Projektaufgaben Block 2

Carlo Michaelis, 573479; David Hinrichs, 572347; Lukas Ruff, 572521

06 Dezember 2016

1 Nichtparametrisches Testen

1.1 Zwillingsstudie

Um zu testen, ob der Kindergartenbesuch einen signifikanten Einfluss auf die sozialen Fähigkeiten eines Kindes hat, führen wir einen zweiseitigen t-Test und einen zweiseitigen Wilcoxon-Vorzeichen-Test, jeweils zum Signifikanzniveau $\alpha = 0.05$, durch. Die Entscheidung fiel dabei auf einen zweiseitigen Test, da zuvor keine begründete Vermutung über einseitige Effekte angenommen wurde.

```
# Enter data
x <- c(82,69,73,43,58,56,76,65)
y <- c(63,42,74,37,51,43,80,62)

# Two-sided t-test
t.test(x, y, alternative = "two.sided", mu = 0, conf.level = 0.95, paired = TRUE)
```

```
##
##  Paired t-test
##
## data:  x and y
## t = 2.3791, df = 7, p-value = 0.04895
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   0.05320077 17.44679923
## sample estimates:
## mean of the differences
##                    8.75
```

```
# t.test(x-y, alternative = "two.sided", mu = 0, conf.level = 0.95)  # alternative

# Two-sided Wilcoxon signed rank test
wilcox.test(x, y, alternative = "two.sided", mu = 0, conf.level = 0.95,
            paired = TRUE, conf.int = TRUE)
```

```
##
##  Wilcoxon signed rank test
##
## data:  x and y
## V = 32, p-value = 0.05469
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  -0.5 19.0
## sample estimates:
## (pseudo)median
##           7.75
```

```r
# wilcox.test(x-y, alternative = "two.sided", mu = 0, conf.level = 0.95,
#             conf.int = TRUE)  # alternative
```

Wir können sehen, dass der t-Test die Nullhypothese ablehnt ($p = 0.04895 < 0.05 = \alpha$) und somit einen signifikanten Einfluss feststellt. Der Wilcoxon-Vorzeichen-Test verwirft die Nullhypothese dagegen nicht ($p = 0.05469 > 0.05 = \alpha$). Durch die Normalverteilungsannahme $X_i - Y_i \sim N(0, \sigma^2)$, die für gegebenes Sample in Frage gestellt werden kann, besitzt der t-Test eine größere Power. Der nichtparametrische Wilcoxon-Vorzeichen-Test benötigt hingegen keine Verteilungsannahme, besitzt jedoch eine kleinere Power.

Wenn statt dem zweiseitigen ein einseitiger Test durchgeführt wird, mit der Annahme, dass die sozialen Fähigkeiten durch den Kindergartenbesuch signifikant höher sind, erhalten wir folgende Ergebnisse:

```r
# Two-sided t-test
t.test(x, y, alternative = "greater", mu = 0, conf.level = 0.95, paired = TRUE)
```

```
##
##  Paired t-test
##
## data:  x and y
## t = 2.3791, df = 7, p-value = 0.02447
## alternative hypothesis: true difference in means is greater than 0
## 95 percent confidence interval:
##  1.781971      Inf
## sample estimates:
## mean of the differences
##                    8.75
```

```r
# t.test(x-y, alternative = "greater", mu = 0, conf.level = 0.95)  # alternative

# Two-sided Wilcoxon signed rank test
wilcox.test(x, y, alternative = "greater", mu = 0, conf.level = 0.95,
            paired = TRUE, conf.int = TRUE)
```

```
##
##  Wilcoxon signed rank test
##
## data:  x and y
## V = 32, p-value = 0.02734
## alternative hypothesis: true location shift is greater than 0
## 95 percent confidence interval:
##    1 Inf
## sample estimates:
## (pseudo)median
##           7.75
```

```r
# wilcox.test(x-y, alternative = "greater", mu = 0, conf.level = 0.95,
#             conf.int = TRUE)  # alternative
```

Die p-Werte werden nur an einer Seite bestimmt und entsprechen daher nun exakt der Hälfte der p-Werte des zweiseitigen Tests. Beide Tests indizieren nun einen signifikanten Anstieg der sozialen Fähigkeiten durch den Kindergartenbesuch. Der einseitige t-Test hat eine stärkere Annahme, da eine Änderung des Wertes von vorne herein nur in eine Richtung angenommen wird. Es kommt daher bei gleichem Niveau α schneller zu sinifikanten Ergebnissen, als bei einem zweiseitigen t-Test.

