Nonparametric Regression

Instructor: Dr. Sharandeep Singh Pandher*

1 Introduction

In the previous Chapter, we studied possible extension of generalized linear model but we will emphasize on non-parametric regression in this Chapter. Basically, Regression analysis is divided into two groups as **parametric** and **nonparametric** regression. In **non-parametric** regression analysis, the function is **not pre-defined** and there are **no significant assumptions** as in the parametric regression analysis. The main estimation methods used in non-parametric regression are based on **smoothing**. For this reason, non-parametric regression methods are also called as **smoother**.

in the nonparametric regression model, we assume

$$y_i = f(x_i) + \epsilon_i, \quad i = 1, 2, 3, .., n$$

where $\epsilon_i \quad i = 1, 2, 3, .., n$ are i.i.d. with $E(\epsilon_i) = 0$ and unknown variance σ^2. $f(.)$ is some unknown smooth function.

Note that: The assumption of homogeneous error variance is not actually needed for these methods. however, this assumption is often useful for tuning and inference. The assumption of mean zero error terms is necessary, given that this implies that $E(y_i \mid x_i) = f(x_i)$, which implies that the unknown smooth function $f()$ describes the condition mean of response given predictor. Nonparametric regression estimators (also known as **smoothers**) attempt to estimate the unknown function $f(.)$ from a sample of noisy data.

The following Nonparametric regression estimators(smoothers) will discuss in this chapter:

1. **Kernel** Estimators or Kernel Smoothing
2. **Splines** Estimators including Smoothing Splines and Regression Splines.
3. **Local Polynomials** Estimators
4. **Wavelets Estimators**

Finally, we will compare the above methods.

*Address: Department of Mathematics and Statistics, University of Alberta, Edomoton, AB,T6G 2G1, Canada, e-mail: sharand1@ualberta.ca;

2 Kernel Estimators

In Kernel Smoothing, weights are defined by a kernel function. These kernel functions; Epanechnikov(parabolic), biweight(Quartic), triangular, Gaussian and uniform etc.
The estimate of f, called is $\hat{f}_\lambda(x)$: and is defined by

$$\hat{f}_\lambda(x) = \frac{1}{n\lambda}\sum_{j=1}^{n} K(\frac{x-x_j}{\lambda})y_j = \frac{1}{n}\sum_{j=1}^{n} w_j y_j$$

where $w_j = K(\frac{x-x_j}{\lambda})/\lambda$, K is a kernel and $\int K = 1$, λ is called the **bandwidth**, window width or smoothing parameter.
Some common examples of K are

Uniform ("rectangular window") kernel:

$$K(x) = \{ \begin{matrix} \frac{1}{2} & |x| \leq 1 \\ 0 & otherwise \end{matrix} \}$$

Triangular kernel :

$$K(x) = \{ \begin{matrix} (1-\mid x \mid) & |x| \leq 1 \\ 0 & otherwise \end{matrix} \}$$

the Gaussian kernel:

$$K(x) = \frac{1}{\sqrt{2\pi}} exp(-\frac{x^2}{2})$$

Epanechnikov kernel:

$$K(x) = \{ \begin{matrix} \frac{3}{4}(1-x^2) & |x| \leq 1 \\ 0 & otherwise \end{matrix} \}$$

biweight kernel:

$$K(x) = \{ \begin{matrix} \frac{15}{16}(1-x^2)^2 & |x| \leq 1 \\ 0 & otherwise \end{matrix} \}$$

Triweight kernel:

$$K(x) = \{ \begin{matrix} \frac{35}{32}(1-x^2)^3 & |x| \leq 1 \\ 0 & otherwise \end{matrix} \}$$

Tricube kernel:

$$K(x) = \{ \begin{matrix} \frac{70}{81}(1-\mid x \mid^3)^3 & |x| \leq 1 \\ 0 & otherwise \end{matrix} \}$$

Cosine kernel:

$$K(x) = \{ \begin{matrix} \frac{\pi}{4} cos(\frac{\pi}{2}x) & |x| \leq 1 \\ 0 & otherwise \end{matrix} \}$$

Logistic kernel:

$$K(x) = \frac{1}{e^x + 2 + e^{-x}}$$

Sigmoid function kernel:

$$K(x) = \frac{2}{\pi} \frac{1}{e^x + e^{-x}}$$

- **Note**: The most preferred function in Kernel Smoothing is the **Epanechnikov** function and efficiency of the above kernels relative to **Epanechnikov** describe in the following table.

Kernel	**Efficiency relative to Epanechnikov kernel in percentage**
Uniform	92.9
Triangular	98.6
Gaussian	95.1
Epanechnikov	100
Quartic (biweight)	99.4
Triweight	98.7
Tricube	99.8
Cosine	99.9
Logistic	88.7
Sigmoid function	84.3

- If the x's are spaced very unevenly, then $\hat{f}_\lambda(x)$ can give poor results. This problem can be rectified by the **Nadaraya-Watson** estimator:

$$\hat{f}_\lambda(x) = \frac{\sum_{j=1}^{n} w_j y_j}{\sum_{j=1}^{n} w_j}$$

  This is a true weighted average where the weights for each y will sum to one.

- **Epanechnikov** kernel has the advantage of some smoothness, compactness and rapid computation. Rapid computation feature is important for larger datasets, particularly when resampling techniques like bootstrap are being used. Even so, any sensible choice of kernel will produce acceptable results, so the choice is not crucially important.

- The choice of smoothing parameter λ is critical to the performance of the estimator and far more important than the choice of kernel. If the smoothing parameter is **too small**, the estimator will be **too rough**; but if it is **too large**, important features will be **smoothed out**. For this reason, We demonstrate the **Nadaraya-Watson** estimator next for a variety of choices of bandwidth on the **simulated data** and use the **ksmooth** function which is part of the R base package. The r-code explain below

```
rm(list=ls())
library(faraway)
################## Generate a Simulated Data#################
X = runif(100, min=0, max=4*pi)
Y = sin(X) + rnorm(100, sd=0.3)
####################Choices of bandwidth ############
##########choices of bandwidth#####################
Kreg1 = ksmooth(x=X,y=Y,kernel = "normal",bandwidth = 0.3)
Kreg2 = ksmooth(x=X,y=Y,kernel = "normal",bandwidth = 0.8)
Kreg3 = ksmooth(x=X,y=Y,kernel = "normal",bandwidth = 2.6)
Kreg4 = ksmooth(x=X,y=Y,kernel = "normal",bandwidth = 4.0)
plot(X,Y,pch=20)
lines(Kreg1, lwd=4, col="red")
lines(Kreg2, lwd=4, col="green")
lines(Kreg3, lwd=4, col="blue")
lines(Kreg4, lwd=4, col="purple")
legend("topright", c("h=0.3","h=0.8","h=2.6","h=4.0"),
             lwd=7, col=c("red","green","blue","purple"))
```

