STAT 542 / CS 598: Homework 3

Fall 2019, by Regan Chan (ttchan2)

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

Contents

Question 1 [50 Points] A Simulation Study

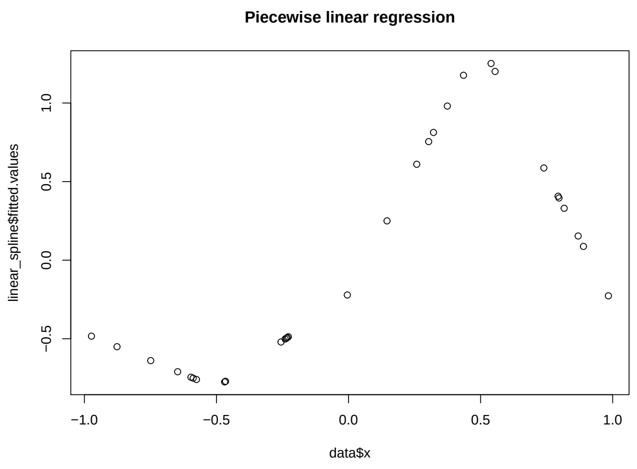
We will perform a simulation study to compare the performance of several different spline methods. Consider the following settings:

- 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

```r
library(purrr)

gen_data <- function(n) {
  x <- runif(n, -1, 1)
  y <- sin(pi*x) + rnorm(n)
  list(x=x, y=y)
}

n <- 30
set.seed(0)
data <- gen_data(n)
```

- 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 $(-0.5, 0, 0.5)$

```r
getFx <- function(X) {
  myknots <- c(-0.5, 0, 0.5)
  h <- function(x){x * (x>0)}
  n <- length(X)
  result <- cbind(X=X, mapply(function(knot){ h(X-knot) }, myknots))
  m <- length(myknots)
  colnames(result)[2:(m+1)] <- sapply(1:m, function(i) paste("K", i, sep=""))
  return(as.data.frame(result))
}

getPiecewiseLinear <- function(data) {
  modelDf <- cbind(getFx(data$x), Y=data$y)
  lm(Y ~ ., data=modelDf)
}
linear_spline <- getPiecewiseLinear(data)
plot(data$x, linear_spline$fitted.values, main="Piecewise linear regression")
```



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

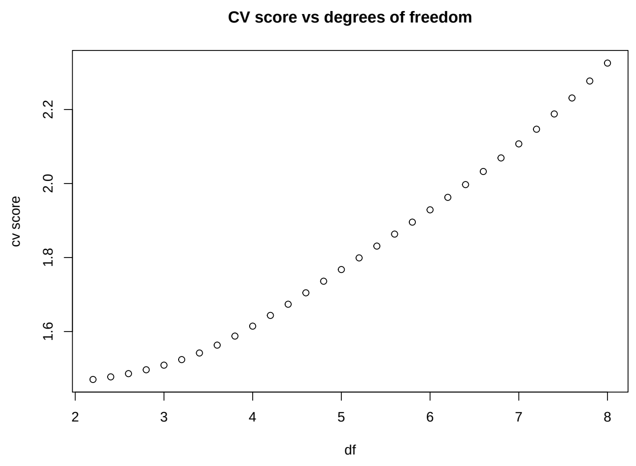
```
library(splines)
getQuadraticSpline <- function(data) lm(y ~ bs(x, Boundary.knots=c(-1, 1)), data=data)
quad_spline <- getQuadraticSpline(data)
```

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

```
getNaturalCubicSpline <- function(data) lm(y ~ ns(x, knots=c(-0.5, 0, 0.5)), data=data)
ncs_spline <- getNaturalCubicSpline(data)
```

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

```
getSmoothingSpline <- function(data, plotCV=FALSE) {
  m <- length(data$x)
  df <- 2+(1:m)/5
  mycv <- sapply(df, function(df_i) smooth.spline(data$x, data$y, df=df_i, cv=T)$cv);
  if (plotCV) {
    plot(df, mycv, xlab="df", ylab="cv score", main="CV score vs degrees of freedom")
  }
  optdf = df[which.min(mycv)]

  smooth.spline(data$x, data$y, df=optdf)
}
ss_spline <- getSmoothingSpline(data, plotCV=TRUE)
```