1.2 t-Test vs. Wilcoxon-Vorzeichen-Test

```
fnTestPowerMC <- function(fnError, n = 30, alpha = 0.05, nSim = 10^4, ...) {
  # This function estimates the probability of rejecting the null hypothesis of a
  # t-test and a Wilcoxon signed rank test using Monte Carlo simulations of iid
  # random variables X_i = \theta + \epsilon_i.
  #
  # Args:
  #   fnError:  Function which generates random samples from a symmetric error
  #             distribution \epsilon
  #   n:        Number of random samples used in tests
  #   alpha:    Significance level used in tests
  #   nSim:     Number of MC simulations of size n
  #   ...:      Further arguments to be passed to fnError
  #
  # Returns:
  #   A list containing the following elements:
  #     $TProb:       MC estimation of rejection probability for the t-test
  #     $WilcoxProb:  MC estimation of rejection probability for the Wilcoxon
  #                   signed rank test

  # Perform MC simulation
  matX <- matrix(fnError(n*nSim, ...), ncol = n)

  # Define sub-functions which only return p-values from the two tests
  fnPvalT <- function(x) {
    return(t.test(x)$p.value)
  }
  fnPvalWilcox <- function(x) {
    return(wilcox.test(x)$p.value)
  }

  # Perform nSim number of tests with sample size n for each of the two tests
  vecPvalT <- apply(matX, 1, fnPvalT)
  vecPvalWilcox <- apply(matX, 1, fnPvalWilcox)

  # Compute and return estimations of rejection probabilities
  result <- list()
  result$TProb <- mean(vecPvalT < alpha)
  result$WilcoxProb <- mean(vecPvalWilcox < alpha)
  return(result)
}
```

```
# Set seed
set.seed(42)

# Normal errors
fnTestPowerMC(fnError = rnorm, nSim = 10^5, mean = 0)
```

```
## $TProb
## [1] 0.04999
##
```

```
## $WilcoxProb
## [1] 0.04981
```

```
# Cauchy errors (t-distribution with df = 1)
fnTestPowerMC(fnError = rt, nSim = 10^5, df = 1)
```

```
## $TProb
## [1] 0.02034
##
## $WilcoxProb
## [1] 0.04947
```

```
# Uniform errors
fnTestPowerMC(fnError = runif, nSim = 10^5, min = -1, max = 1)
```

```
## $TProb
## [1] 0.05177
##
## $WilcoxProb
## [1] 0.05073
```

2 Dichteschätzung

2.1 Kerndichteschätzer

```
fnKernelDensityEst <- function(x, X, K, h) {
  # This function computes the kernel density estimates for a given sample X and
  # kernel K with bandwidth h at points x.
  #
  # Args:
  #   x: Points for which the kernel density estimates are computed
  #   X: Data sample on which the kernel density estimation is fitted
  #   K: Kernel function to be used for smoothing
  #   h: Smoothing bandwidth
  #
  # Returns:
  #   A vector of the kernel density estimates at points x

  # Compute kernel density estimation values using outer
  n <- length(X)
  mKernels <- K((1/h) * outer(X, x, FUN = "-"))
  return((1/(n*h)) * colSums(mKernels))
}
```

```
fnKernelPlot <- function(x, X, K, vh = NULL, vm = NULL,
                         fnEstimation = fnKernelDensityEst,
                         title = NULL, trueDensity = NULL, ...) {
  # This function plots the kernel density estimates for different bandwidths.
  # If trueDensity is specified, the true density will be plotted as well.
```

```r
#
# Args:
#   x:            Points for which the kernel density estimates are computed
#   X:            Data sample on which the kernel density estimation is fitted
#   K:            Kernel function to be used for smoothing
#   vh:           Vector of smoothing bandwidths
#   vm:           Vector of nearest neigbor positions
#   fnEstimation: Type of Estimation used
#   title:        Main title of the plot
#   trueDensity:  True density to be plotted (e.g. dnorm)
#   ...:          Further arguments passed to or from other methods
#
# Returns:
#   -

if(!is.null(vh)) {
  vPar <- vh
  parString <- "h"
} else if(!is.null(vm)) {
  vPar <-  vm
  parString <- "m"
} else {
  stop("You need to set the bandwith vh OR the neighbor vm")
}

nh <- length(vPar)

# Create color palette for number of bandwidths
cols <- rainbow(nh, alpha = 1)

# Initialize plot with first kernel density
plot(x, fnEstimation(x, X, K, vPar[1]), type = "l", col = cols[1],
     main = title, xlab = "x", ylab = "Density", ...)

# Add kernel density for each additional bandwidth
if (nh > 1) {
  for (i in 2:nh) {
    lines(x, fnEstimation(x, X, K, vPar[i]), col = cols[i])
  }
}

# Create legend
legend <- sapply(vPar, function(par) {par <- paste(parString, " = ", round(par, 4))})

# Plot true density if specified
if (!is.null(trueDensity)) {
  lines(x, trueDensity(x), col = "black")
  legend <- c("True density", legend)
  cols <- c("black", cols)
}

# Plot legend
legend("topright", legend = legend, col = cols, lty = 1, cex = 0.75)
```

```
}
```

