Kernel Smoothing

Ruoqing Zhu

Last Updated: October 03, 2024

Contents

KNN vs. Kernel

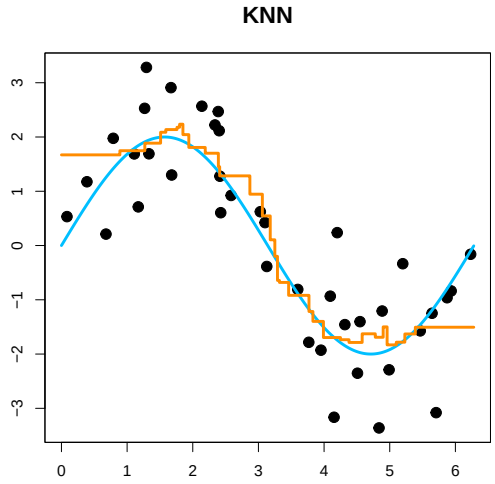
The main goal of this week is to learn a new, local smoothing estimator, the Nadaraya-Watson kernel regression. It has the following form, with a tuning parameter h, called the bandwidth.

$$\widehat{f}(x) = \frac{\sum_i K_h(x, x_i) y_i}{\sum_i K_h(x, x_i)}.$$

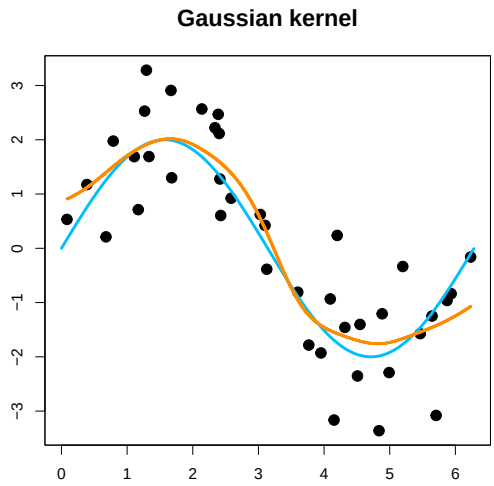
In practice, we can choose different versions of the kernel function $K_h(\cdot, \cdot)$. We will introduce details later, but let's first compare the KNN method with a Gaussian kernel regression. KNN has jumps while Gaussian kernel regression is smooth. And in fact, even we increase k, it would still be non-smooth. What causes this difference? For this example, we will use the `locpoly()` function from the `KernSmooth` package.

```
# generate some data
set.seed(1)
x <- runif(40, 0, 2*pi)
y <- 2*sin(x) + rnorm(length(x))
testx = seq(0, 2*pi, 0.01)

# compare two methods: KNN or Kernel regression
library(kknn)
knn.fit = kknn(y ~ x, train = data.frame("x" = x, "y" = y),
               test = data.frame("x" = testx), k = 10, kernel = "rectangular")

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

```r
    # true function
    lines(testx, 2*sin(testx), col = "deepskyblue", lwd = 3)

    # KNN estimated function
    lines(testx, knn.fit$fitted.values, type = "s", col = "darkorange", lwd = 3)
    box()
```




```r
    # we use the locpoly() function from the KernSmooth package
    # please be careful about the parameter "bandwidth"
    # This will be discussed later
    library(KernSmooth)
## Warning: package 'KernSmooth' was built under R version 4.3.3
## KernSmooth 2.23 loaded
## Copyright M. P. Wand 1997-2009
    NW.fit = locpoly(x, y, degree = 0, bandwidth = sd(x)*0.25,
                     kernel = "normal")

    plot(x, y, xlim = c(0, 2*pi), cex = 1.5, xlab = "", ylab = "",
         cex.lab = 1.5, pch = 19)
    title(main=paste("Gaussian kernel"), cex.main = 1.5)