- 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.

```
newX <- seq(-1, 1, 2/999)
trueY <- sin(pi * newX)
newData <- data.frame(x=newX)

evalModelErrors <- function(linear_spline, quad_spline, ncs_spline, ss_spline) {
  linearErr <- sum((predict(linear_spline, getFx(newX)) - trueY)^2)
  quadErr <- sum((predict(quad_spline, newdata=newData) - trueY)^2)
  ncsErr <- sum((predict(ncs_spline, newdata=newData) - trueY)^2)
  ssErr <- sum((predict(ss_spline, newX)$y - trueY)^2)
  list(Linear=linearErr, Quadratic=quadErr, NaturalCubic=ncsErr, Smooth=ssErr)
}

data.frame(evalModelErrors(linear_spline, quad_spline, ncs_spline, ss_spline))
```

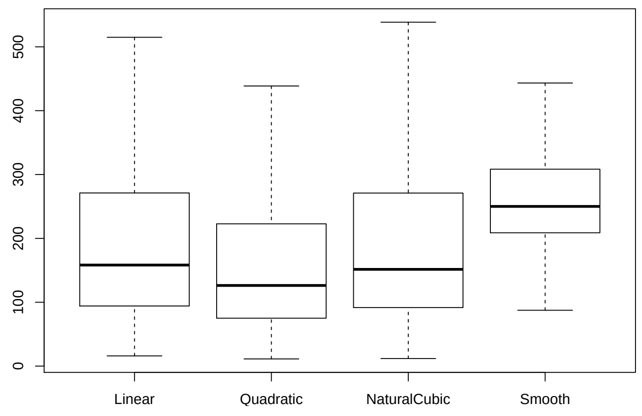
```
##     Linear Quadratic NaturalCubic   Smooth
## 1 39.10067  21.15459     20.37562 192.9344
```

- 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.

```
allErrors <- data.frame(Linear=c(), Quadratic=c(), NaturalCubic=c(), Smooth=c())
for(i in 1:200) {
  data <- gen_data(n)
  linear_spline <- getPiecewiseLinear(data)
  quad_spline <- getQuadraticSpline(data)
  ncs_spline <- getNaturalCubicSpline(data)
  ss_spline <- getSmoothingSpline(data)
  allErrors <- rbind(allErrors, evalModelErrors(linear_spline, quad_spline, ncs_spline, ss_spline))
}
report <- sapply(c(mean, median, sd), function(f){ apply(allErrors, 2, f) })
colnames(report) <- c("mean", "median", "stddev")
report
```

```
##                     mean   median       stddev
## Linear          266.4818 158.2410     804.7628
## Quadratic       189.1251 126.2980     276.8321
## NaturalCubic    213.4603 151.5209     202.1239
## Smooth       388702.7400 250.0650 5493183.0527
```

```
boxplot(allErrors, outline=F)
```



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

Ans: Due to the highly random nature of the generated data (after adding noise), the spline models could not accurately capture the underlying data model. The models tested only tried to overfit the data in hopes that the test error will be low. Out of all four models, the quadratic model performed best because our data model has only a tiny domain X. Where piecewise splines shine are situations where the domain has a large range.

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:

$$K_{\boldsymbol{\lambda}}(x_i, x_j) = e^{-\frac{1}{2}\sum_{k=1}^{p}((x_{ik}-x_{jk})/\lambda_k)^2}$$

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

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

where $\widehat{\sigma}_k$ is the estimated standard deviation for variable k, 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

```r
ccpp <- read.csv("ccpp.csv")
n <- nrow(ccpp)
nTrain <- n * 2/3
nTest <- n - nTrain

set.seed(0)
trainIdx <- sample(1:nTrain)
trainSet <- ccpp[trainIdx, ]
testSet <- ccpp[-trainIdx, ]
```

- 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

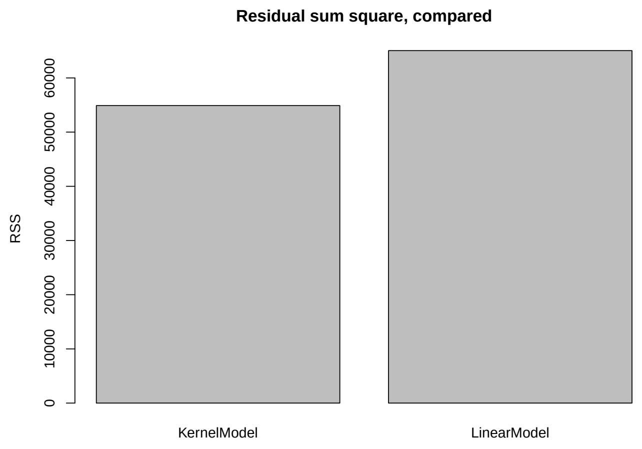
```r
getSilvermanLambda <- function(X) {
  p <- ncol(X)-1
  sd_k <<- apply(X, 2, sd)
  (4/(p+2))^(1/(p+4))*n^(-1/(p+4)) * sd_k
}

getKernel <- function(Xi, Xj, lambda_k) {
  p <- ncol(Xi)-1   # Last col is output
  n <- nrow(Xi)

  exponentSum <- matrix(0, nrow=nrow(Xi), ncol=nrow(Xj))
  for (k in 1:p) {
    xik <- matrix(rep(Xi[,k], nrow(Xj)), ncol=nrow(Xj))
    xjk <- matrix(rep(Xj[,k], nrow(Xi)), nrow=nrow(Xi), byrow=TRUE)
    exponent_k <- (xik - xjk) / lambda_k[k]
    exponentSum <- exponentSum + exponent_k^2
  }
  unscaled <- t(exp(-exponentSum/2))
  scale <- apply(unscaled, 1, sum)
  return(unscaled / scale)
}

mykernel <- getKernel(trainSet, testSet, getSilvermanLambda(trainSet))
predictedPe <- mykernel %*% trainSet$PE
predictRss <- sum((predictedPe - testSet$PE)^2)
lmPe <- predict(lm(PE~., trainSet), newdata=subset(testSet, select=-c(PE)))
lmRss <- sum((lmPe - testSet$PE)^2)
predictRss
```

```
## [1] 54894.7
```

```r
lmRss
```

```
## [1] 65039.63
```