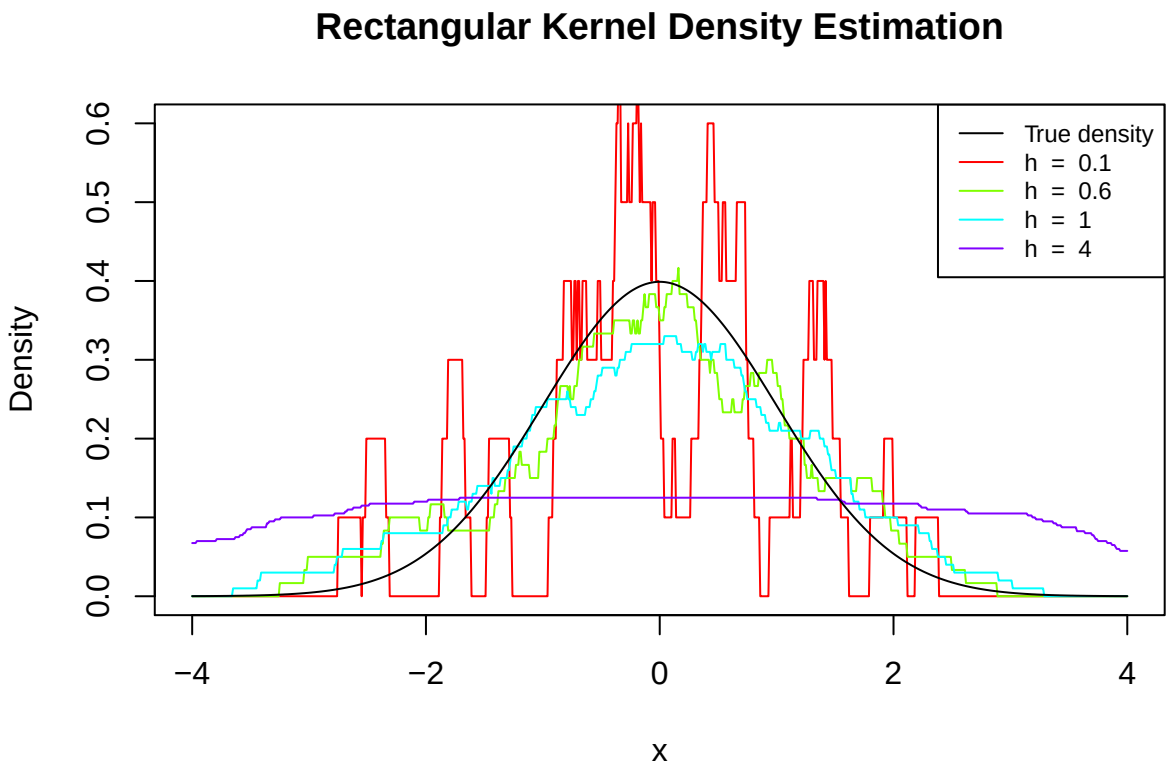
```
# Define different smoothing kernels

fnRectangularKernel <- function(x) {
  return(0.5 * (abs(x) <= 1))
}

fnGaussianKernel <- function(x) {
  return((1/sqrt(2*pi)) * exp((-0.5) * x^2))
}

fnEpanechnikovKernel <- function(x) {
  return(0.75 * (1 - x^2) * (abs(x) <= 1))
}
```

2.1.1 Anwendung auf standardnormalverteilte Zufallszahlen

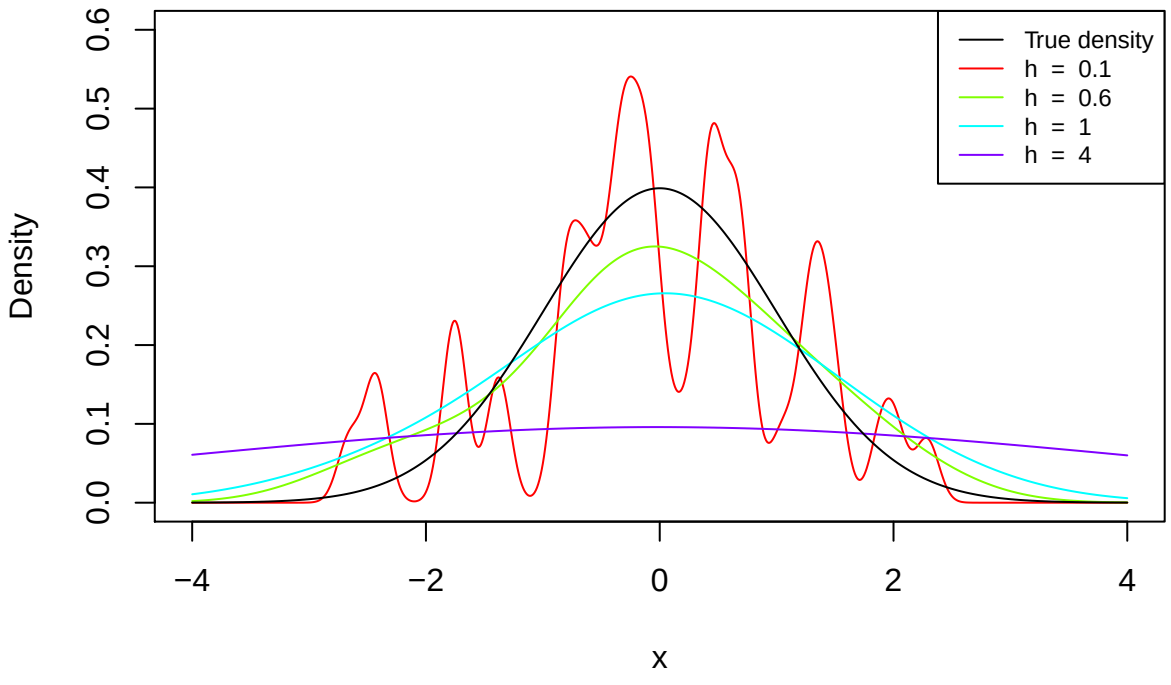
```
# Generate sample
set.seed(42)
n <- 50
X <- rnorm(n)

# Define points to estimate kernel density on
x <- seq(-4, 4, 0.01)

# Set different bandwidths
h <- c(0.1, bw.ucv(X), 1, 4)
```

```
## Warning in bw.ucv(X): minimum occurred at one end of the range
```

