Miscelaneous Topics

Stats 102A

Miles Chen

Department of Statistics

Week 10 Monday

UCLA

Section 1

Function `optim()`

Back to Optimization

After a small detour covering We revisit the topic of optimization.

R's function optim()

R has a general-purpose optimization function. It can be used to find the minimum of a function.

From the help file, this is the structure of a call to optim()

```
optim(par, fn, gr = NULL, ...,
      method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN",
                 "Brent"),
      lower = -Inf, upper = Inf,
      control = list(), hessian = FALSE)
```

R's function optim()

- par is a vector of initial parameter values to be optimized over
- fn is the function to be minimized. The function's first argument needs to be a vector of parameters over which minimization takes place. The function must return a single scalar value.
- gr allows you to provide another function that will return the gradient. Some of the methods (BFGS, CG, L-BFGS-B) use gradients in the optimization process.
  - analogy: Newton-Raphson method uses the derivative of the function to find the next value to use. For multidimensional functions, the gradient is conceptually similar to derivative and these gradient-based methods are somewhat similar in concept to Newton-Raphson.
  - If the gr function is not provided, optim() will find a finite-difference approximation (similar to the derivative approximation from Week 7 supplement).
- method allows you to specify the optimization method
- control allows you to provide some additional parameters that control some of the algorithm's behaviors.

The two arguments that you must provide are par and fn.

Example Usage

Let's say we wish to optimize (find the minimum) of the following function.

$$g(x, y) = x^2 - 2x - .5xy + 2.5y^2$$

We write the function in R. We may traditionally write the function like this:

```
g <- function(x, y) {
    x ^ 2 - 2 * x - .5 * x * y + 2.5 * y ^ 2
}
```

For our function to work with `optim()`, all of the parameter/variables values must be in a single vector argument. It is very easy to fix. I gather `x` and `y` into `par` and inside the function I assign the values in `par` to the different variable names. The last line in the function is the scalar value that is returned.

```
g <- function(par) {
    x <- par[1]
    y <- par[2]
    x ^ 2 - 2 * x - .5 * x * y + 2.5 * y ^ 2
}
```

Example Usage

To use the function, we choose some arbitrary starting values as our `par` (-1, 1.5). After the function runs, the values of `par` that minimize the function are in `par` and the function's value itself is in `value`. The true optimal values are $x = 40/39 \approx 1.025641$ and $y = 4/39 \approx 0.1025641$.

```
optim(par = c(-1, 1.5), fn = g)
```

```
## $par
## [1] 1.0255565 0.1024714
##
## $value
## [1] -1.025641
##
## $counts
## function gradient
##       85       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Example Usage

You can use other optimization methods to see how those values compare.

The true optimal values are $x = 40/39 \approx 1.025641$ and $y = 4/39 \approx 0.1025641$.

```
optim(par = c(-1, 1.5), fn = g, method = "BFGS")
```

```
## $par
## [1] 1.0256410 0.1025641
##
## $value
## [1] -1.025641
##
## $counts
## function gradient
##       17        5
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Another example

This example is unnecessary because we have the function `lm()`. But let's say you wish to fit the least squares regression line for

```
x <- c(1, 2, 3, 4)
y <- c(2, 6, 4, 8)
```

This can be done easily using `lm()`, but I wish to illustrate how we can use `optim()` to solve the problem as well.

```
lm(y ~ x)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Coefficients:
## (Intercept)            x
##         1.0          1.6
```

The function to optimize

First, we write the function we wish to optimize as a function of the parameters: the intercept and slope.

The cost we wish to optimize is the sum of squares residuals. We want the intercept and slope that results in the smallest sum of squares

```
ss <- function(par) {
  b0 <- par[1] # intercept
  b1 <- par[2] # slope
  x <- c(1, 2, 3, 4) # the data
  y <- c(2, 6, 4, 8)
  yhat <- b0 + b1 * x # predicted values
  residuals <- y - yhat
  sum(residuals^2) # the function returns the sum of squared residuals
}
```