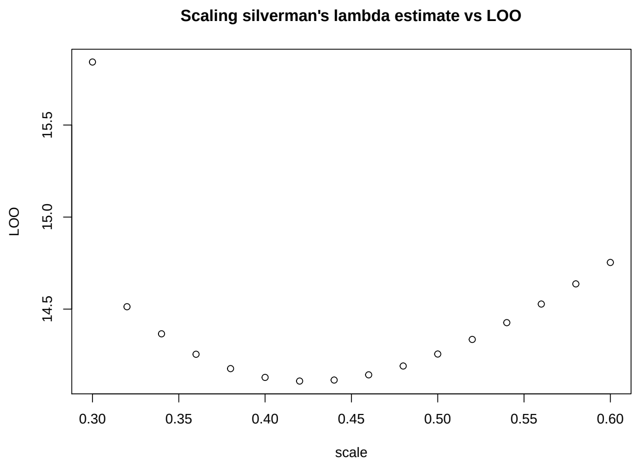
```r
barplot(c(predictRss, lmRss), names.arg=c("KernelModel", "LinearModel"), ylab="RSS", main="Residual sum
```



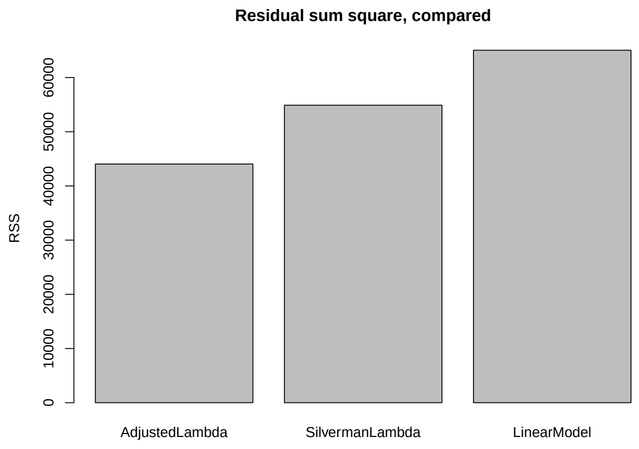
Ans: The kernel estimator is slightly better than a linear model (55,000 RSS vs 65,000 RSS from linear model)

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

```
library(parallel)
originalLambda <- getSilvermanLambda(trainSet)
lambdaMultipliers <- seq(0.3, 0.6, 0.02)

looScore <- mcmapply(function(lambdaMultiplier) {
  mykernel <- getKernel(trainSet, trainSet, originalLambda * lambdaMultiplier)
  y_hat <- mykernel %*% trainSet$PE
  mean(((trainSet$PE - y_hat) / (1-diag(mykernel)))^2)
}, lambdaMultipliers, mc.cores=4)
plot(lambdaMultipliers, looScore, xlab="scale", ylab="LOO", main="Scaling silverman's lambda estimate v
```



```
mykernel <- getKernel(trainSet, testSet, lambdaMultipliers[which.min(looScore)] * originalLambda)
adjustedPe <- mykernel %*% trainSet$PE
adjustedRss <- sum((adjustedPe - testSet$PE)^2)
barplot(c(adjustedRss, predictRss, lmRss), names.arg=c("AdjustedLambda", "SilvermanLambda", "LinearMode
```



Ans: The original silverman's lambda estimate is a bit larger than optimal. By reducing it to 0.42x, the test RSS can be reduced from 55,000 to 44,000.