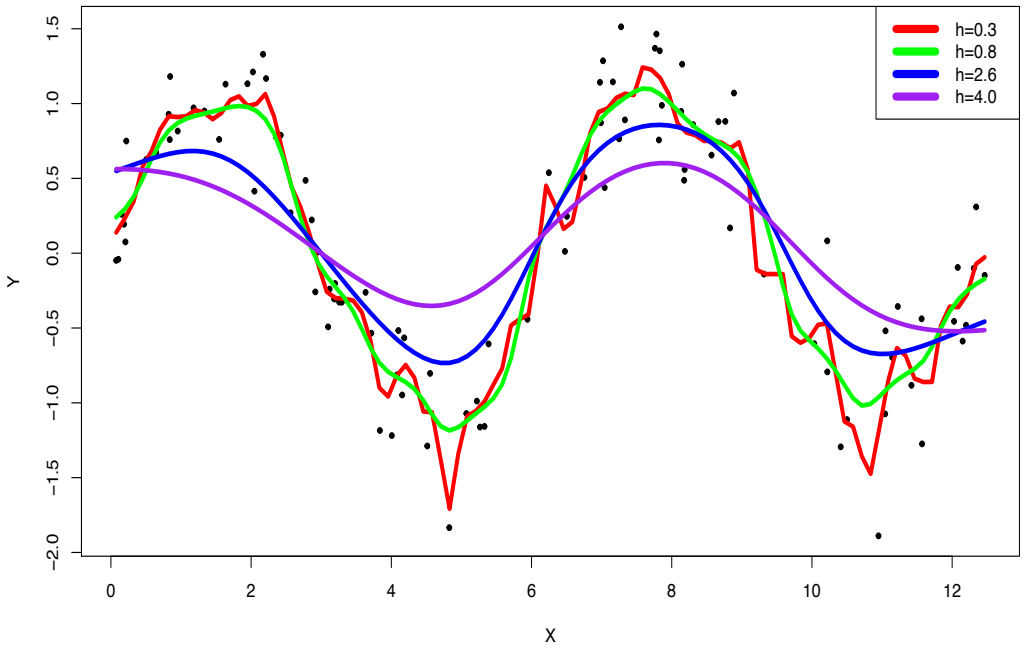


Figure 1: Nadaraya-Watson kernel smoother with a normal kernel for four different bandwidths on the simulated data

Hence, The figure-1 showed that this kernel regression also suffers from the **bias-variance tradeoff**: when h=0.3 is small (the red curve), the **variabiility** is **large** but the **bias**

is **small**; when h=4.0 is large (purple curve), the **variability** is **small** but the **bias** is **large**! **Note:** Below is a code that you can run to see how the **variability** and **bias** are interacting with each other using bandwidth=0.4:

```
 h0 = 0.4
for(i in 1:20){
  X1 = runif(100, min=0, max=4*pi)
  Y1 = sin(X1) + rnorm(50, sd=0.3)
  Kreg = ksmooth(x=X1,y=Y1,kernel = "normal",bandwidth = h0)
  plot(X1,Y1,pch=20, main=paste("h =",h0))
  lines(Kreg, lwd=4, col="purple")
  Sys.sleep(0.5)
}
```

Note: The **best smoothing bandwidth** should balance both the bias and variability So in this concern, we will introduce an elegant approach called cross-validation that allows us to choose the smoothing bandwidth subject to certain optimality criterion . Cross-validation (CV) is a popular general-purpose method that can be written as

$$CV(\lambda) = \frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{f_{\lambda_j}}(x_j))^2$$

We pick the λ that minimizes this criterion. True cross-validation is computationally expensive, so an approximation to it, known as generalized cross-validation(GCV), There are also many other methods of automatically selecting the λ such as leave-one-out cross-validation (LOOCV),leave "p" out cross-validation, Holdout cross-validation Repeated random subsampling validation, **k-fold cross-validation**,Stratified k-fold cross-validation, **Time Series cross-validation** and Nested cross-validation etc. Most of the researchers are using **k-fold cross-validation** that consider the following steps:

1. We random split the data into k roughly equal size subsamples, called them $D_1, , D_k$.
2. We use D_1 as the validation set and others as training set, compute the predition error, called it $Err1$.
3. Then, we use D_2 as the validation set and others as training set, compute the predition error, called it $Err2$.
4. We keep doing this until every subsample has been used as a validation set.
5. We use the average of these k prediction errors $Err = \frac{1}{k}\sum_{l=1}^{k} Err_l$ as an estimate of the prediction error.
6. . After obtaining one such an estimate, we will repeat step 1-5 several times (re-split the data into k subsamples, computing the prediction error) and use the average prediction errors as the final error estimate.

Below is an example of 10-fold:

```
################ k-fold cross validation#######
n = length(X)
```

```
N_cv = 50
k = 10
cv_lab = sample(n,n,replace=F) %% k
## randomly split all the indices into k numbers
h_seq = seq(from=0.1,to=2.0, by=0.2)
CV_err_h = rep(0,length(h_seq))
for(i_tmp in 1:N_cv){
  CV_err_h_tmp = rep(0, length(h_seq))
  cv_lab = sample(n,n,replace=F) %% k
  for(i in 1:length(h_seq)){
    h0 = h_seq[i]
    CV_err =0
    for(i_cv in 1:k){
      w_val = which(cv_lab==(i_cv-1))
      X_tr = X[-w_val]
      Y_tr = Y[-w_val]
      X_val = X[w_val]
      Y_val = Y[w_val]
      kernel_reg = ksmooth(x = X_tr,y=Y_tr,kernel = "normal",bandwidth=h0,
                           x.points=X_val)
      # WARNING! The ksmooth() function will order the x.points from
      # the smallest to the largest!
      CV_err = CV_err+mean((Y_val[order(X_val)]-kernel_reg$y)^2,na.rm=T)
      # na.rm = T: remove the case of 'NA'
    }
    CV_err_h_tmp[i] = CV_err/k
  }
  CV_err_h = CV_err_h+CV_err_h_tmp
}
CV_err_h = CV_err_h/N_cv
plot(h_seq,CV_err_h, type="b", lwd=4, col="blue", xlab="Smoothing Bandwidth",
     ylab="5-CV Error")
h_opt = h_seq[which(CV_err_h==min(CV_err_h))]
h_opt=0.7
```

- Then we fit optimal bandwidth =0.7 for the simulated data and see the performance in figure-3 using the following code.

```
###############Fit optimal h= 0.7###
Kreg = ksmooth(x=X,y=Y,kernel = "normal",bandwidth = 0.7)
plot(X,Y,pch=20)
lines(Kreg, lwd=4, col="red")
```

- **Note:** Maybe the researchers find different optimal bandwidth h due to samplying fluctuation in the data set.

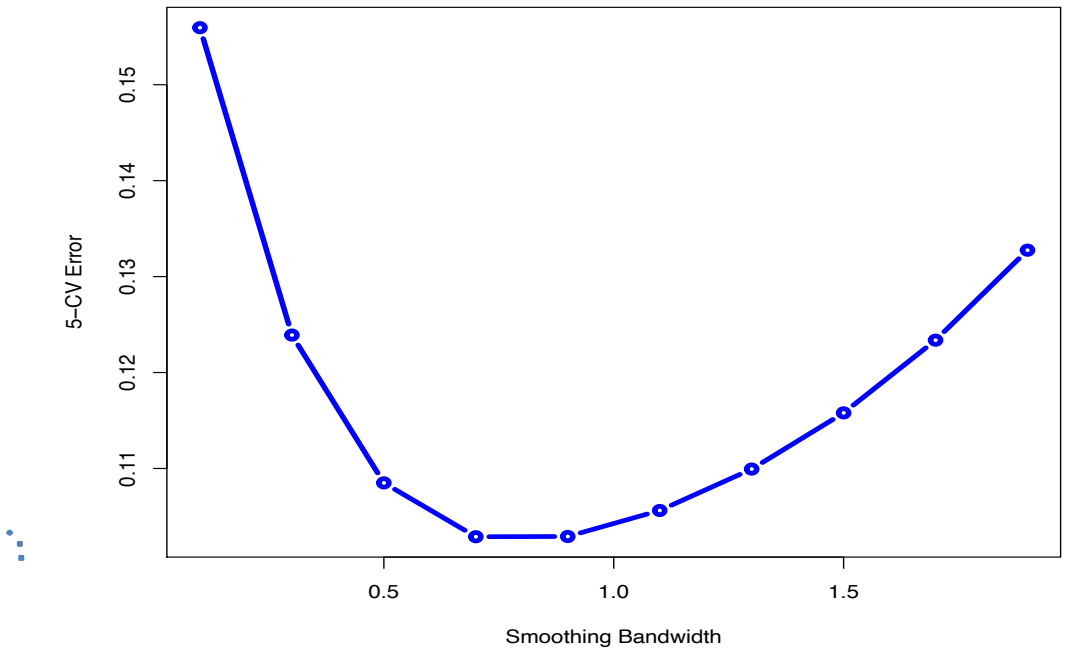


Figure 2: k=10 Cross-validation selection of smoothing for simulated data. The k=10 cross-validation criterion is shown a minimum is at h=0.7.