Example Usage

`optim()` will search for the values of `par` that will minimize the function `ss`. From `lm()`, we know the true optimal values are $b_0 = 1$ and $b_1 = 1.6$ and `optim()` manages to find them.

```
optim(par = c(0, 0), fn = ss, method = "BFGS")
```

```
## $par
## [1] 1.0 1.6
##
## $value
## [1] 7.2
##
## $counts
## function gradient
##       15        5
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Section 2

Kernel Density Estimation

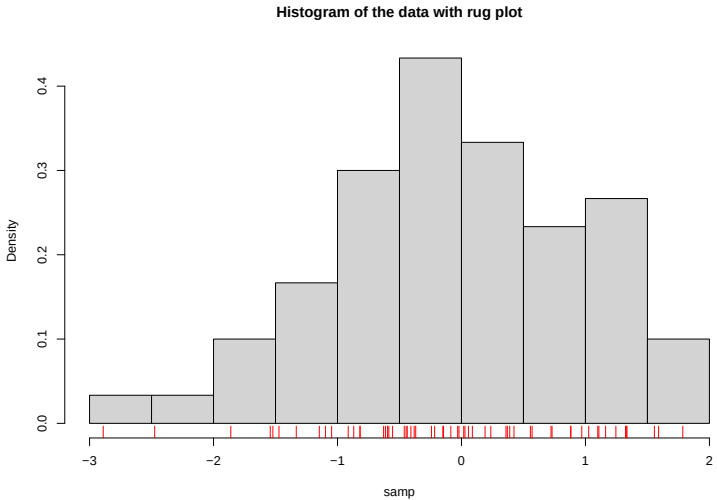
Kernel Density Estimation

Kernel Density Estimation (KDE) allows us to make smooth line approximations for density curves based on a sample of data.

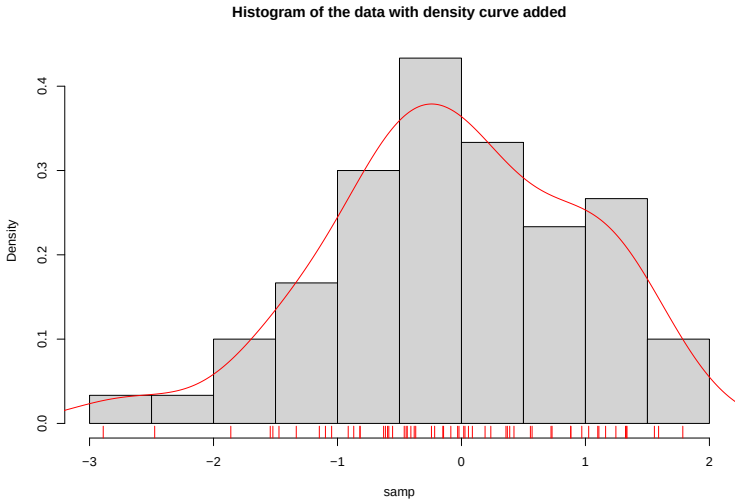
To illustrate, I'll first create a simple sample.

```
set.seed(20)
samp <- rnorm(60)
```

```r
hist(samp, freq = FALSE, main = "Histogram of the data with rug plot")
rug(samp, col = "red")
```

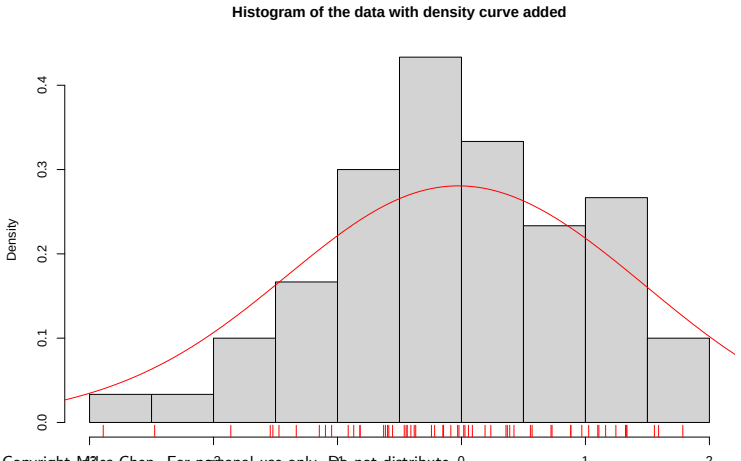


```r
hist(samp, freq = FALSE, main = "Histogram of the data with density curve added")
rug(samp, col = "red")
lines(density(samp), col = "red")
```



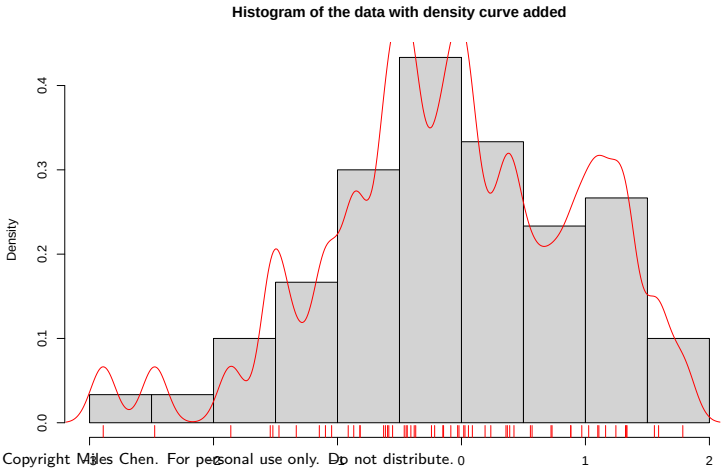
You can change the appearance of the density curve by adjusting the bandwidth with argument `bw=`. Here I set the bandwidth to 1, which for this data smooths the curve even further.

