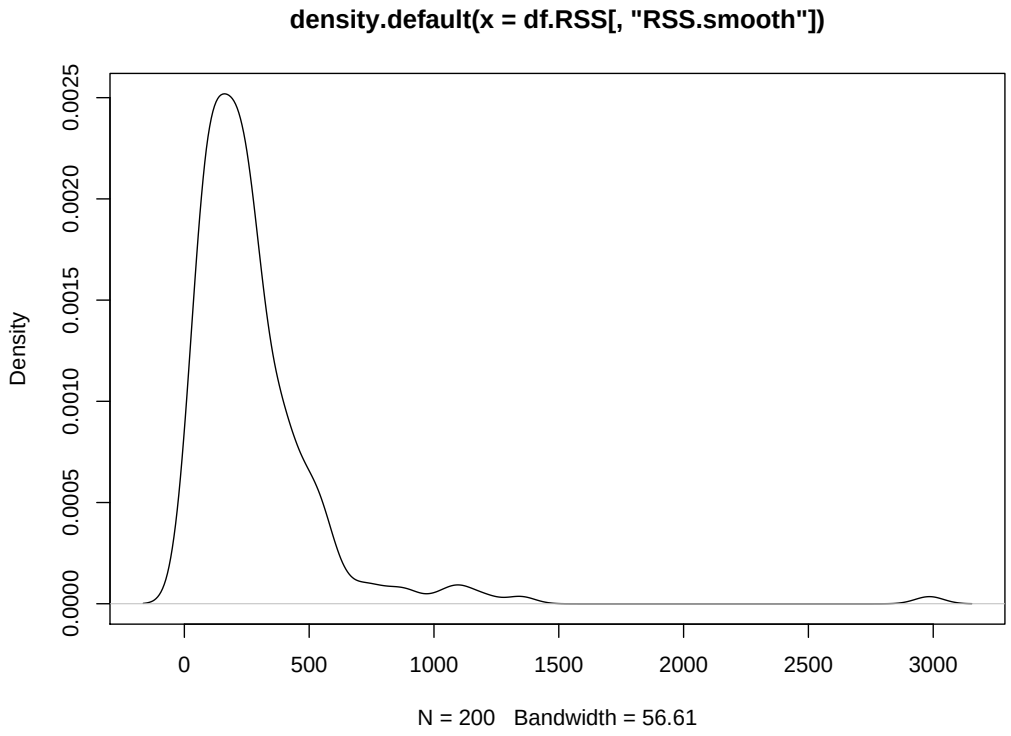

```
mu.piece <- mean(df.RSS[,"RSS.piece"])
med.piece <- median(df.RSS[,"RSS.piece"])
sd.piece <- sd(df.RSS[,"RSS.piece"])

d.quad <- density(df.RSS[,"RSS.quad"])
plot(d.quad)
```



```r
mu.quad <- mean(df.RSS[,"RSS.quad"])
med.quad <- median(df.RSS[,"RSS.quad"])
sd.quad <- sd(df.RSS[,"RSS.quad"])

d.ncubic <- density(df.RSS[,"RSS.ncubic"])
plot(d.ncubic)
```




```r
mu.ncubic <- mean(df.RSS[,"RSS.ncubic"])
med.ncubic <- median(df.RSS[,"RSS.ncubic"])
sd.ncubic <- sd(df.RSS[,"RSS.ncubic"])

d.smooth <- density(df.RSS[,"RSS.smooth"])
plot(d.smooth)
```