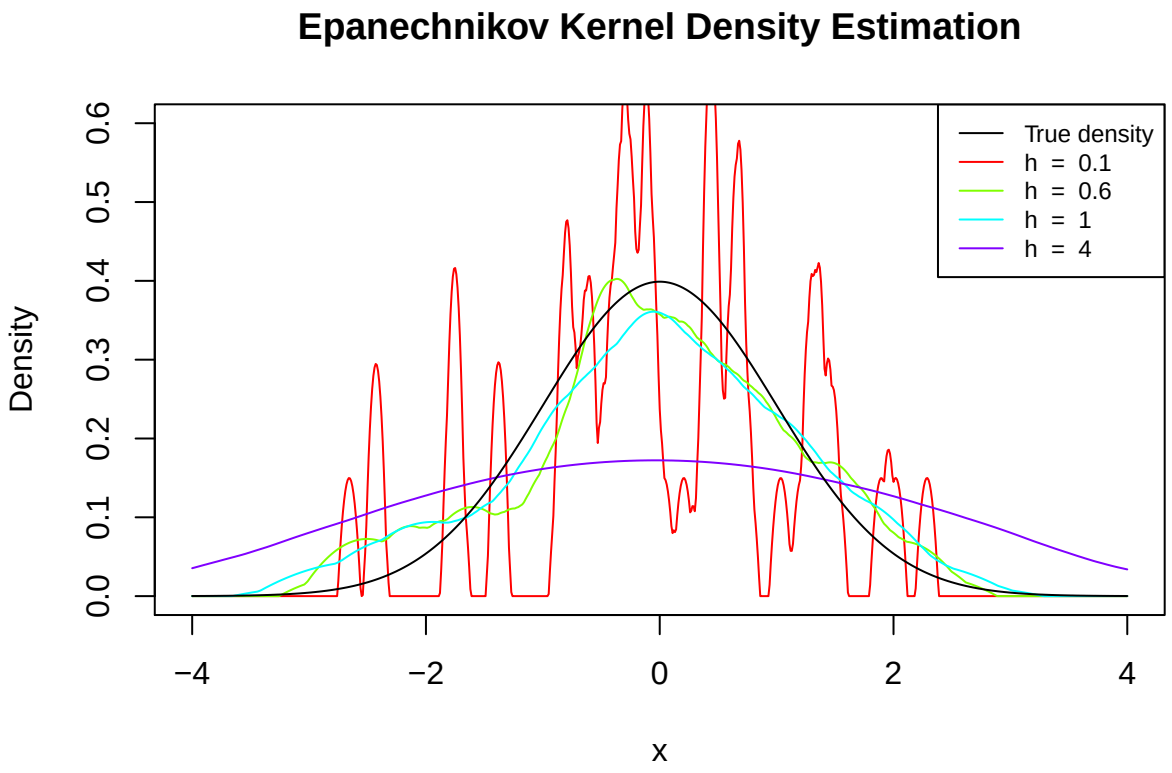
```
# Rectangular kernel density estimation
fnKernelPlot(x, X, fnRectangularKernel, h,
             title = "Rectangular Kernel Density Estimation",
             trueDensity = dnorm, ylim = c(0, 0.6))
```




```r
# Gaussian kernel density estimation
fnKernelPlot(x, X, fnGaussianKernel, h,
             title = "Gaussian Kernel Density Estimation",
             trueDensity = dnorm, ylim = c(0, 0.6))
```

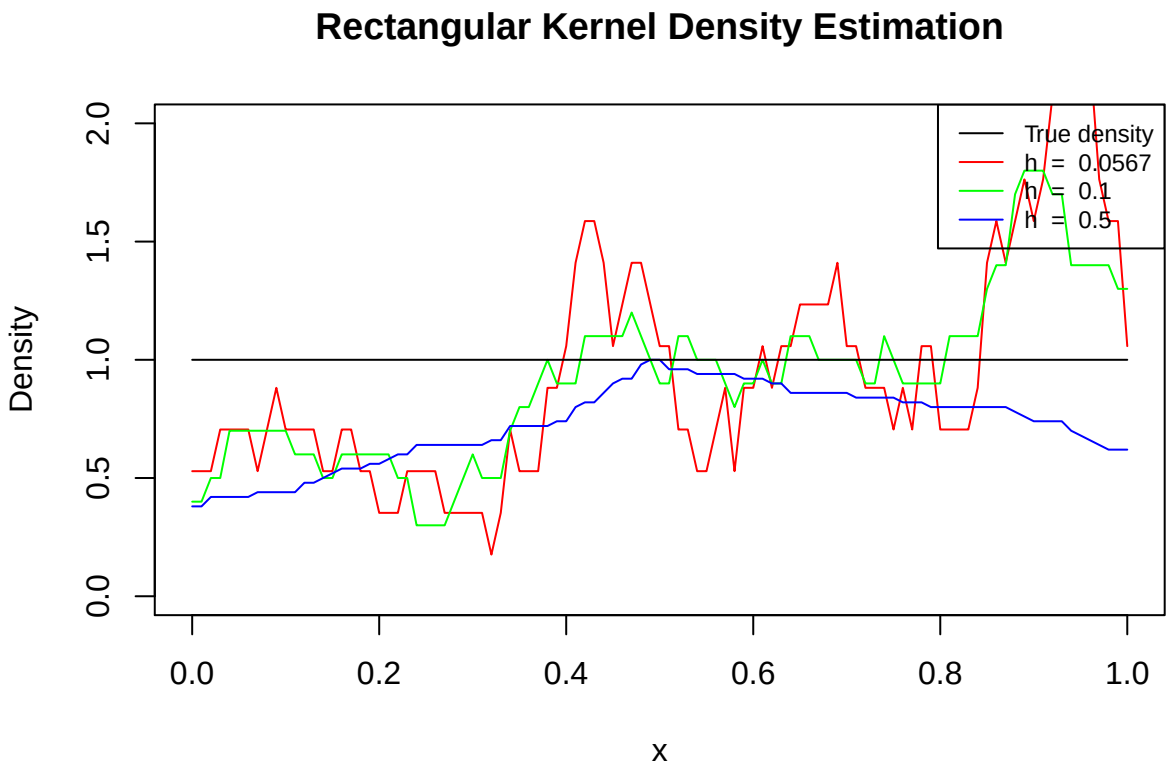


```r
# Epanechnikov kernel density estimation
fnKernelPlot(x, X, fnEpanechnikovKernel, h,
             title = "Epanechnikov Kernel Density Estimation",
             trueDensity = dnorm, ylim = c(0, 0.6))
```