```
hist(samp, freq = FALSE, main = "Histogram of the data with density curve added")
rug(samp, col = "red")
lines(density(samp, bw = 1), col = "red")
```



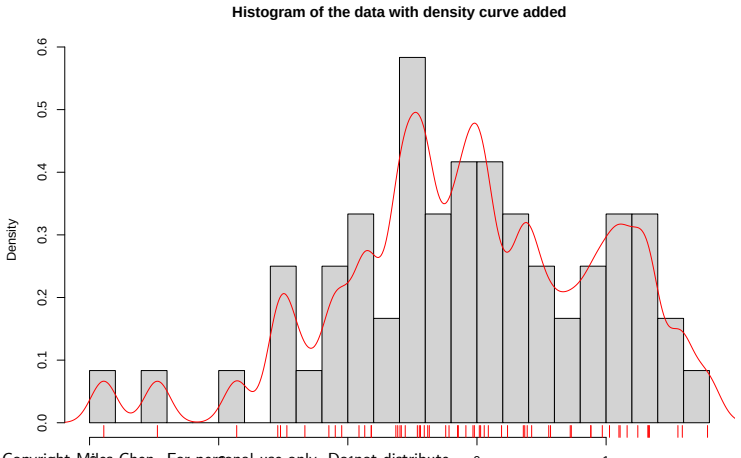
A thinner bandwidth results in more jagged peaks. This is akin to changing the bin width of the histogram.

```
hist(samp, freq = FALSE, main = "Histogram of the data with density curve added")
rug(samp, col = "red")
lines(density(samp, bw = 0.1), col = "red")
```



A thinner bandwidth results in more jagged peaks. This is akin to changing the bin width of the histogram.

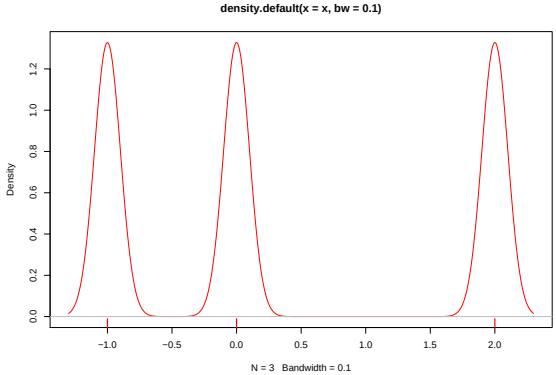
```
hist(samp, freq = FALSE, breaks = 30, main = "Histogram of the data with density curve added")
rug(samp, col = "red")
lines(density(samp, bw = 0.1), col = "red")
```



How it works

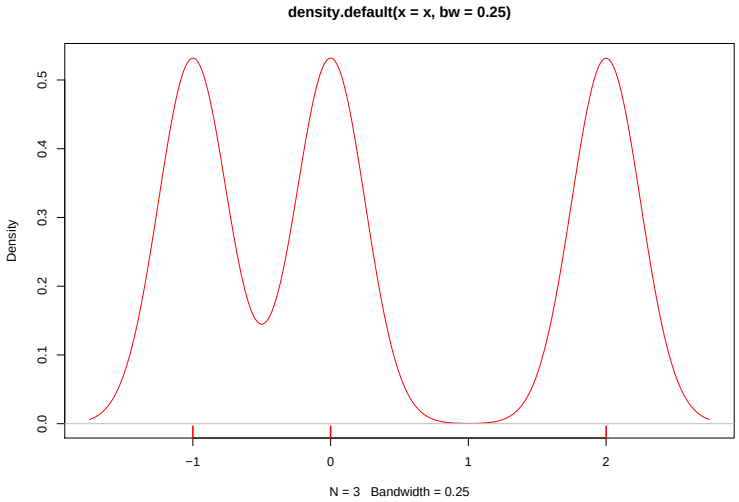
KDE works by "stacking" small curves at each data point. I'll illustrate with a tiny data set consisting of three data values and a very small bandwidth.