    # the true function
    lines(testx, 2*sin(testx), col = "deepskyblue", lwd = 3)

    # Kernel estimated function
    lines(NW.fit$x, NW.fit$y, type = "s", col = "darkorange", lwd = 3)
    box()
```



Gaussian Kernel Regression

First, let's introduce a kernel function. In most applications, we will consider using density functions as a kernel. And for the most part of this lecture, we only consider 1-dimensional kernels. Then we have

$$K_h(u, v) = K(|u - v|/h)/h$$

where the function $K(\cdot)$ is a density function of a random variable. h has a very important role, but we will introduce that later. For now, if we consider the standard normal distribution density function (pdf), we have

$$K(t) = \frac{1}{\sqrt{2\pi}} \exp\left\{-t^2/2\right\}.$$

Then we can plug-in $t = |u - v|/h$. For any two data points u and v. This leads to

$$K_h(u, v) = \frac{1}{h\sqrt{2\pi}} \exp\left\{-\frac{(u - v)^2}{2h^2}\right\}.$$

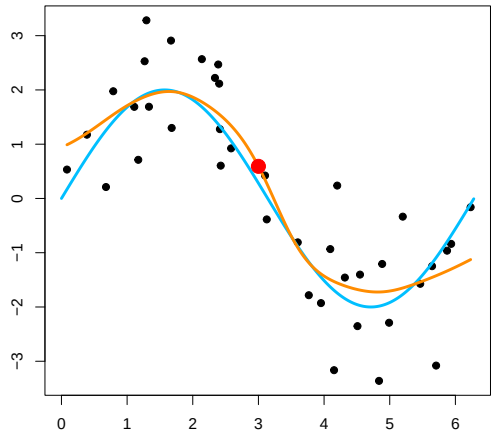
Recall that the Nadaraya-Watson (NW) kernel regression is

$$\widehat{f}(x_0) = \frac{\sum_i K_h(x_0, x_i) y_i}{\sum_i K_h(x_0, x_i)},$$

The quantity $K_h(x_0, x_i)$ is the **kernel weight** of the ith subject when estimating the function value at target point x_0. Let's calculate this quantity and the NW estimator at a target point using our own code:

```
# the target point
x0 = 3

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

```r
# the kernel weights for each observation
# lets use a bandwidth = 0.5
h = 0.5
# calculate the kernel weights
w = dnorm( (x0 - x)/h )/h
# calculate the NW estimator
fhat = sum(w*y)/sum(w)

est <- locpoly(x, y, degree = 0, bandwidth = h,
               kernel = "normal")

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

# the true function
lines(testx, 2*sin(testx), col = "deepskyblue", lwd = 3)

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

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

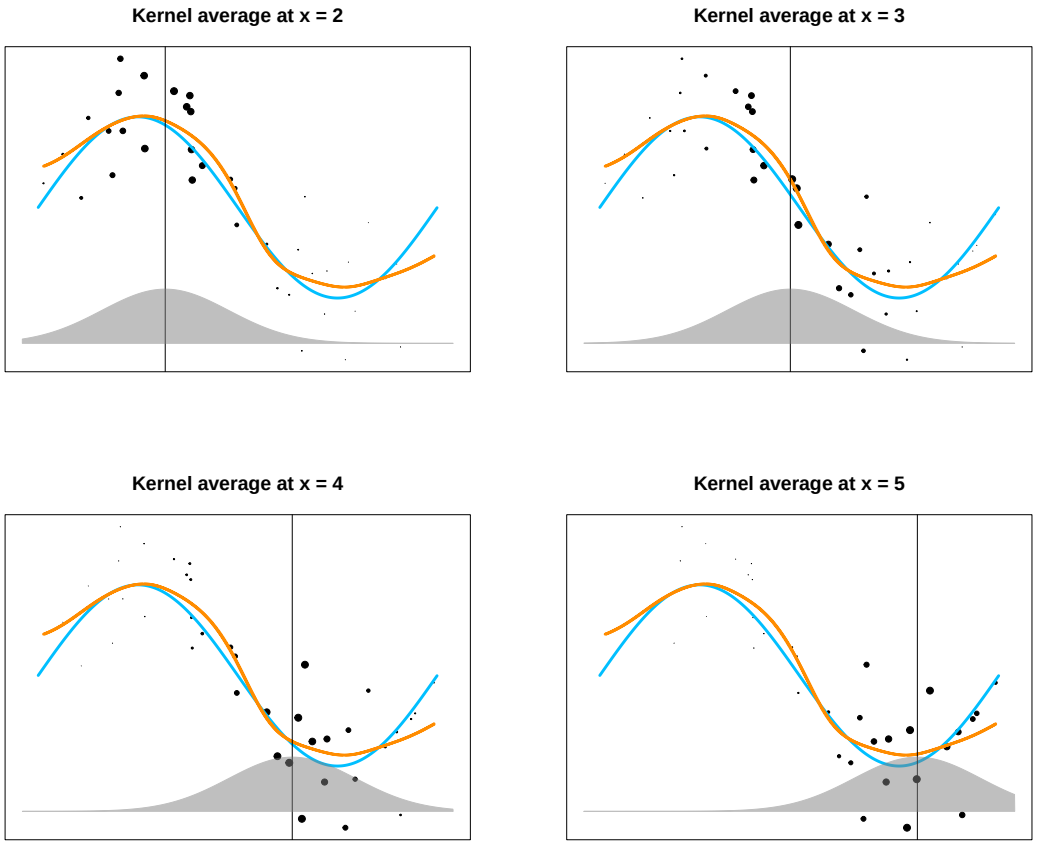


Understanding the Local Averaging

At each target point x, training data x_is that are closer to x receives higher weights $K_h(x, x_i)$, hence their y_i values are more influential in terms of estimating $f(x)$.