3 Splines Estimators

3.1 Smoothing Spline

Here we introduce another nonparametric regression method, the **smoothing spline** approach. The idea of smoothing spline is, we want to find a function $f(x))$ that fits the **data well** but also is **very smooth**. The model is

$$y_i \quad = \quad f(x_i) + \epsilon_i, \qquad i = 1, 2, 3, .., n$$

In the context of least square we might choose $\hat{f}$ to minimize the $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i))$. The solution is $\hat{f_{x_i}} = y_{i*}$ This is a join the dots regression that is almost certainly too rough. Instead, suppose we choose to minimize a modified least squares criterion:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i)) + \lambda \int_{x_{min}}^{x_{max}} | f''(x) |^2 \, dx$$

Where The quantity $\lambda > 0$ is quantity that is called smoothness penalty (or smoothing parameter in some literature) and $| f''(x) |^2$ is a roughness penalty.When f is rough, the penalty is large, but when f is smooth, the penalty is small. Thus the two parts of the criterion balance fit against smoothness. This is the **smoothing spline fit**. Note that the x_{min} and x_{max} are the minimum and maximum value of all the observed covariates.

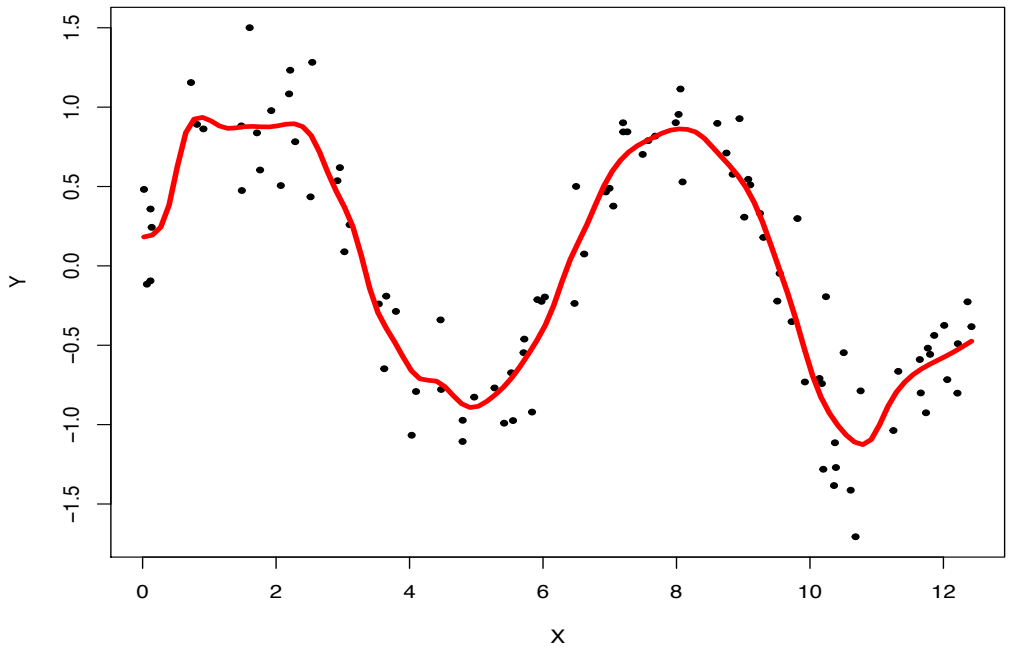


Figure 3: The use of optimal h=0.7 on simulated data set.

- For this choice of roughness penalty, the solution is of a particular form: is a cubic spline. This means that is a piecewise cubic polynomial in each interval (x_i, x_{i+1}). . It has the property $\hat{f}, \hat{f}'$, and $\hat{f}''$ that are continuous.

- Several variations on the basic theme of **roughness penalty** are possible such as penalties on **higher-order derivatives** lead to fits with more continuous derivatives. Also, We can use modefied the **sum of squares part** of the criterion. This feature is useful when smoothing splines are means to an end for some larger procedure that requires weighting. A robust version can be developed by modifying the sum of squares criterion to:

$$\frac{1}{n}\sum_{i=1}^{n}\rho(y_i - f(x_i)) + \lambda\int_{x_{min}}^{x_{max}} \mid f''(x) \mid^2 dx$$

where $\rho(x) =\mid x \mid$ is one possible choice.

Here are some examples of fitting the smoothing spline to the data under different values of λ. In R, we use the function **smooth.spline()** to fit a smoothing spline and the **argument spar** is how we can specify the value of λ in the above criterion: See the r-code below

```
########################################## fit smoothing splines ##########
SS1 = smooth.spline(x=X,y=Y,spar=0.3)
SS2 = smooth.spline(x=X,y=Y,spar=0.8)
```

```
SS3 = smooth.spline(x=X,y=Y,spar=1.2)
SS4 = smooth.spline(x=X,y=Y,spar=1.9)
plot(X,Y,pch=20)
lines(SS1, lwd=4, col="red")
lines(SS2, lwd=4, col="green")
lines(SS3, lwd=4, col="blue")
lines(SS4, lwd=4, col="purple")
legend("topright", c("spar=0.3","spar=0.8","spar=1.2","spar=1.9"), lwd=6,
       col=c("red","green","blue","purple"))
```