```
x <- c(-1, 0, 2)
plot(density(x, bw = 0.1), col = "red")
rug(x, col = "red", lwd = 2)
```

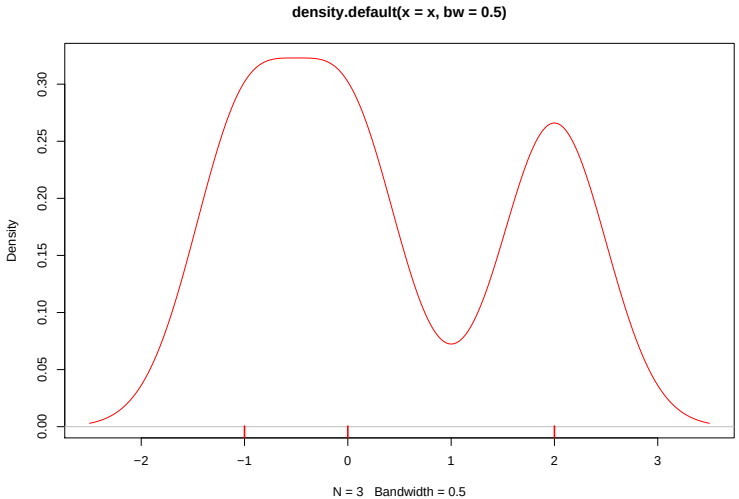


As I adjust the bandwidth, we can see how the overlapping areas stack on to each other.

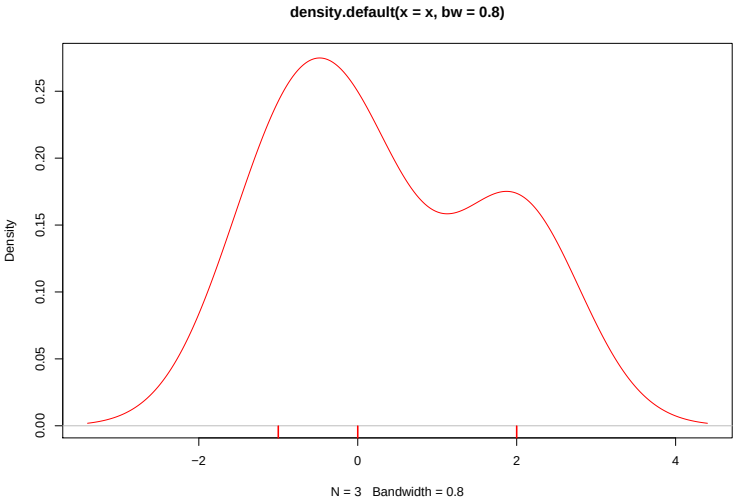
```
plot(density(x, bw = 0.25), col = "red")
rug(x, col = "red", lwd = 2)
```