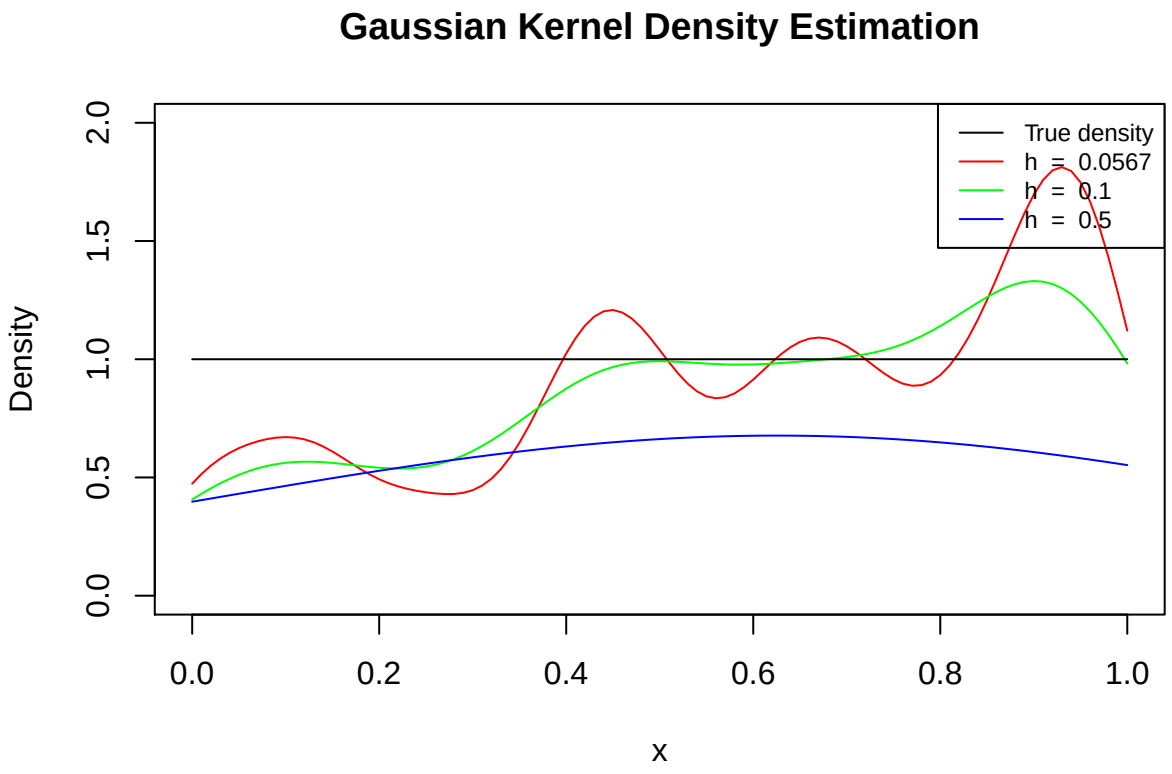


2.1.2 Anwendung auf gleichverteilte Zufallszahlen

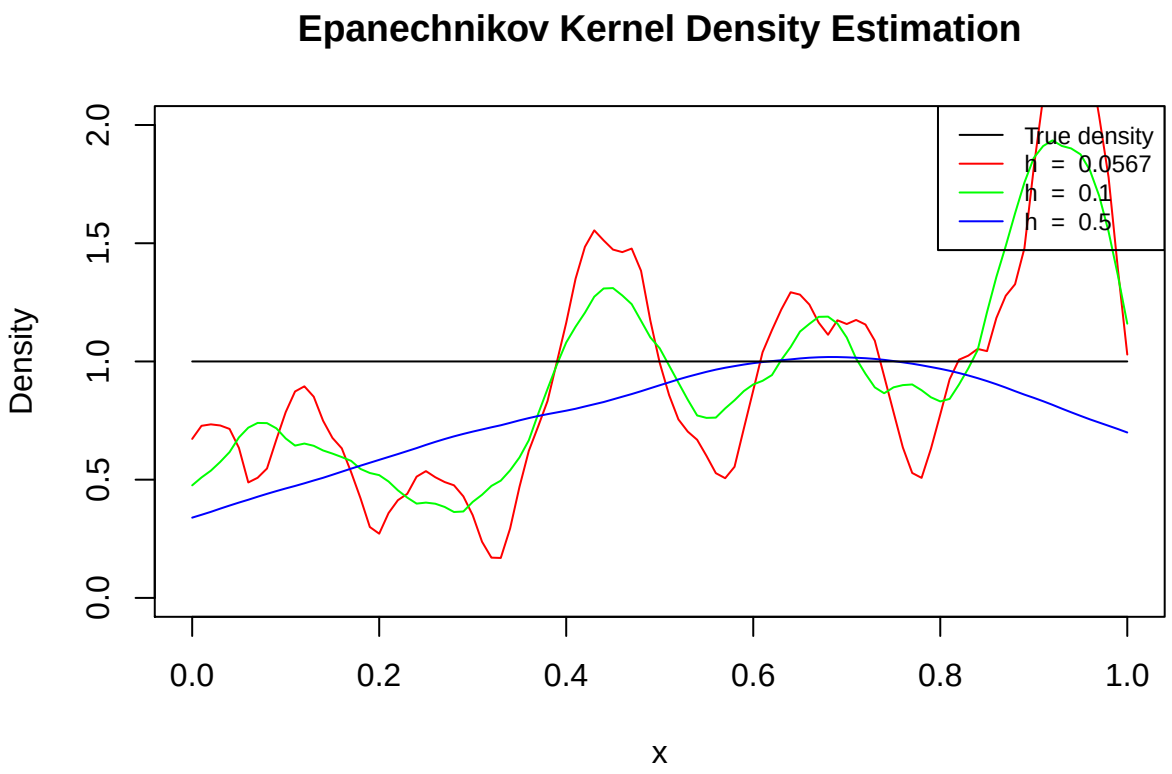
```r
# Generate sample
set.seed(42)
n <- 50
X <- runif(n)

# Define points to estimate kernel density on
x <- seq(0, 1, 0.01)

# Set different bandwidths
h <- c(bw.ucv(X), 0.1, 0.5)

# Rectangular kernel density estimation
fnKernelPlot(x, X, fnRectangularKernel, h,
             title = "Rectangular Kernel Density Estimation",
             trueDensity = dunif, ylim = c(0, 2))
```




```r
# Gaussian kernel density estimation
fnKernelPlot(x, X, fnGaussianKernel, h,
             title = "Gaussian Kernel Density Estimation",
             trueDensity = dunif, ylim = c(0, 2))
```



```r
# Epanechnikov kernel density estimation
fnKernelPlot(x, X, fnEpanechnikovKernel, h,
             title = "Epanechnikov Kernel Density Estimation",
             trueDensity = dunif, ylim = c(0, 2))
```