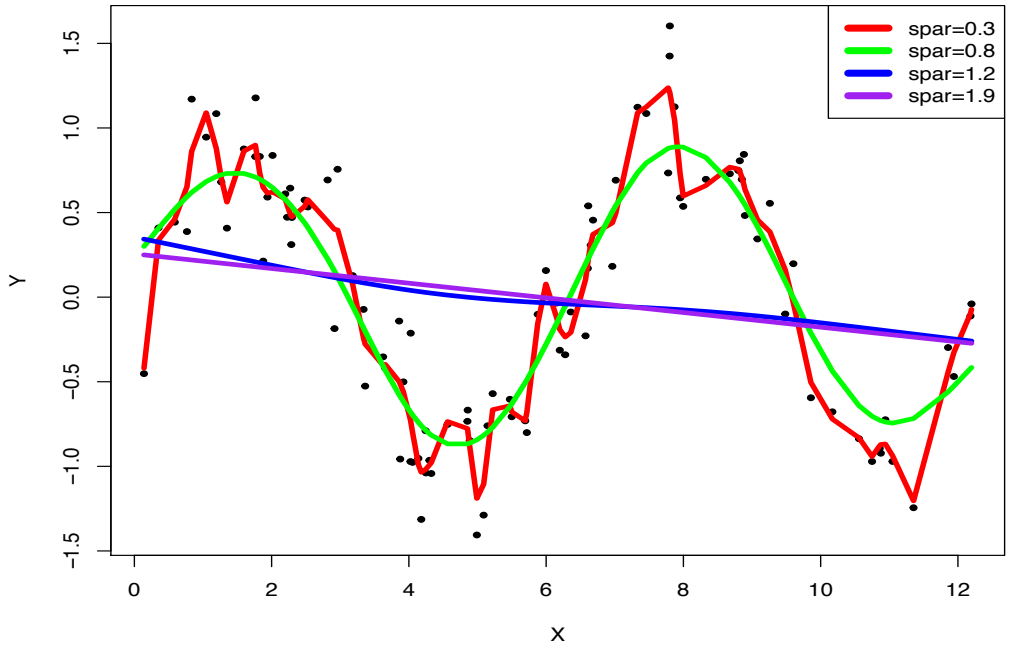
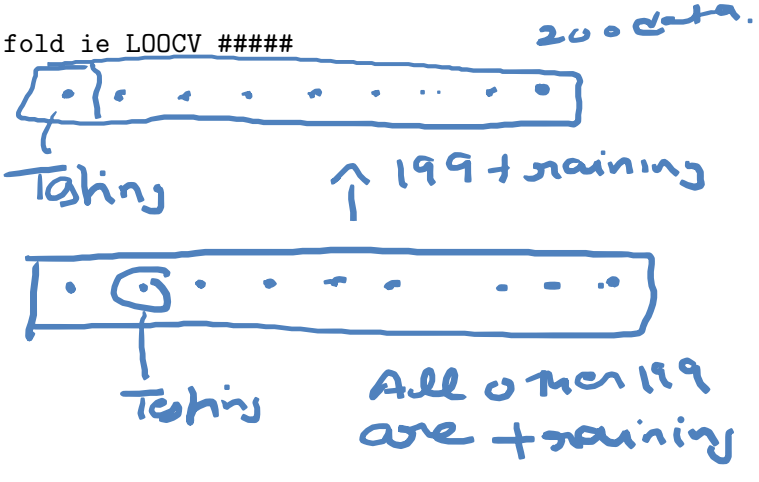
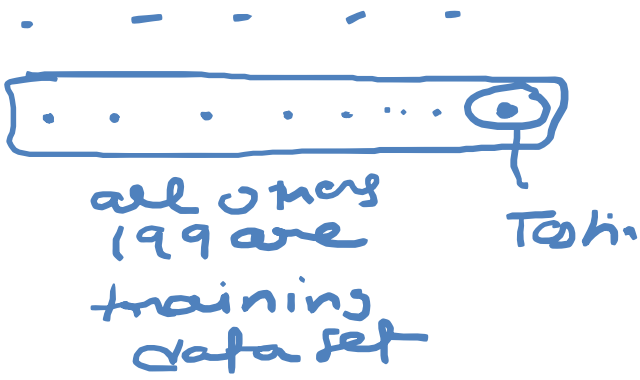


Figure 4: Smoothing spline fits for simulated data with four choices of λ.

Again,here a simplest form of cross-validation is the **leave-one-out cross-validation (LOOCV)**. The idea is: each time we leave one observation out as the validation set and we use the remaining data points to fit the data and then predict the value that we left out. Here is what we write in R:

```
##################################### k=1 fold ie LOOCV #####
n = length(X)
# n: sample size
sp_seq = seq(from=0.10,to=1.0, by=0.05)
# values of spar we are exploring
CV_err_sp = rep(NA,length(sp_seq))
for(j in 1:length(sp_seq)){
  spar_using = sp_seq[j]
```






```
  CV_err = rep(NA, n)
  for(i in 1:n){
    X_val = X[i]
    Y_val = Y[i]
    # validation set
    X_tr = X[-i]
    Y_tr = Y[-i]
    # training set
    SS_fit = smooth.spline(x=X_tr,y=Y_tr,spar=spar_using)
    Y_val_predict = predict(SS_fit,x=X_val)
    # we use the 'predict()' function to predict a new value
    CV_err[i] = (Y_val - Y_val_predict$y)^2
  }
  CV_err_sp[j] = mean(CV_err)
}
CV_err_sp
plot(x=sp_seq, y=CV_err_sp, type="b", lwd=3, col="blue",
     xlab="Value of 'spar'", ylab="LOOCV prediction error")
sp_seq[which(CV_err_sp == min(CV_err_sp))]=0.7
```

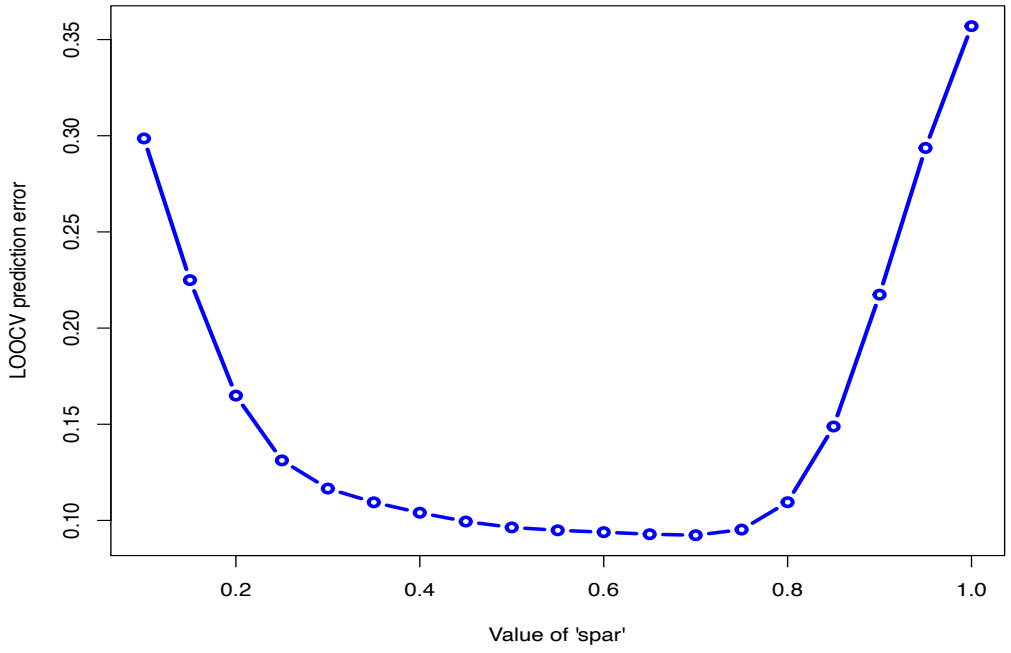


Figure 5: k=1 Cross-validation(LOOCV) selection of smoothing for simulated data. k=1 cross-validation criterion is shown a minimum is at λ=0.7.

3.2 Regression splines

Regression splines differ from smoothing splines in the following way: For regression splines, the **knots** of the **B-splines** used for the basis are typically much smaller in number than the sample size. The **number** of **knots** chosen **controls** the **amount** of smoothing. For **smoothing splines**, the observed unique **x values** are the knots and λ is used to control the smoothing. It is arguable whether the regression spline method is parametric or nonparametric, because once the knots are chosen, a parametric family has been specified with a finite number of parameters. It is the freedom to choose the number of knots that makes the method nonparametric. One of the desirable characteristics of a nonparametric regression estimator is that it should be consistent for smooth functions. This can be achieved for regression splines if the number of knots is allowed to increase at an appropriate rate with the sample size.
we demonstrate some regression splines here.

1. We use **piecewise linear splines** in this example, which are constructed and plotted as follows:

```
rm(list=ls())
library(faraway)
library(graphics)
######### Define Knots and compute a design matrix of splines with knots######
rhs <- function (x,c) ifelse (x>c, x-c, 0)
curve(rhs(x,0.5), 0,1)
knots <- 0:9/10
dm <- outer (exa$x,knots,rhs)
###matplot(exa$x,dm,type = c("b","p","o"))####
matplot(exa$x,dm,type = c("o"))

############### compute and display the regression fit:
g <- lm(exa$y ~ dm)
S1=plot (y ~ x, exa,pch=".",xlab="x",ylab="y")
lines(exa$x,predict(g),col="red")
################ Adjusting the knots ######

newknots = c(0,0.5,0.6,0.65,0.7,0.75,0.8,0.85,0.9,0.95)
dmn <- outer(exa$x,newknots,rhs)
gn <- lm(exa$y ~ dmn)
S2=plot(y ~ x, exa,pch=".",xlab="x",ylab="y")
lines(exa$x,predict(gn),col="green")
```

Note: fig-6 and fig-7 showed that a better fit may be obtained by adjusting the knots so that they are denser in regions of greater curvature. We obtain a better fit but only by using our knowledge of the true curvature. This knowledge would not be available for real data, so more practical methods place the knots adaptive according to the estimated curvature.