```r
plot(density(x, bw = 0.5), col = "red")
rug(x, col = "red", lwd = 2)
```



```r
plot(density(x, bw = 0.8), col = "red")
rug(x, col = "red", lwd = 2)
```



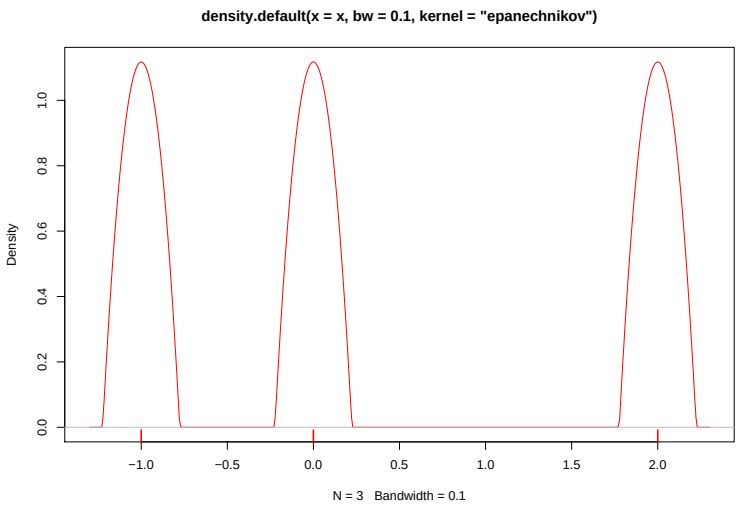
Kernels

The default "kernel" is a Gaussian curve.

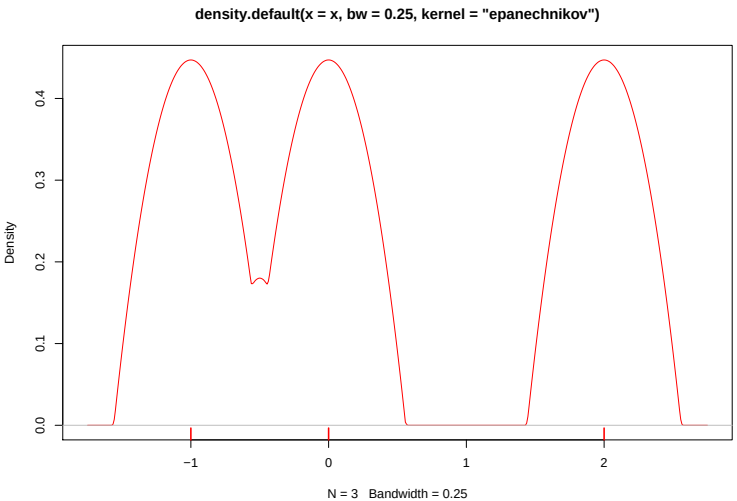
The `density` function in R allows for different types of Kernels to be used.

- `epanechnikov` (upside down parabola)
- `rectangular` (rectangular, but because of the default plot settings, it may appear trapezoidal)
- `triangular`
- `cosine`

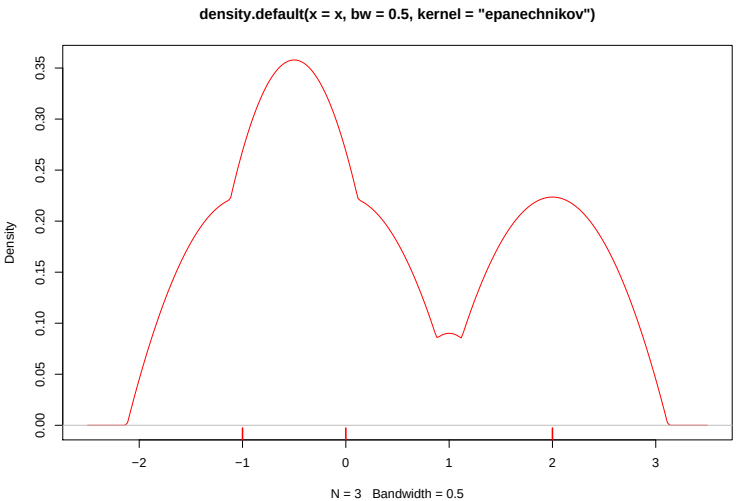
```r
plot(density(x, bw = 0.1, kernel = "epanechnikov"), col = "red")
rug(x, col = "red", lwd = 2)
```



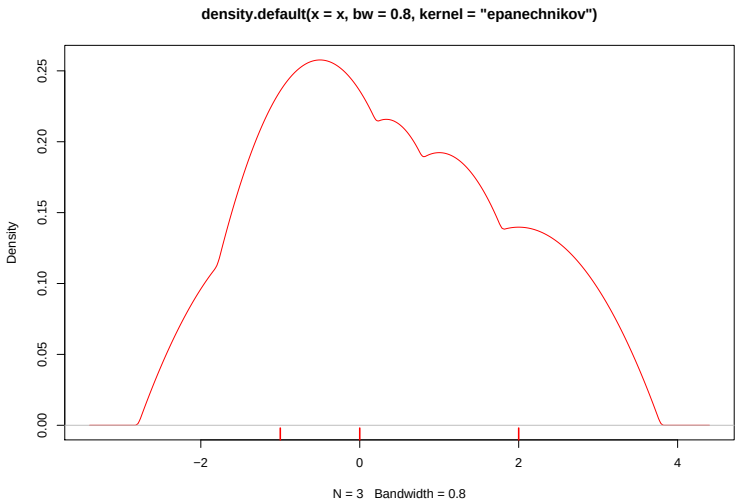
```r
plot(density(x, bw = 0.25, kernel = "epanechnikov"), col = "red")
rug(x, col = "red", lwd = 2)