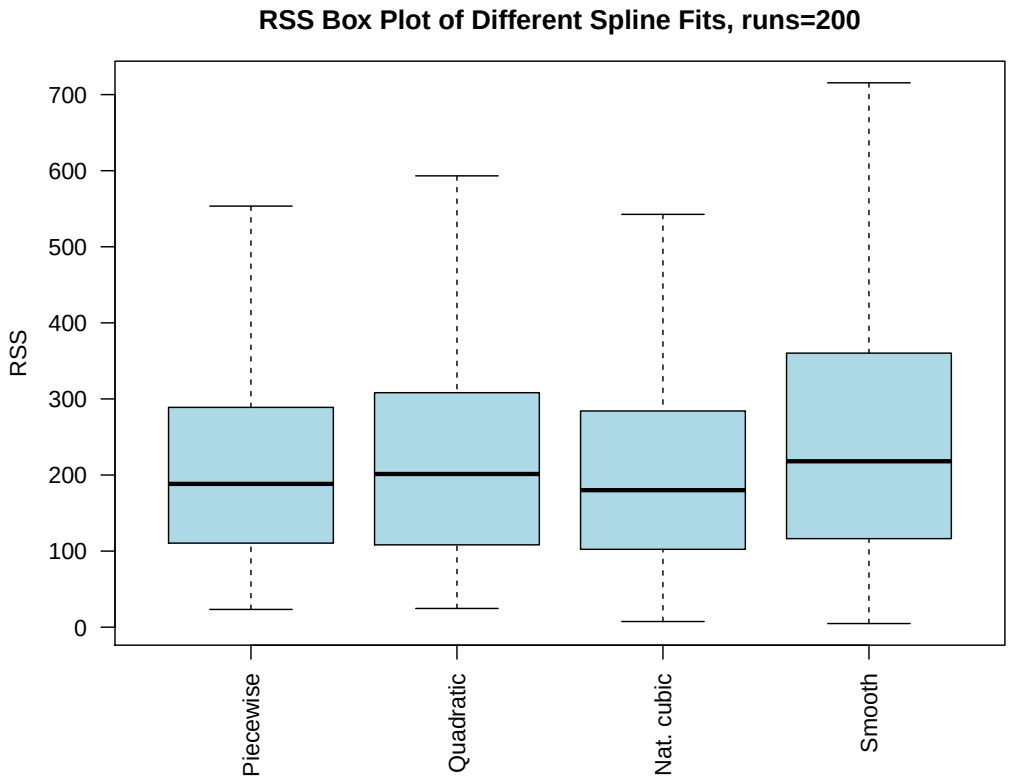



```
mu.smooth <- mean(df.RSS[,"RSS.smooth"])
med.smooth <- median(df.RSS[,"RSS.smooth"])
sd.smooth <- sd(df.RSS[,"RSS.smooth"])

library(knitr)
df.stats <- data.frame(matrix(ncol = 3, nrow = 4))

colnames(df.stats) <- c("mean","median", "sd")
rownames(df.stats) <- c("RSS.piece",
                  "RSS.quad",
                  "RSS.ncubic",
                  "RSS.smooth")

df.stats["RSS.piece","mean"] <- mu.piece
df.stats["RSS.piece","median"] <- med.piece
df.stats["RSS.piece","sd"] <- sd.piece

df.stats["RSS.quad","mean"] <- mu.quad
df.stats["RSS.quad","median"] <- med.quad
df.stats["RSS.quad","sd"] <- sd.quad

df.stats["RSS.ncubic","mean"] <- mu.ncubic
df.stats["RSS.ncubic","median"] <- med.ncubic
df.stats["RSS.ncubic","sd"] <- sd.ncubic

df.stats["RSS.smooth","mean"] <- mu.smooth
df.stats["RSS.smooth","median"] <- med.smooth
df.stats["RSS.smooth","sd"] <- sd.smooth

kable(df.stats, digits=9, caption="RSS for Diff. Spline Fits")
```

Table 1: RSS for Diff. Spline Fits

	mean	median	sd
RSS.piece	226.1297	188.3432	162.8461
RSS.quad	235.3849	201.3569	177.1553
RSS.ncubic	224.3411	180.0998	177.2491
RSS.smooth	281.9004	218.0702	292.2880

```
# mar is bottom, left, top, right
#The default is c(5.1, 4.1, 4.1, 2.1)
par(mar=c(6,5,3,3))
boxplot(df.RSS[,"RSS.piece"], df.RSS[,"RSS.quad"],df.RSS[,"RSS.ncubic"],  df.RSS[,"RSS.smooth"],
        names = c("Piecewise", "Quadratic", "Nat. cubic", "Smooth"),
        main = "RSS Box Plot of Different Spline Fits, runs=200",
        ylab = "RSS",
        las = 2, # perpendicular y labels
        col = "lightblue",
        border = "black",
        outline = FALSE) #don't show the outliers
```



- Comment on your findings. Which method would you prefer?

The natural cubic spline almost always performed best in terms of μ for RSS, with piecewise linear spline coming in a close second. This was the case for testing with different seeds and also works similarly when the sample size is increased to more data points. However, I could force any of the methods to perform best by intentionally selecting a seed value specifically searching for a seed where a specific μ was minimized. I wrote separate code to specifically to test this varying the seed value incrementally from 1 to 10,000.

I would prefer the natural cubic spline method for this particular artificial data set since it performs better,

but it might not fare as well with other data sets and then the auto selection of parameters for the smoothing spline might be a better **first** choice when I don't have prior knowledge of the data.