1. Another approach to achieve a smoother fit by using **higher-order splines**. The **bs ()** function can be used to generate the appropriate spline basis. The default is cubic B-splines. We display 12 cubic B-splines evenly spaced on the [0, 1] interval. We then display the fit as seen in the fig-8

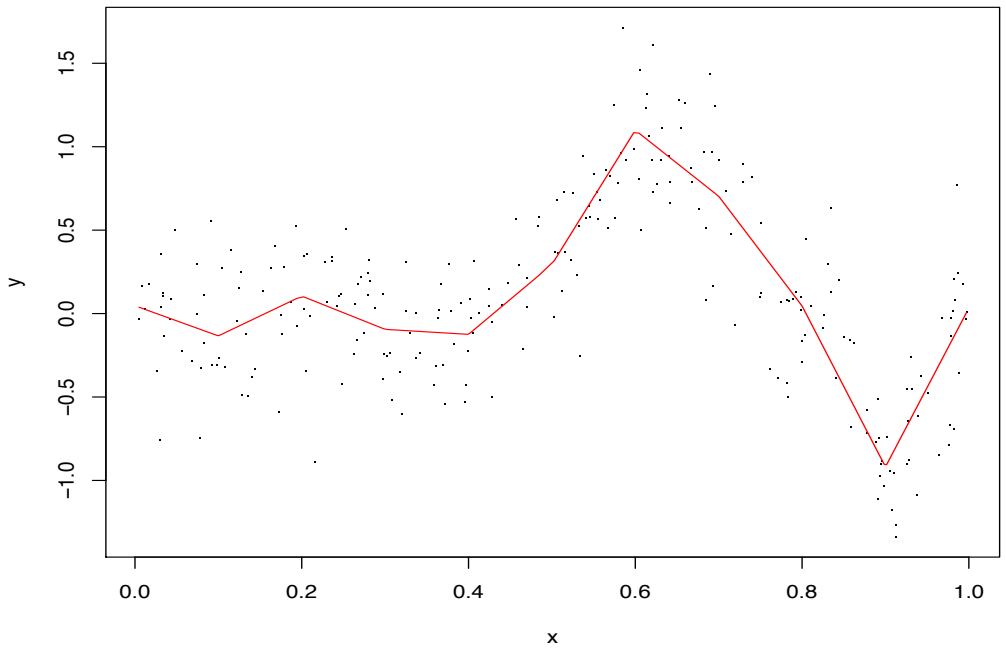


Figure 6: Evenly spaced knots fit.

```
 ##########smoother fit by using higher-order splines#########
library(splines)
matplot(bs(seq(0,1,length=1000),df=12),type="o",ylab="",col=1)
sml=lm(y ~ bs(x,12),exa)
plot (y ~ x, exa, pch=".")
lines(m ~ x, exa)
lines(predict(sml) ~ x, exa, lty=2)
```

Note: We see a smooth fit, but again we could do better by placing more knots at the points of high curvature and fewer in the flatter regions

4 Local Polynomials

Both the kernel and spline methods are not good for the data set that have outliers. In this situation, The local polynomial method is appropriate that combines robustness ideas from linear regression and local fitting ideas from kernel methods. Cleveland (1979) proposed the idea of local regression to estimate the unknown function $f(.)$. The local regression estimate of the function $f(.)$ evaluated at the point x is constructed by fitting a weighted least squares regression model to the y_i values corresponding to x_i values that are nearby the given x. The local regression estimate can be written as

$$\hat{f}(x) = \hat{\beta}_0 + \hat{\beta}_1 x$$

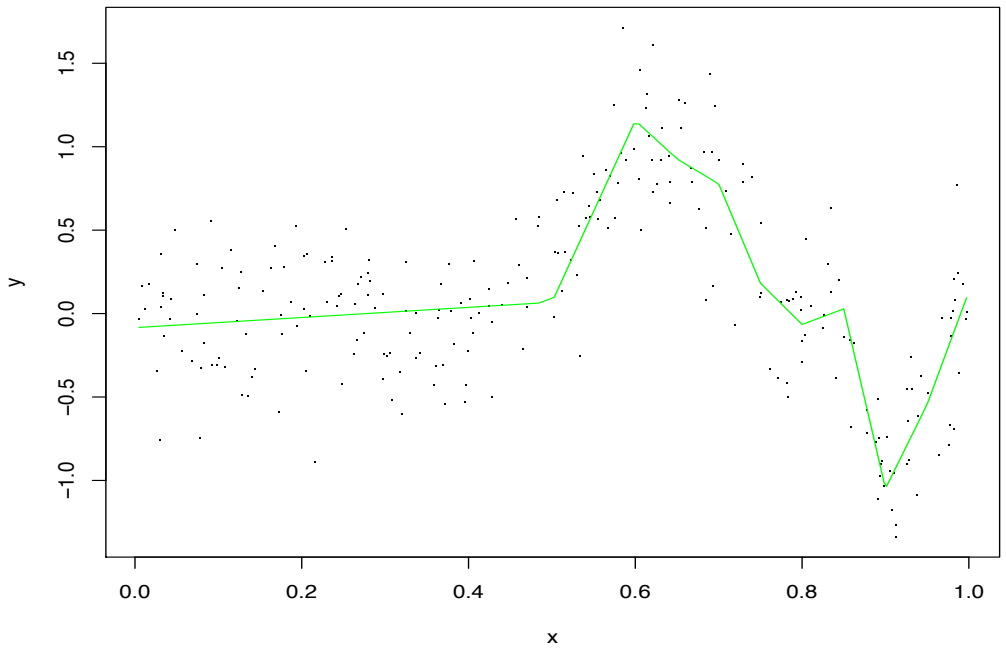


Figure 7: Adjusting the knots fit.

where $\hat{\beta}_0$ and $\hat{\beta}_1$ are the minimizers of the weighted least squares problem

$$\sum_{i=1}^{n} w_i(x)(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2$$

Note that $\hat{\beta}_0$ and $\hat{\beta}_1$ are functions of x given that the weights $w_i(x)$ are a function of x.
In this case, the weights are defined such that $w_i(x)$ is a non-increasing function of $|x - x_i|$ that takes nonzero values only when x_i is nearby x. A popular weight function is the tricube function

$$w_i(x) = \{ \begin{array}{cc} (1- \mid x - x_i \mid^3)^3 & |x - x_i| < \delta_i \\ 0 & otherwise \end{array} \}$$

where $\delta_i > 0$ is some scalar that is used to determine which points are close enough to the given x to receive a non-zero weight.
The above formulation is for local linear regression, where a simple linear regression model is fit in the neighborhood of x. The idea could be easily extended to include high-order polynomial terms. For example, the local quadratic regression estimate has the form

$$\hat{f}(x) = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2$$

where $\{\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2\}$ are the minimizers of the weighted least squares problem

$$\sum_{i=1}^{n} w_i(x)(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i - \hat{\beta}_2 {x_i}^2)^2$$