```



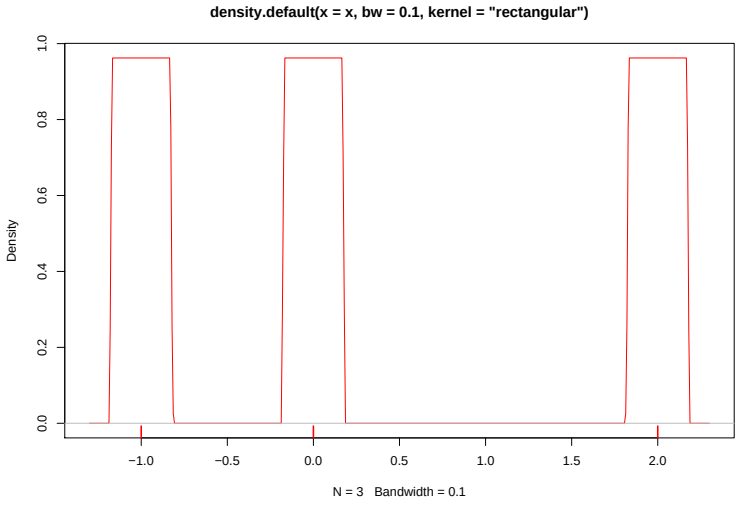
```r
plot(density(x, bw = 0.5, kernel = "epanechnikov"), col = "red")
rug(x, col = "red", lwd = 2)
```



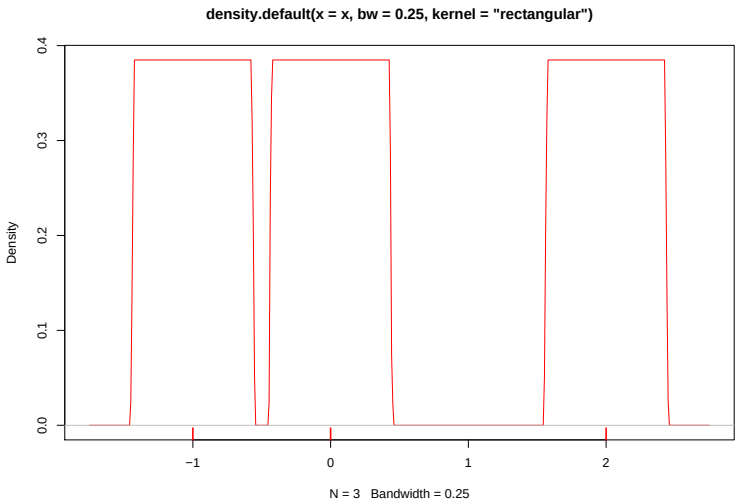
```r
plot(density(x, bw = 0.8, kernel = "epanechnikov"), col = "red")
rug(x, col = "red", lwd = 2)
```



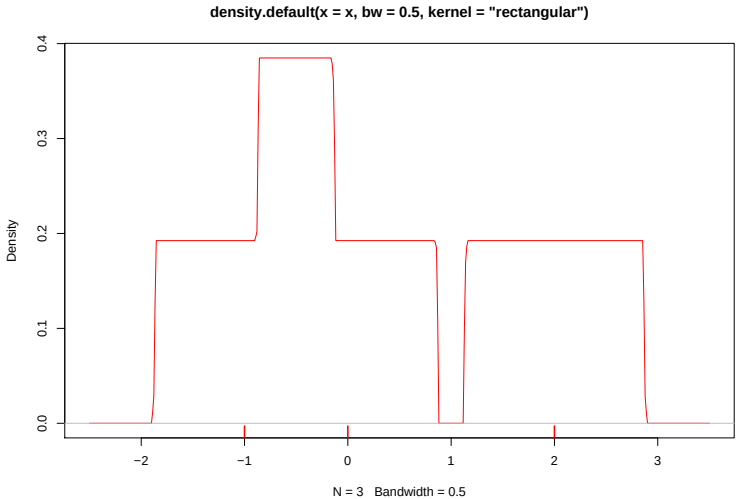
```r
plot(density(x, bw = 0.1, kernel = "rectangular"), col = "red")
rug(x, col = "red", lwd = 2)
```



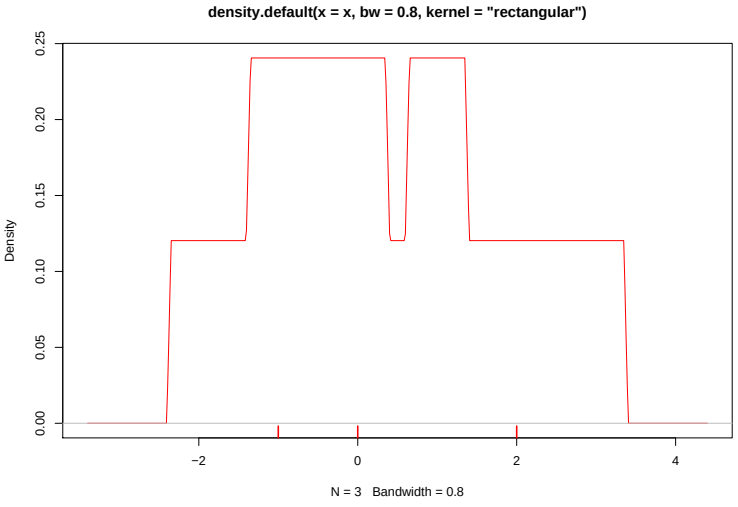
```r
plot(density(x, bw = 0.25, kernel = "rectangular"), col = "red")
rug(x, col = "red", lwd = 2)
```



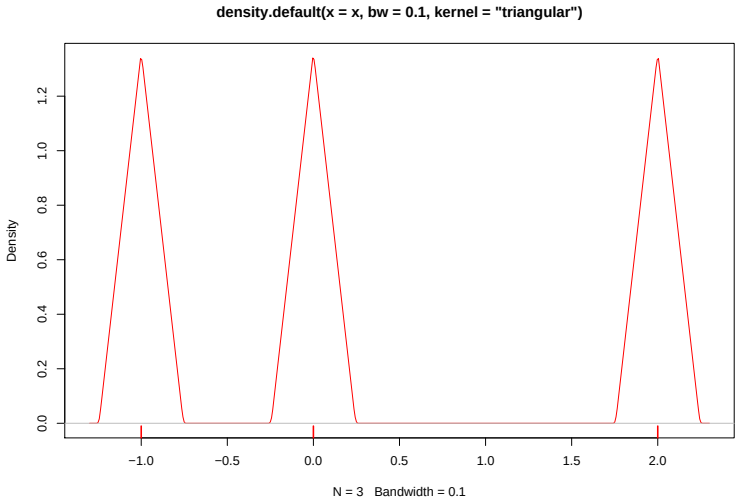
```r
plot(density(x, bw = 0.5, kernel = "rectangular"), col = "red")
rug(x, col = "red", lwd = 2)
```



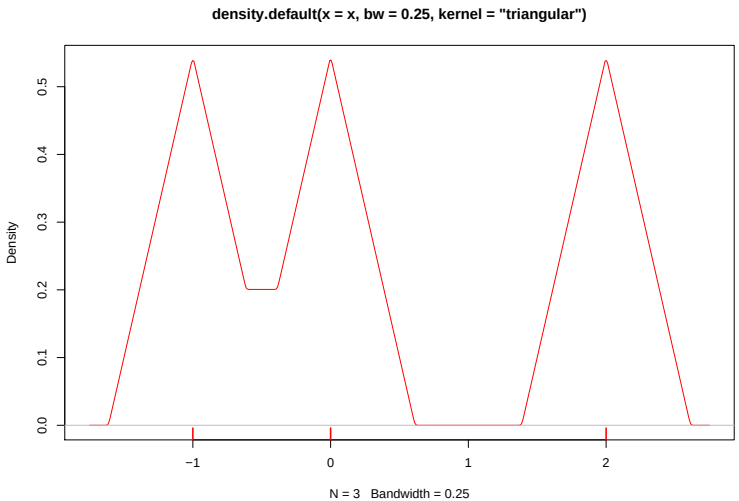
```r
plot(density(x, bw = 0.8, kernel = "rectangular"), col = "red")