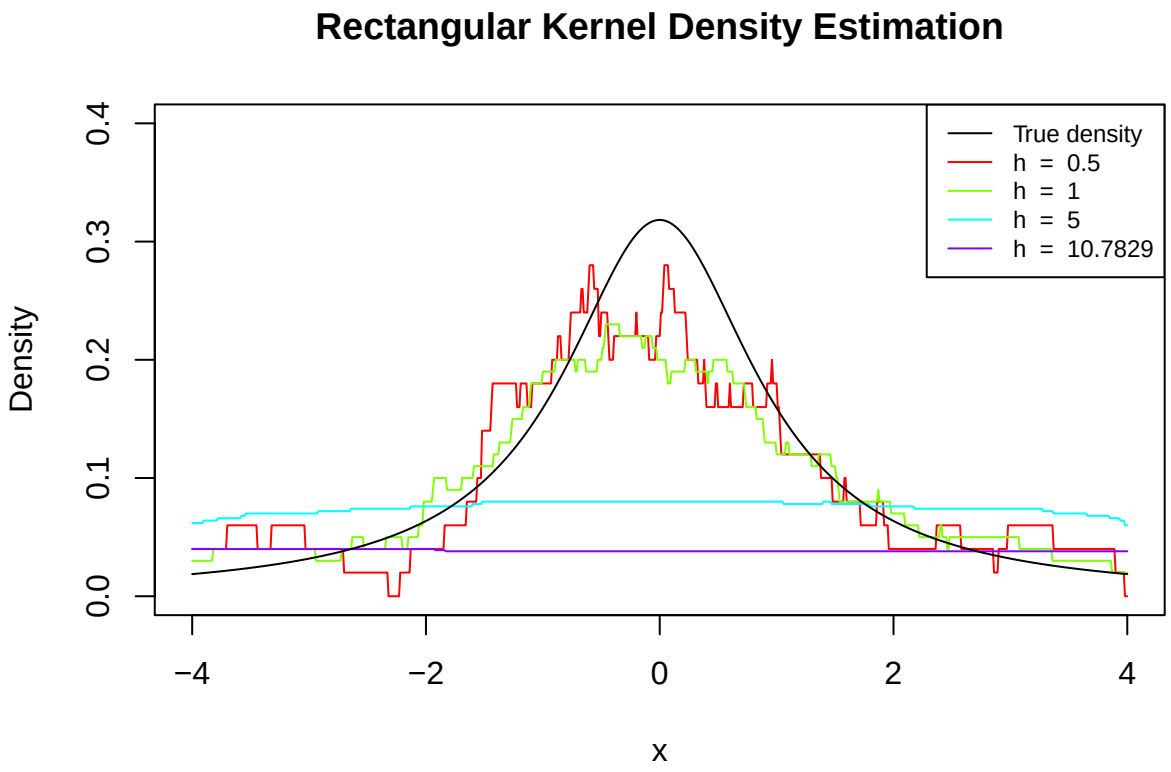


2.1.3 Anwendung auf Cauchy-verteilte Zufallszahlen

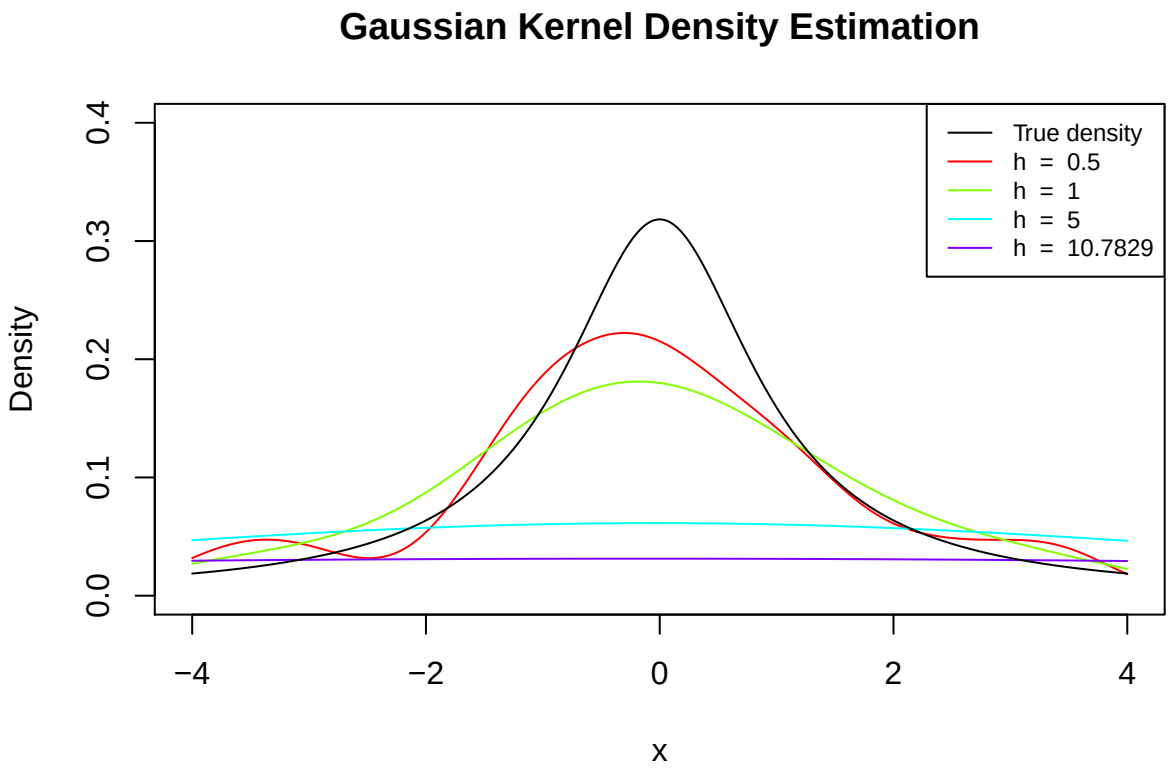
```r
# Generate sample
set.seed(42)
n <- 50
X <- rt(n, df = 1)

# Define points to estimate kernel density on
x <- seq(-4, 4, 0.01)

# Set different bandwidths
h <- c(0.5, 1, 5, bw.ucv(X))
```

```
## Warning in bw.ucv(X): minimum occurred at one end of the range
```

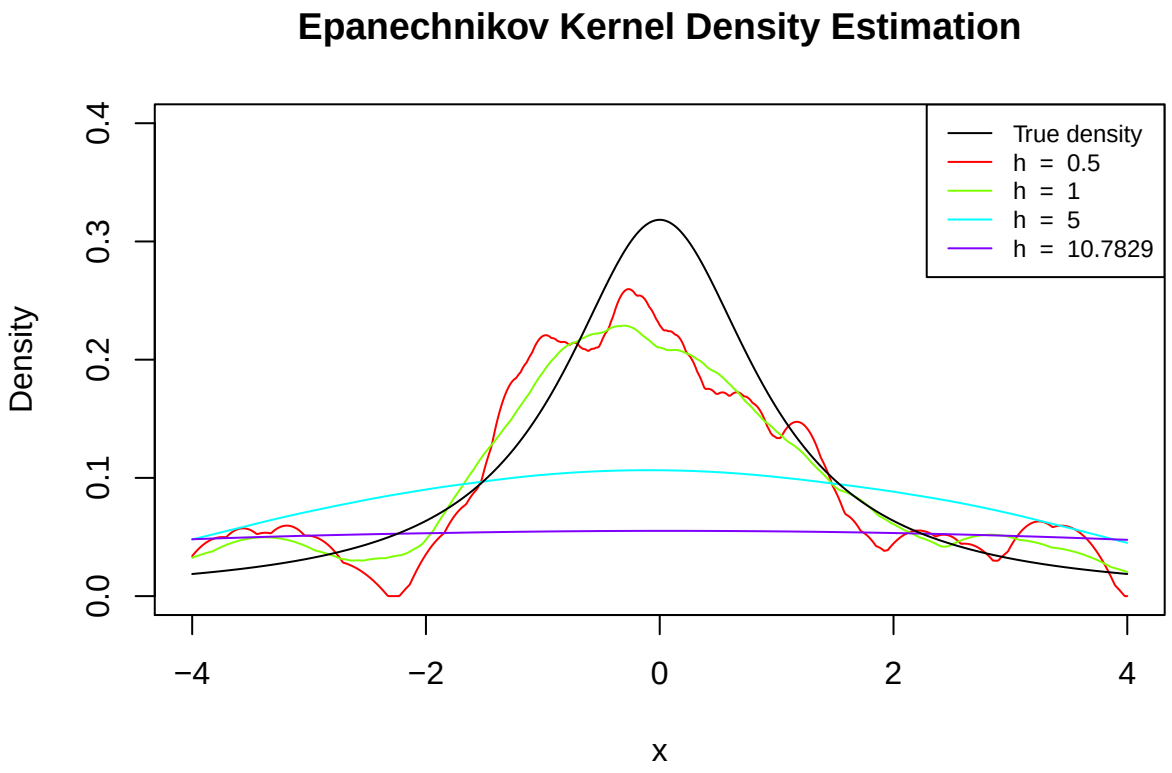
```r
# Define Cauchy density function
dcauchy <- function(x) {y <- dt(x, df = 1)}

# Rectangular kernel density estimation
fnKernelPlot(x, X, fnRectangularKernel, h,
             title = "Rectangular Kernel Density Estimation",
             trueDensity = dcauchy, ylim = c(0, 0.4))
```




```r
# Gaussian kernel density estimation
fnKernelPlot(x, X, fnGaussianKernel, h,
             title = "Gaussian Kernel Density Estimation",
             trueDensity = dcauchy, ylim = c(0, 0.4))
```