Question 2 [50 Points] Multi-dimensional Kernel and Bandwidth Selection

Let's consider a regression problem with multiple dimensions. For this problem, we will use the Combined Cycle Power Plant (CCPP) Data Set available at the UCI machine learning repository. The goal is to predict the net hourly electrical energy output (EP) of the power plant. Four variables are available: Ambient Temperature (AT), Ambient Pressure (AP), Relative Humidity (RH), and Exhaust Vacuum (EV). For more details, please go to the dataset webpage. We will use a kernel method to model the outcome. A multivariate Gaussian kernel function defines the distance between two points:

$$K_{\boldsymbol{\lambda}}(x_i, x_j) = e^{-\frac{1}{2}\sum_{j=1}^{p}((x_i - x_j)/\lambda_j)^2}$$

The most crucial element in kernel regression is the bandwidth λ_j. A popular choice is the Silverman formula. The bandwidth for the jth variable is given by

$$\lambda_j = \left(\frac{4}{p+2}\right)^{\frac{1}{p+4}} n^{-\frac{1}{p+4}} \widehat{\sigma}_j,$$

where $\widehat{\sigma}_j$ is the estimated standard deviation for variable j, p is the number of variables, and n is the sample size. Based on this kernel function, use the Nadaraya-Watson kernel estimator to fit and predict the data. You should consider the following:

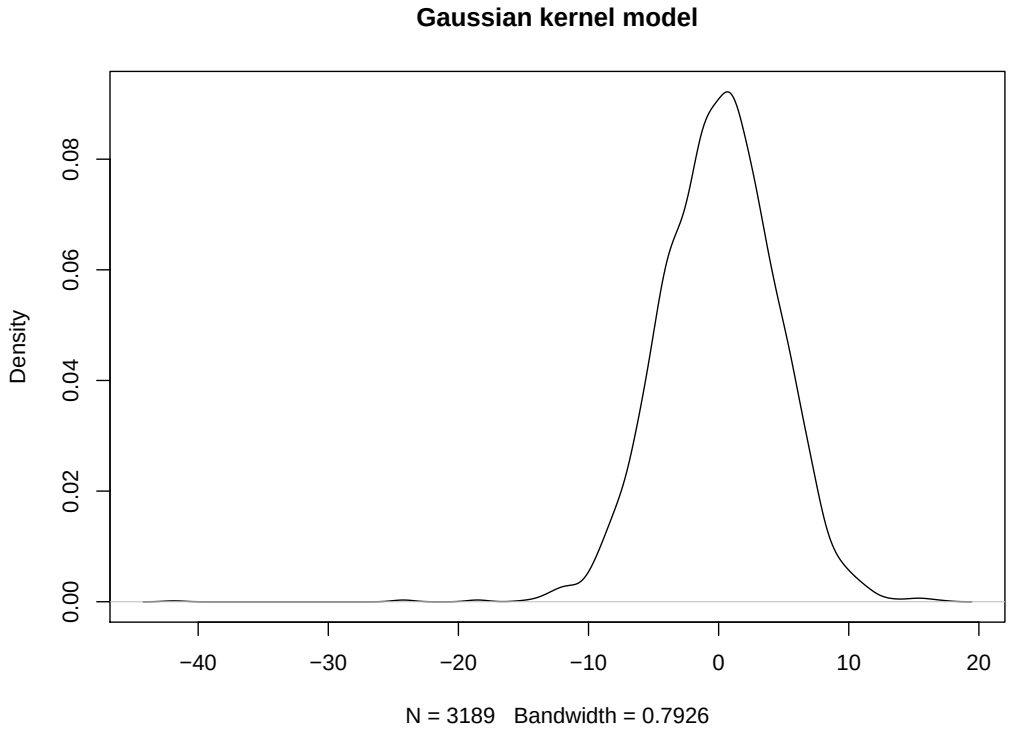
- Randomly select 2/3 of the data as training data, and rest as testing. Make sure you set a random seed. You do not need to repeat this process — just fix it and complete the rest of the questions

```
set.seed(1)
ccpp.data <- read.csv(file="CCPP.csv",header=TRUE, sep=",")

num.rows <- nrow(ccpp.data)
num.cols <- ncol(ccpp.data)

# 2/3 of the data for training, 1/3 for testing
train.ind <- sample(1:num.rows, size=round(num.rows*2/3))

train.data <- ccpp.data[train.ind,]
X <- train.data[, !colnames(train.data) %in% "PE"]
Y <- train.data[, "PE"]
df.train <- cbind(X, Y)

test.data <- ccpp.data[-train.ind,]
test.X <- test.data[, !colnames(test.data) %in% "PE"]
test.Y <- test.data[, "PE"]
df.test <- cbind(test.X, test.Y)
```