rug(x, col = "red", lwd = 2)
```



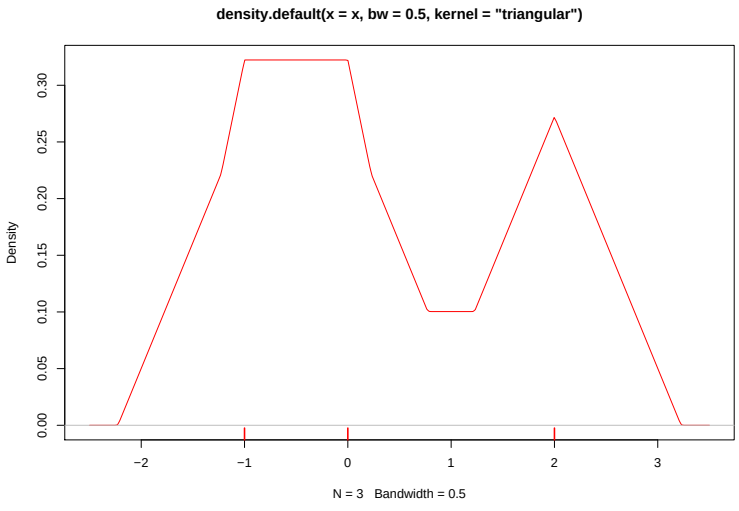
```r
plot(density(x, bw = 0.1, kernel = "triangular"), col = "red")
rug(x, col = "red", lwd = 2)
```



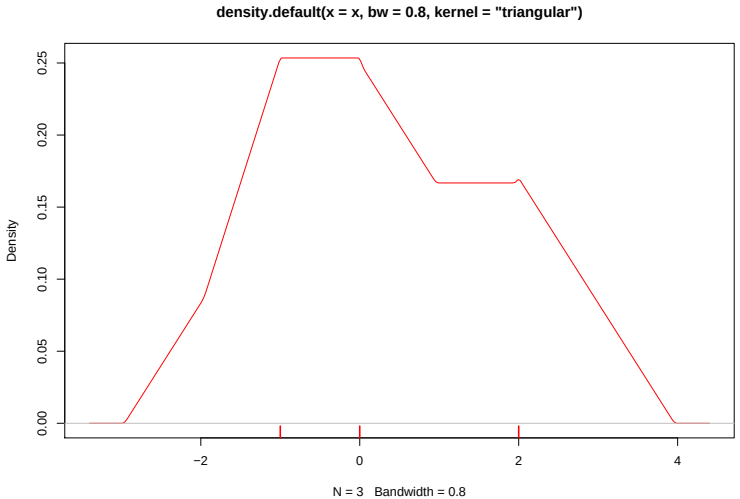
```r
plot(density(x, bw = 0.25, kernel = "triangular"), col = "red")
rug(x, col = "red", lwd = 2)
```



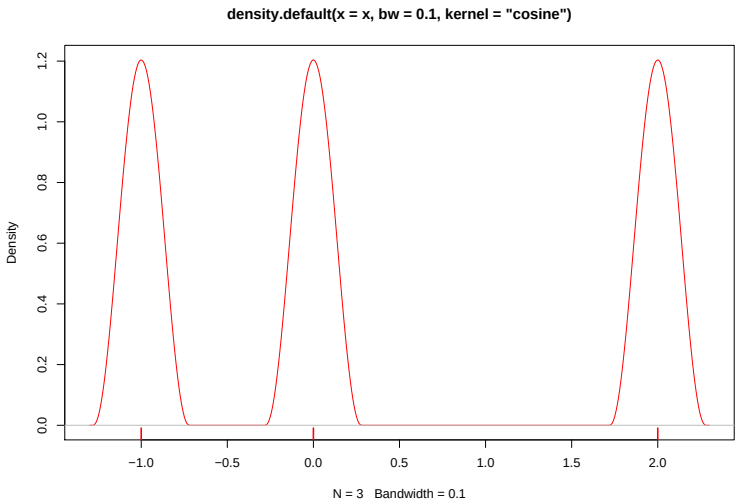
```r
plot(density(x, bw = 0.5, kernel = "triangular"), col = "red")
rug(x, col = "red", lwd = 2)
```



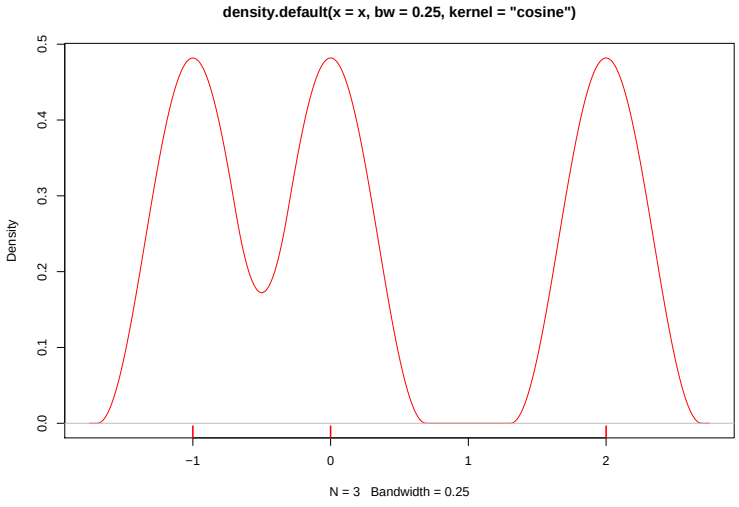
```r
plot(density(x, bw = 0.8, kernel = "triangular"), col = "red")
rug(x, col = "red", lwd = 2)
```



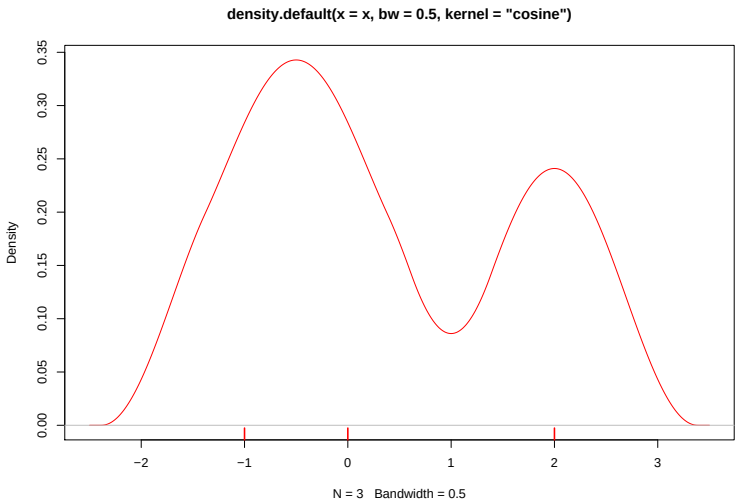
```r
plot(density(x, bw = 0.1, kernel = "cosine"), col = "red")
rug(x, col = "red", lwd = 2)
```



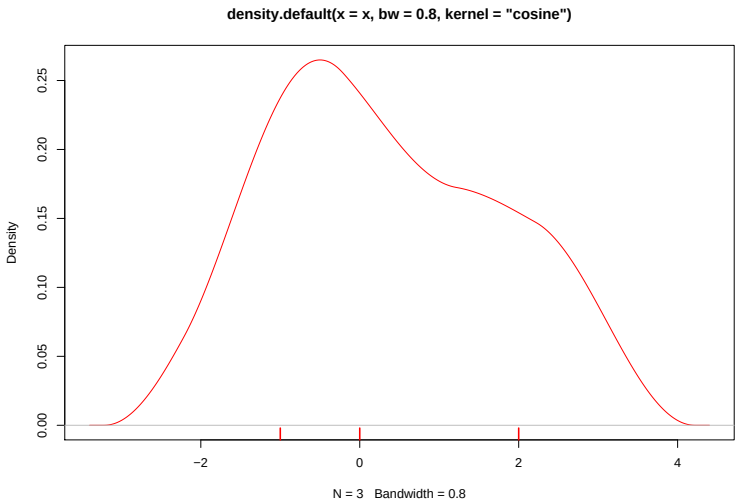
```r
plot(density(x, bw = 0.25, kernel = "cosine"), col = "red")
rug(x, col = "red", lwd = 2)
```



```r
plot(density(x, bw = 0.5, kernel = "cosine"), col = "red")
rug(x, col = "red", lwd = 2)
```



```r
plot(density(x, bw = 0.8, kernel = "cosine"), col = "red")
rug(x, col = "red", lwd = 2)
```



Default bandwidth

The default settings of density will automatically calculate the bandwidth using a formula:

- Select a measure of spread: either sd or IQR / 1.34, whichever is smaller.
- Then the `bw = 0.9 * selected_spread * n^(-0.2)`

Other bandwidth choices are available or you can manually select one.

Warnings

Keep in mind that the Density estimate can produce "tails" in regions where no data exists (such as negative values for a distribution that is strictly positive). This is an artifact of the KDE process.