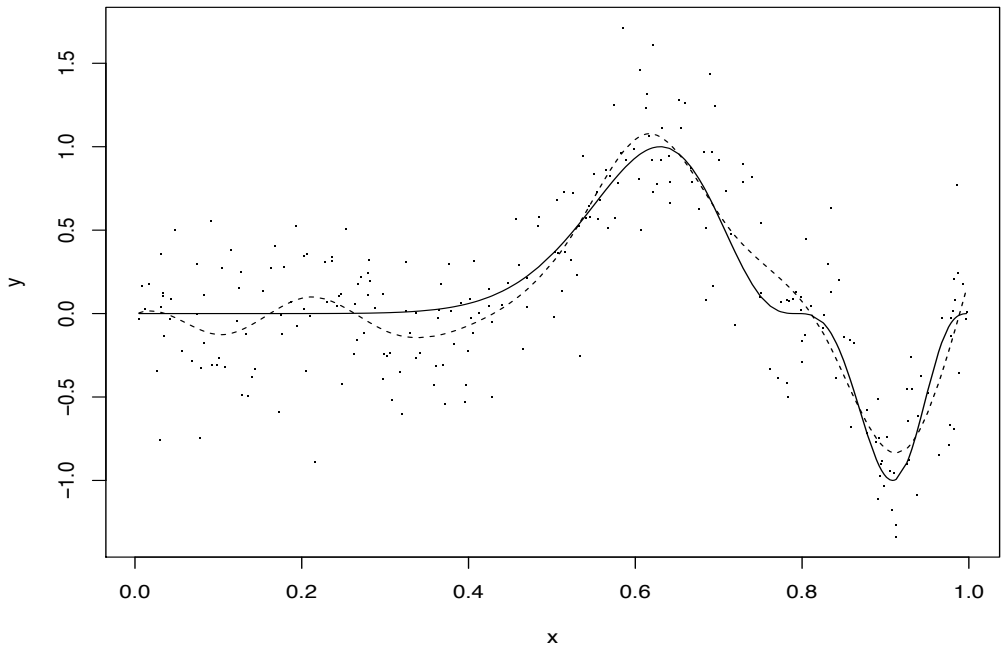


Figure 8: A cubic B-spline basis fit.

similarly the local p^{th} order regression estimate has the form

$$\hat{f}(x) = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2 + ... + \hat{\beta}_p x^p$$

- **LOWESS(LOcally WEighted Scatterplot Smoothing)** for univariate smoothing and **LOESS(LOcal regrESSion)** is for fitting a smooth surface to multivariate data.

- **How Close is Nearby?** Cleveland proposed defining δ_i such that the weights have non-zero values for sn observations, where $s(0, 1]$ is the span parameter. Assuming that the x_i are uniformly spread between $a = min(x_i)$ and $b = max(x_i)$, we have that $\delta_i = \frac{s}{2}$ for points x_i that are sufficiently far from the boundary points a and b. The r-code is below and performance can be seen in fig-9.

```
 rm(list=ls())
library(faraway)
library(graphics)
set.seed(112102)
n <- 50
x <- seq(0, 1, length.out = n)
fx <- sin(2 * pi * x)
y <- fx + rnorm(n, sd = 0.5)

# Define tricube weight function
```

```
tricube <- function(x, delta = 0.2) {
  ifelse(abs(x) < delta, (1 - abs(x)^3)^3, 0)
}

# plot tricube weight function
xstar <- 0.4
plot(x, tricube(x - xstar, 0.2 / 2), t = "l", ylab = "W(x)",
     cex.lab = 1.5, cex.axis = 1.25)
deltas <- c(0.3, 0.4)
for(k in 1:2){
  lines(x, tricube(x - xstar, deltas[k] / 2), lty = k + 1)
}
legend("topright", legend = c(expression(delta * " = 0.10"),
                              expression(delta * " = 0.15"),
                              expression(delta * " = 0.20")),
       lty = 1:5, bty = "n")
```

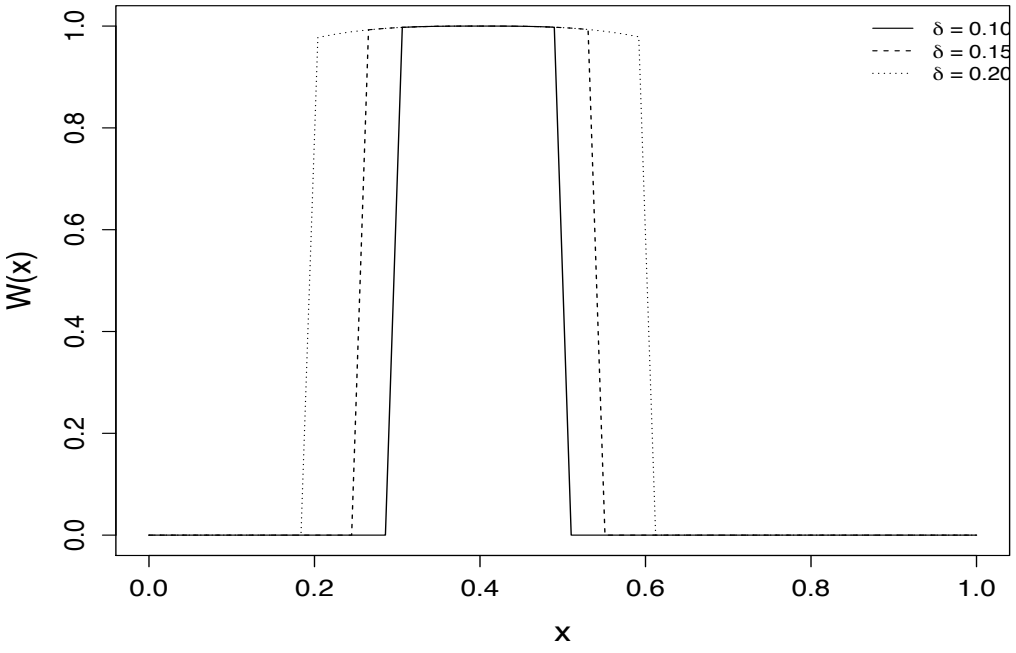


Figure 9: Graphical representation of tricube weight function.

.

- **Visual Intuition**:To get an understanding of the **local regression estimator**, we will look at how the estimate is formed for the point $x^* = 0.4$ using different values of the span parameter. We will start by looking at the LOESS estimator using local

linear regression, and then we will look at the LOESS estimator using **local quadratic regression**. We used the following r-code to prepare **fig-10**.

```
 #################local linear regression with LOESS ####
xstar <- 0.4
cols <- rgb(190/255,190/255,190/255,alpha=0.5)

# set-up 2 x 2 subplot
par(mfrow = c(2,2))

# loop through spans ( 0.2, 0.3, 0.4)
for(s in c( 0.2, 0.3, 0.4)){

  # plot data and true function
  plot(x, y, main = paste0("span = ", s), ylim = c(-2.8, 2.8),
       cex.lab = 1.5, cex.axis = 1.25)
  lines(x, fx, col = "blue", lwd = 2)

  # plot window
  window <- c(xstar - s / 2, xstar + s / 2)
  rect(window[1], -3, window[2], 3, col = cols)

  # define weights
  w <- tricube(x - xstar, delta = s / 2)

  # plot estimate
  X.w <- sqrt(w) * cbind(1, x)
  y.w <- sqrt(w) * y
  beta <- solve(crossprod(X.w)) %*% crossprod(X.w, y.w)
  ystar <- as.numeric(cbind(1, xstar) %*% beta)
  points(xstar, ystar, pch = 17, col = "red", cex = 1)

  # add regression line
  abline(beta, lty = 3)

  # add legend
  legend("topright", legend = c("data", "truth"),
         pch = c(1, NA), lty = c(NA, 1), col = c("black", "blue"), bty = "n")
  legend("bottomright", legend = c("estimate", "window"),
         pch = c(17, 15), col = c("red", "gray"), bty = "n")

}
```