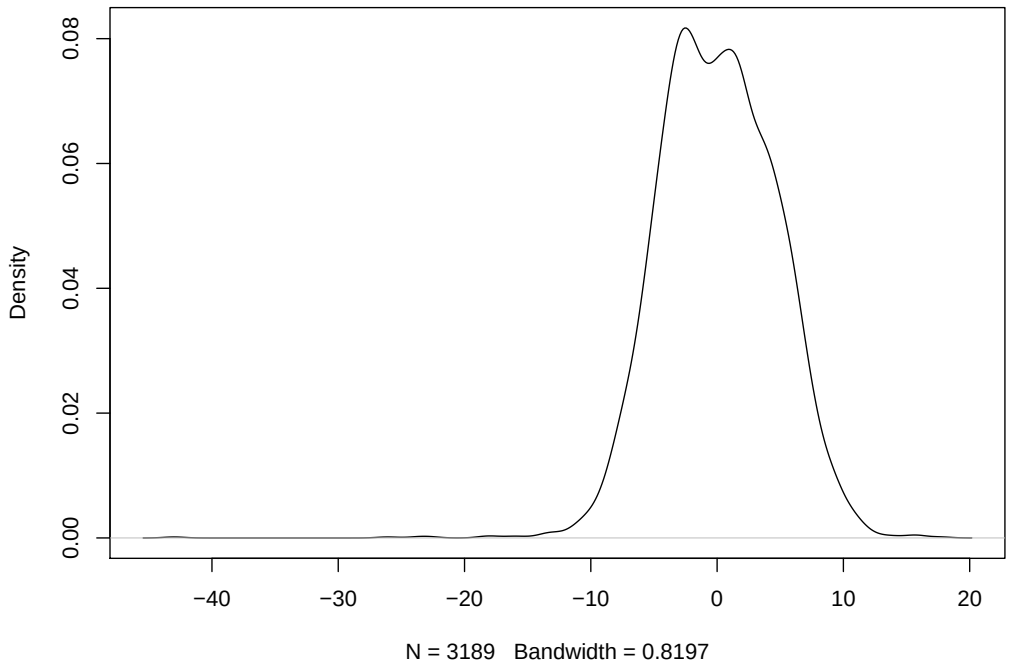
- Fit the model on the training samples using the kernel estimator and predict on the testing sample. Calculate the prediction error and compare this to a linear model

```
MSE <- function(x1, x2) {
    return(RSS(x1,x2)/length(x1))
}

bandwidth <- function(varcol, n, p) {
```

```
  term1 <- (4/(p+2))^(1/(p+4))
  term2 <- n^((-1)/(p+4))
  term3 <- sd(varcol)

  return (term1*term2*term3)
}

# second argument of 2 means apply over columns-- n = nrow of train.X!!!!
bandwidth.vals.orig <- t(data.frame(apply(as.matrix(X), 2, bandwidth,
                          n=nrow(X), p=num.cols)))

gauss.kernel <- function(rowOfTest, entireX, bw) {
  m1 <- sweep(entireX,2,rowOfTest,"-")
  m2 <- (sweep(m1,2,bw,"/"))^2
  return( exp(-0.5*rowSums(m2)))
}

#Nadaraya-Watson kernel estimator
NW.kernel.estimator <- function(X, test.X, bw.vals) {
  my.K.values <- data.frame(matrix(ncol = nrow(X), nrow = nrow(test.X)))

  my.K.values <- (apply(as.matrix(test.X), 1, gauss.kernel, entireX=as.matrix(X), bw=bw.vals))

  # the apply is writing out the data column first, so writes entire row of
  # 6379 from 1 to 6379 in COLUMN 1, not row 1, so have to transpose the matrix.
  my.K.values <- t(my.K.values)
  W <- my.K.values/rowSums(my.K.values)

  Wm <- as.matrix(W)

  P <- drop(Wm %*% as.matrix(Y))
  return(P)
}


P <- NW.kernel.estimator(X, test.X, bandwidth.vals.orig)
plot(density(test.Y-P), main="Gaussian kernel model")
```




```
lmfit <- lm(Y ~ ., data = df.train)
lm.predicted.Y <- predict(lmfit, test.X)
plot(density(test.Y-lm.predicted.Y), main="Linear fit model")
```




```
print(sprintf("Gaussian kernel model. RSS = %.2f, MSE = %.2f", RSS(test.Y, P),
              MSE(test.Y, P)))
```

```
## [1] "Gaussian kernel model. RSS = 62314.43, MSE = 19.54"
```