```r
par(mfrow = c(2, 2))

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

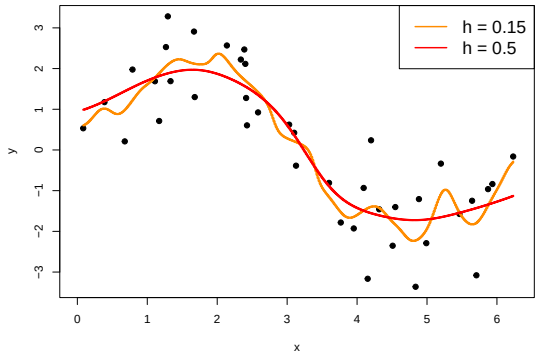
```
testx = seq(0, 2*pi, 0.01)

# plots for different h values
for (x0 in c(2, 3, 4, 5))
{
    # local points, with size proportional to their influence
    plot(x, y, xlim = c(-0.25, 2*pi+0.25), cex = 3*dnorm(x, x0), xlab = "", ylab = "",
         cex.lab = 1.5, pch = 19, xaxt='n', yaxt='n')
    title(main=paste("Kernel average at x =", x0), cex.main = 1.5)

    # the true function
    lines(testx, 2*sin(testx), col = "deepskyblue", lwd = 3)

    # the estimated function
    lines(NW.fit$x, NW.fit$y, type = "s", col = "darkorange", lwd = 3)

    # the target point
    abline(v = x0)

    # the local density around the target point
    # The Gaussian Kernel Function in the shaded area
    cord.x <- seq(-0.25, 2*pi+0.25, 0.01)
    cord.y <- 3*dnorm(cord.x, x0) - 3 # Gaussian density with h = 1

    polygon(c(-0.25, cord.x, 2*pi+0.25),
            c(-3, cord.y, -3),
            col=rgb(0.5, 0.5, 0.5, 0.5),
            border = rgb(0.5, 0.5, 0.5, 0.5))
}
```



Bias-variance Trade-off

The bandwidth h is an important tuning parameter that controls the bias-variance trade-off. It behaves similarly as the KNN. By setting a large h, the estimator is more stable but has more bias.

```
# a small bandwidth
NW.fit1 = locpoly(x, y, degree = 0, bandwidth = 0.15, kernel = "normal")

# a large bandwidth
NW.fit2 = locpoly(x, y, degree = 0, bandwidth = 0.5, kernel = "normal")

# plot both
plot(x, y, xlim = c(0, 2*pi), pch = 19)
lines(NW.fit1$x, NW.fit1$y, type = "s", col = "darkorange", lwd = 3)
lines(NW.fit2$x, NW.fit2$y, type = "s", col = "red", lwd = 3)
legend("topright", c("h = 0.15", "h = 0.5"), col = c("darkorange", "red"),
       lty = 1, lwd = 2, cex = 1.5)
```