. **Note:** The estimate (denoted by the **red triangle**) is calculated by fitting a **weighted linear regression model** to the y_i values that are within the local window (denoted by the gray box). The above example shows the window for the point $x^* = 0.4$, but the same idea applies to other points. To form the function estimate at a different x value,

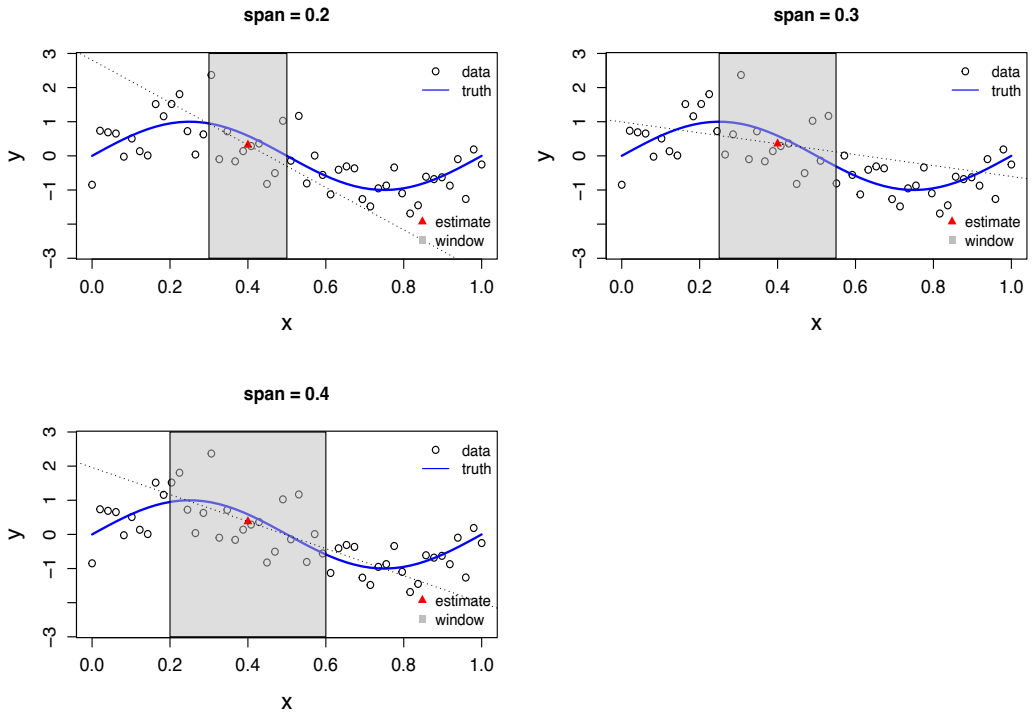


Figure 10: Local linear regression estimate of f(0.4) with different span values.

we would just slide the window down the x-axis to be centered at the new x value, and redefine the weights accordingly.

- For **Local quadratic** regression estimate, we prepared the following r-code and see the performance in **fig-11**.

```
###################Local Quadratic Regression ######
# define x* and color for window
xstar <- 0.4
cols <- rgb(190/255,190/255,190/255,alpha=0.5)

# set-up 2 x 2 subplot
par(mfrow = c(2,2))

# loop through spans ( 0.2, 0.3, 0.4)
for(s in c( 0.2, 0.3, 0.4)){

  # plot data and true function
  plot(x, y, main = paste0("span = ", s), ylim =c(-2.8, 2.8),
       cex.lab = 1.5, cex.axis = 1.25)
  lines(x, fx, col = "blue", lwd = 2)

  # plot window
```

```
  window <- c(xstar - s / 2, xstar + s / 2)
  rect(window[1], -3, window[2], 3, col = cols)

  # define weights
  w <- tricube(x - xstar, delta = s / 2)

  # plot estimate
  X <- cbind(1, x - 0.5, (x - 0.5)^2)
  X.w <- sqrt(w) * X
  y.w <- sqrt(w) * y
  beta <- solve(crossprod(X.w)) %*% crossprod(X.w, y.w)
  ystar <- as.numeric(cbind(1, xstar - 0.5, (xstar - 0.5)^2) %*% beta)
  points(xstar, ystar, pch = 17, col = "red", cex = 1)

  # add regression line
  lines(x, X %*% beta, lty = 3)

  # add legend
  legend("topright", legend = c("data", "truth"),
         pch = c(1, NA), lty = c(NA, 1), col = c("black", "blue"), bty = "n")
  legend("bottomright", legend = c("estimate", "window"),
         pch = c(17, 15), col = c("red", "gray"), bty = "n")

}
```

Note: The estimate (denoted by the red triangle) is calculated by fitting a **weighted quadratic** regression model to the y_i values that are within the local window (denoted by the gray box). The above example shows the window for the point x^* , but the same idea applies to other points. To form the function estimate at a different x value, we would just slide the window down the x-axis to be centered at the new x value, and redefine the weights accordingly.

- **Note**: Similarly, we can apply same strategy on **Local p^{th} polynomial linear regression** estimate.

5 Wavelets Estimator

In **wavelets estimators analysis**, We approximate the curve by a family of basis functions,$\phi_i(x)$ so that $\hat{f}(x) = \sum_{i=1}^{n} c_i \phi_i(x)$. Thus the fit requires **estimating the coefficients**,c_i. The **choice of basis functions** will determine the properties of the fitted curve. The estimation of c_i is particularly easy if the basis functions are orthogonal. **Wavelets** have the **advantage** over **cubic B-splines** and **Fourier Methods** as Wavelets are compactly supported and can be defined so as to possess the **orthogonality** property. They also possess the **multiresolution** property while Cubic B-splines are compactly supported, but they are not orthogonal and Fourier methods are popular for some applications, they are not typically used for general-purpose smoothing. The **mother wavelet algorithm** for the **Haar family** is defined in the following steps

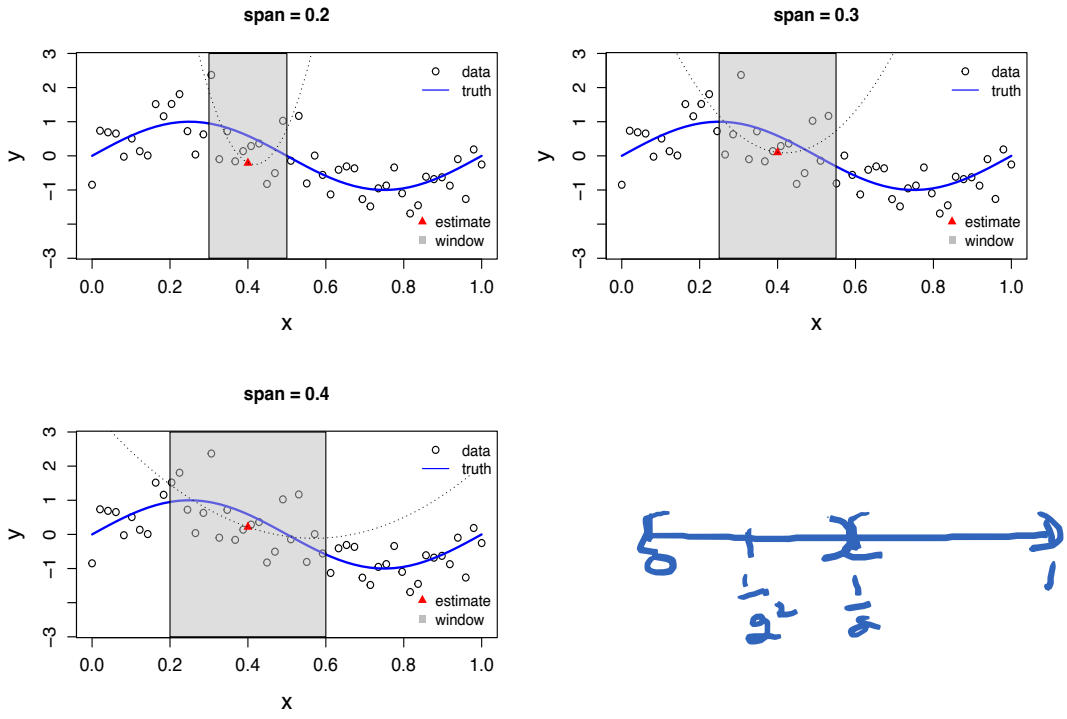


Figure 11: Local Quadratic linear regression estimate of f(0.4) with different span values.