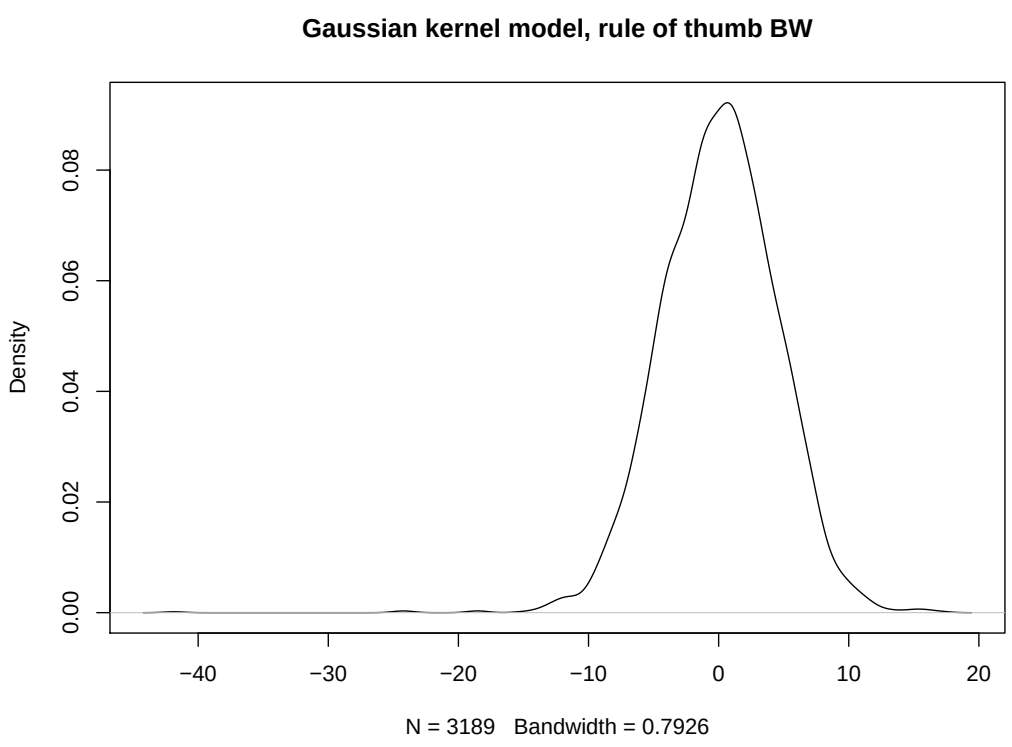
```
print(sprintf("Linear model. RSS = %.2f, MSE = %.2f", RSS(test.Y, lm.predicted.Y),
              MSE(test.Y, lm.predicted.Y)))
```

```
## [1] "Linear model. RSS = 66654.75, MSE = 20.90"
```

The Gaussian kernel model was significantly better than the linear fit model using the specified Silverman bandwidth formula.

- The bandwidth selection may not be optimal in practice. Experiment a few choices and see if you can achieve a better result.

```
bandwidth.rule.thumb <- function(varcol, n, p) {

  term1 <- (4/3)^(1/5)
  term2 <- n^(-1/5)
  term3 <- sd(varcol)

  return (term1*term2*term3)
}

# NOTE rule.thumb BW performs better for this one!
bandwidth.vals.rulethumb <- t(data.frame(apply(as.matrix(X), 2, bandwidth.rule.thumb,
                                               n=nrow(X), p=num.cols)))

P.new <- NW.kernel.estimator(X, test.X, bandwidth.vals.rulethumb)

plot(density(test.Y-P), main="Gaussian kernel model, rule of thumb BW")
```




```
print(sprintf("Gaussian kernel model, new bandwidth. RSS = %.2f, MSE = %.2f",
              RSS(test.Y, P.new), MSE(test.Y, P.new)))
```

```
## [1] "Gaussian kernel model, new bandwidth. RSS = 47183.05, MSE = 14.80"
```

I experimented with different bandwidths and modifying the original formula, and made non-significant improvements, so I did some literature research and found a common bandwidth formula called "the rule of thumb" bandwidth, which is also called "Silverman's rule of thumb", so it is related to our original bandwidth formula but does not use the number of parameters in the calculation. This is the formula used:

$$h = \left(\frac{4\widehat{\sigma}^5}{3n} \right)^{\frac{1}{5}}$$

and it significantly improved the results.

- During all calculations, make sure that you write your code efficiently to improve computational performance

The R code was very slow until I implemented vectorization and the apply function, taking full advantage of R's fast matrix operations. This significantly sped up the code.