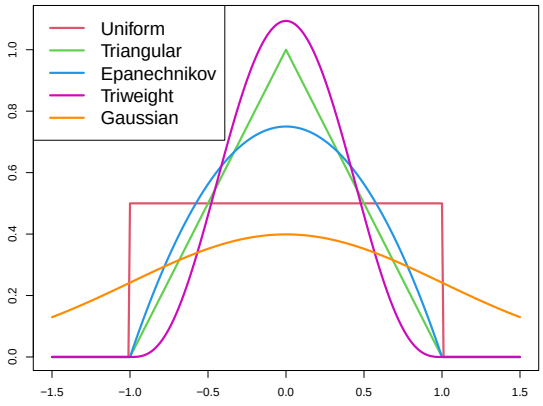
There is a more rigorous explanation of how the bias-variance happened, provided in the SMLR text book. The derivation is using the kernel density estimation as an example, however, this is essentially the same since the regression function estimation is the expectation of y with the joint density $p(x, y)$ over the marginal density $p(x)$, i.e., $E(Y|X) = \int yp(x, y)/p(x)dy$.

The effect of h is similar as k in KNN:

- As h increase, we are using a "larger" neighborhood (more points receive significantly heavy weights) of the target point x_0 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.

Choice of Kernel Functions

Other kernel functions can also be used. The most efficient kernel is the Epanechnikov kernel. Different kernel functions can be visualized in the following. Most kernels are bounded within $[-h/2, h/2]$, except the Gaussian kernel.

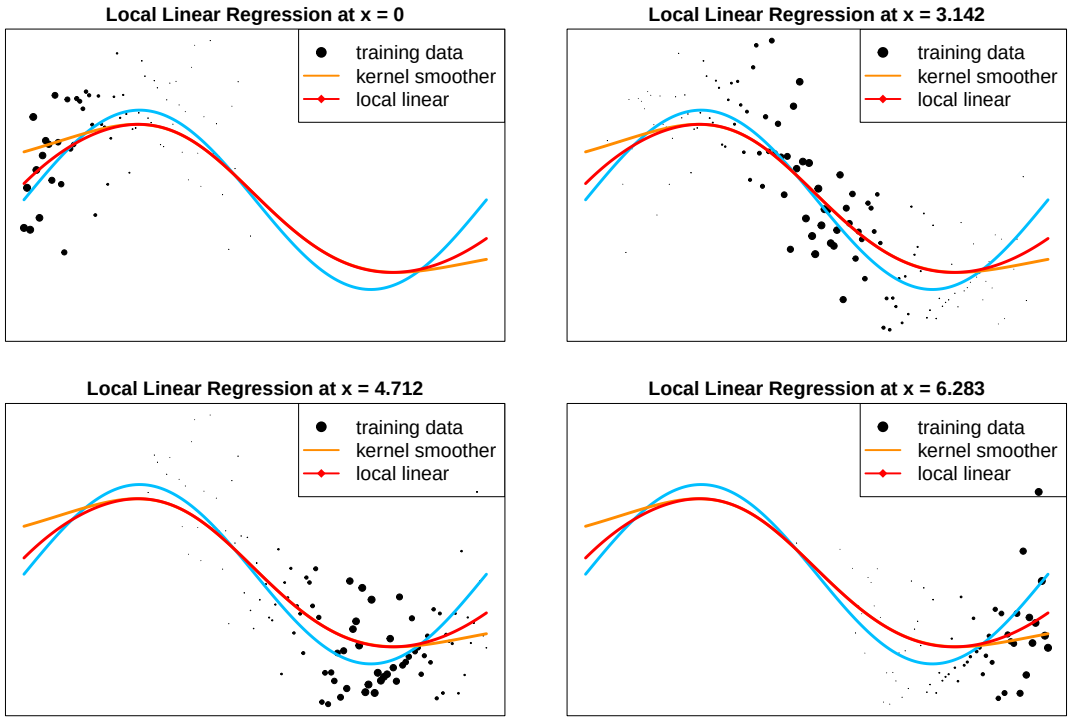


Local Linear Regression

Local averaging will suffer severe bias at the boundaries. One solution is to use the local polynomial regression. The following examples are local linear regressions, evaluated as different target points. We are solving for a linear model weighted by the kernel weights:

$$\sum_{i=1}^{n} K_h(x, x_i)\big(y_i - \beta_0 - \beta_1 x_i\big)^2$$

```r
# generate some data
set.seed(1)
n = 150
x <- seq(0, 2*pi, length.out = n)
y <- 2*sin(x) + rnorm(length(x))

#Silverman optimal bandwidth for univariate regression
h = 1.06*sd(x)*n^(-1/5)

par(mfrow = c(2, 2), mar=rep(2, 4))

for (x0 in c(0, pi, 1.5*pi, 2*pi))
{
    # Plotting the data
    plot(x, y, xlim = c(0, 2*pi), cex = 3*h*dnorm(x, x0, h), xlab = "", ylab = "",
         cex.lab = 1.5, pch = 19, xaxt='n', yaxt='n')
    title(main=paste("Local Linear Regression at x =", round(x0, 3)), cex.main = 1.5)
    lines(x, 2*sin(x), col = "deepskyblue", lwd = 3)

    # kernel smoother
    NW.fit = locpoly(x, y, degree = 0, bandwidth = h, kernel = "normal")
    lines(NW.fit$x, NW.fit$y, type = "l", col = "darkorange", lwd = 3)

    # local linear
    locallinear.fit = locpoly(x, y, degree = 1, bandwidth = h, kernel = "normal")
    lines(locallinear.fit$x, locallinear.fit$y, type = "l", col = "red", lwd = 3)

    legend("topright", c("training data", "kernel smoother", "local linear"),
           lty = c(0, 1, 1), col = c(1, "darkorange", "red"),
           lwd = 2, pch = c(19, NA, 18), cex = 1.5)
}
```

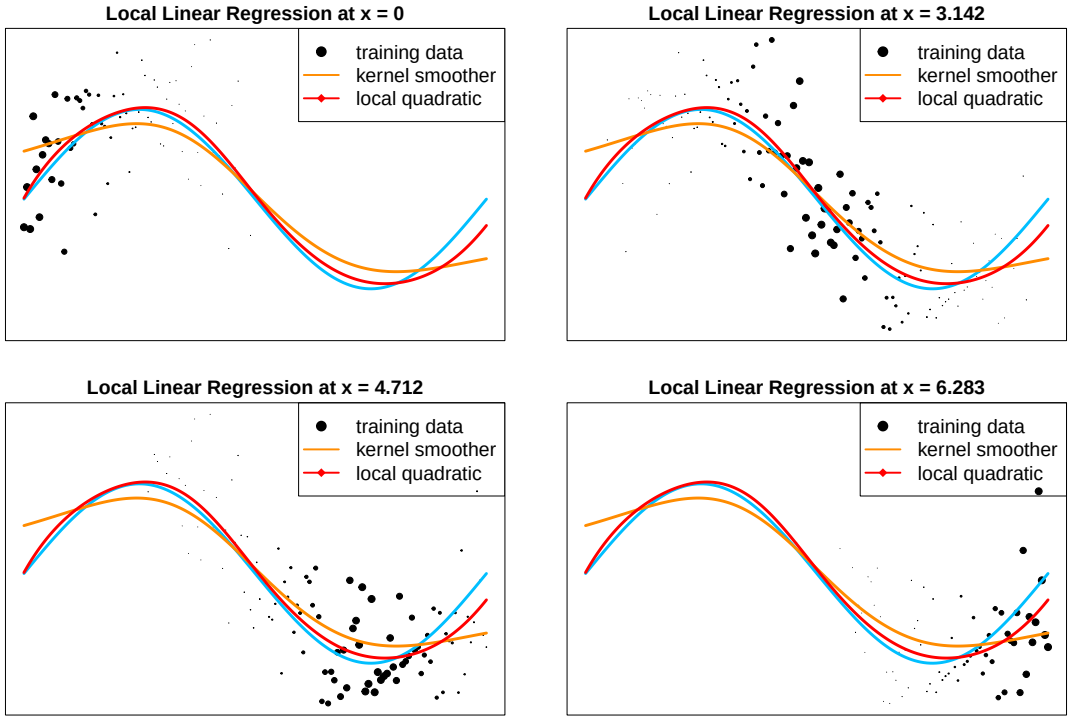


We can further consider a local qudratic fitting:

$$\frac{1}{n}\sum_{i=1}^{n} K_h(x, x_i)\big(y_i - \beta_0 - \beta_1 x_i - \beta_2 x_i^2\big)^2$$

```
# generate some data
set.seed(1)
n = 150
x <- seq(0, 2*pi, length.out = n)
y <- 2*sin(x) + rnorm(length(x))

#Silverman rule of thumb for univariate regression
h = 1.06*sd(x)*n^(-1/5)

par(mfrow = c(2, 2), mar=rep(2, 4))

for (x0 in c(0, pi, 1.5*pi, 2*pi))
{
    # Plotting the data
    plot(x, y, xlim = c(0, 2*pi), cex = 3*h*dnorm(x, x0, h), xlab = "", ylab = "",
         cex.lab = 1.5, pch = 19, xaxt='n', yaxt='n')
    title(main=paste("Local Linear Regression at x =", round(x0, 3)), cex.main = 1.5)
    lines(x, 2*sin(x), col = "deepskyblue", lwd = 3)

    # kernel smoother
    NW.fit = locpoly(x, y, degree = 0, bandwidth = h, kernel = "normal")
    lines(NW.fit$x, NW.fit$y, type = "l", col = "darkorange", lwd = 3)
```

```r
    # local linear
    locallinear.fit = locpoly(x, y, degree = 2, bandwidth = h, kernel = "normal")
    lines(locallinear.fit$x, locallinear.fit$y, type = "l", col = "red", lwd = 3)

    legend("topright", c("training data", "kernel smoother", "local quadratic"),
           lty = c(0, 1, 1), col = c(1, "darkorange", "red"),
           lwd = 2, pch = c(19, NA, 18), cex = 1.5)
}
```



Multi-dimensional Kernels

The NW kernel regression can also be used for multi-dimensional data. The idea is to utilize a certain distance measure, for example, the Euclidean distance:

$$d(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|_2 = \sqrt{\sum_{j=1}^{p} (\mathbf{u}_i - \mathbf{v}_i)^2}$$

Then, we can still plugin this quantity to the kernel function.

$$K_h(\mathbf{u}, \mathbf{v}) = K(\|\mathbf{u} - \mathbf{v}\|_2 / h)/h$$

However, be careful that this model also suffers from the curse of dimensionality, similar to KNN. And the logic behind this is the same. Hence, we need to adjust h accordingly.

R Implementations

Some popular `R` functions implements the local polynomial regressions: `loess`, `locploy`, `locfit`, `ksmooth` etc. However, you should be careful that not all of them uses the same definition of the bandwidth as we did.

In our formulation, the bandwidth is simply a raw value that you directly plug into the density function. This is the implementation in the `locpoly()` function. For this formulation, you can use some default bandwidth value, such as the **Silverman rule of thumb** for univariate regression, defined as

$$h = 1.06\,\sigma_x\,n^{-1/5}$$

where σ_x is the standard deviation that can be estimated from the data. For multi-dimensional kernels, h is generally in the order of $n^{-1/(d+4)}$. However, you should always be aware that these formulas are based on many assumptions that can be violated in practice. Hence, cross-validation for tuning may still be needed.

On the other hand, some other functions implements the bandwidth in a different way. For example, the `ksmooth()` function from the base `R` package will first scale the original covariates in a certain way, and then apply the bandwidth. The popular `loess` function uses a `span` parameter to control the size of neighborhood. In this case, the bandwidth is decided by first finding the closes k neighbors and then use a tri-cubic weighting on the range of these neighbors. Hence, the bandwidth essentially varies depending on the target point. In fact, the `kknn()` also utilize such a feature as default, so when you want to fit the standard KNN, you should specify `method = "rectangular"`, otherwise, the neighboring points will not receive the same weight. The `locfit` package has various ways to specify the bandwidth. Hence, when you use these functions, be careful about this and always read the documentations first.