1. mother wavelet defined on $[0,1)$ as

$$w(x) = \{ \begin{matrix} 1 & x \leq 1/2 \\ -1 & x > 1/2 \end{matrix} \}$$

2. In next step, two members of the family are defined on $[0,1/2)$ and $[1/2,1)$ by rescaling the mother wavelet to these two intervals.

3. Finally, we can index the family members by level j and within the level by k so that each function will be defined on the interval $[k/2^j,(k+1)/2^j)$ and takes the form: $h_n(x) = 2^{j/2}w(2^j x - k)$ where $n = 2^j + k$ and $0 \leq k \leq 2^j$. We can see by simply plotting these functions that they are orthogonal and orthonormal. Furthermore, they have a local basis where the support becomes narrower as the level is increased. Computing the coefficients is particularly quick because of these properties.

Note: **We did little practical analysis with the following r-code and see fig-12 and fig-13**.

```
############# Wavlets#######
######################
rm(list=ls())
library(wavethresh)
wds <- wd (exa$y, filter.number=10)
draw(wds,main="")
plot (wds, main="")
```

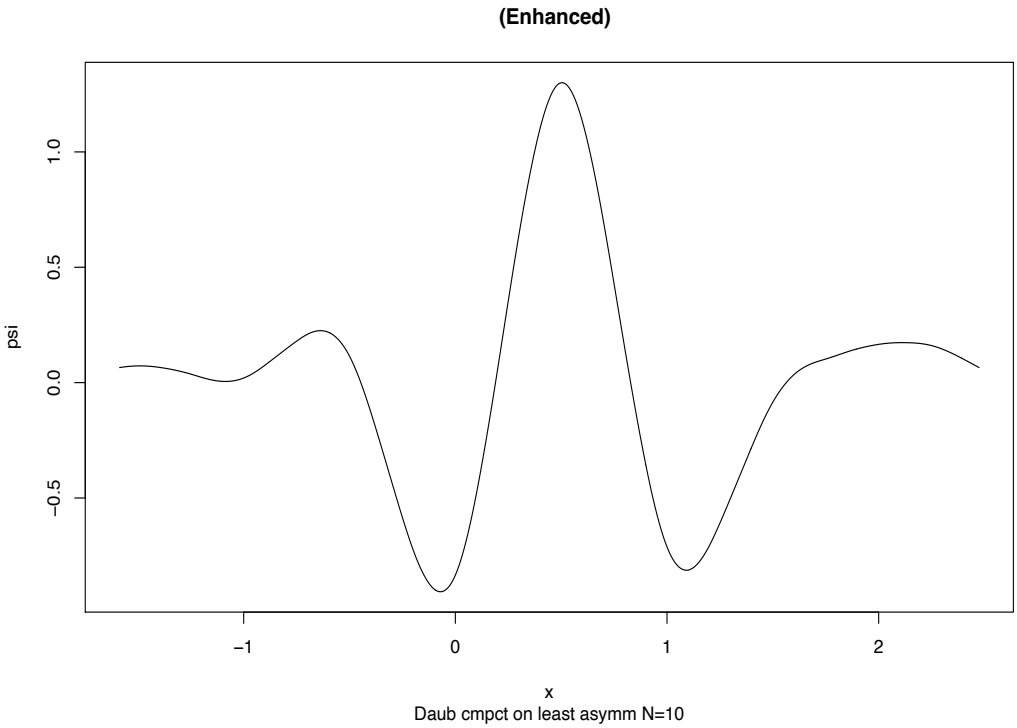


Figure 12: Haar mother wavelet from decomposition for Example A when N=10.

6 Comparison of Methods

In the **univariate** case, we can describe **three** situations.

1. When there is very **little noise**, **interpolation** (or at most, very mild smoothing) is the best way to recover the relation between x and y.

2. When there is a **moderate** amount of noise, **nonparametric** methods are most effective.

3. When the amount of noise becomes **larger**, **parametric** methods become relatively more attractive.

It is not reasonable to claim that any one **smoother** is better than the rest. The **best choice** of smoother will depend on the **characteristics** of the **data** and knowledge about the true underlying relationship. The choice will also depend on whether the fit is to be made automatically or with human intervention. When only a single dataset is being considered, its simple enough to craft the fit and intervene if a particular method produces unreasonable results. If a large number of datasets are to be fit automatically, then human intervention in each case may not be feasible. In such cases, a reliable and robust smoother may be needed. We think the **loess smoother** makes a good all-purpose smoother. It is robust to **outliers** and yet can produce smooth fits. When you are confident that **no outliers** are present, **smoothing splines** is more efficient than local polynomials.

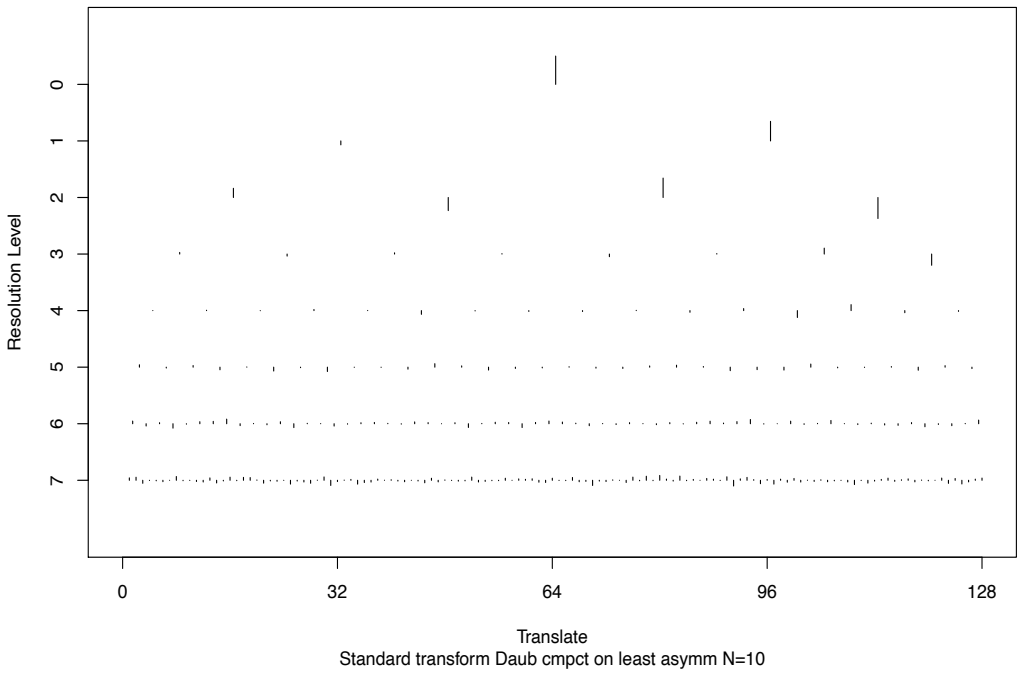


Figure 13: Wavelet coefficients from decomposition for Example A when N=10