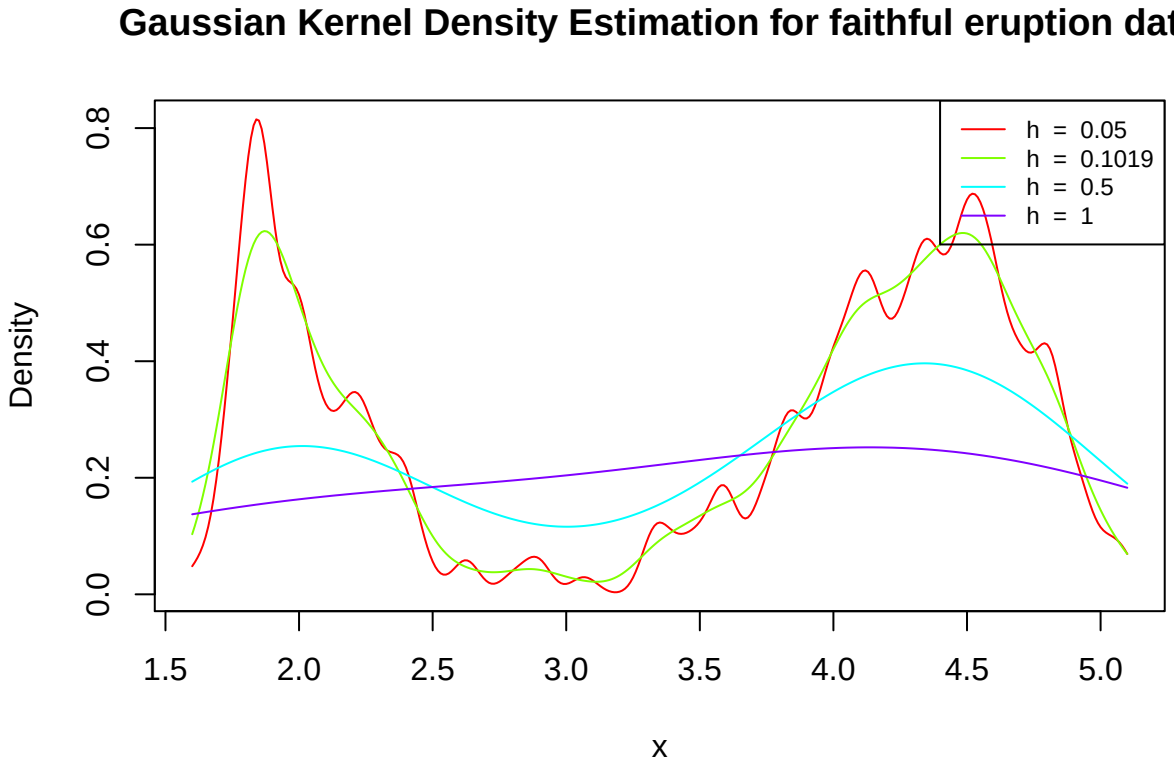


```r
# Epanechnikov kernel density estimation
fnKernelPlot(x, X, fnEpanechnikovKernel, h,
             title = "Epanechnikov Kernel Density Estimation",
             trueDensity = dcauchy, ylim = c(0, 0.4))
```



2.1.4 Anwendung auf faithful Datensatz

```r
# Faithful kernel density estimation
X <- faithful$eruptions

# Define points to estimate kernel density on
x <- seq(min(X), max(X), 0.01)

# Set different bandwidths
h <- c(0.05, bw.ucv(X), 0.5, 1)

# Gaussian kernel density estimation
title <- "Gaussian Kernel Density Estimation for faithful eruption data"
fnKernelPlot(x, X, fnGaussianKernel, h, title = title)
```



2.2 Kreuzvalidierung zur Bandweitenwahl

```
fnJ <- function(X, K, h) {
  # This function computes the value of the unbiased cross validation criteria.
  # The minimum of this function in h gives an estimated "optimal" (in the
  # sense of mean integrated squared error) bandwidth.
  #
  # Args:
  #   X: Data sample on which the kernel density estimation is fitted
  #   K: Kernel function to be used for smoothing
  #   h: Smoothing bandwidth
  #
  # Returns:
  #   Function value

  n <- length(X)

  # Compute integral of squared kernel density estimator
  integrand <- function(x) {return(fnKernelDensityEst(x, X, K, h)^2)}
  SKDEInt <- integrate(integrand, -Inf, Inf)$value

  # Compute G
  mKernels <- K((1/h) * outer(X, X, FUN = "-"))
  diag(mKernels) <- 0
  G <- (1/(n*(n-1)*h)) * sum(mKernels)

  # Return function value of J
```

```
  return(SKDEInt - 2*G)
}

fnUCV <- function(X, K, hmin, hmax, tol = 0.1 * hmin) {
  # This function returns the minimum of the unbiased cross validation criteria
  # in h which is "optimal" in the sense of mean integrated squared error.
  # We use R's optimize function to find the minimum.
  #
  # Args:
  #   X:    Data sample on which the kernel density estimation is fitted
  #   K:    Kernel function to be used for smoothing
  #   hmin: Lower bound for optimal h
  #   hmax: Upper bound for optimal h
  #   tol:  The desired accuracy
  #
  # Returns:
  #   Optimal bandwidth h*

  # Find and return the minimum using optimize
  fnTarget <- function(h) {return(fnJ(X, K, h))}
  hOpt <- optimize(fnTarget, c(hmin, hmax), tol = tol)
  return(hOpt$minimum)
}
```

```
hmin <- 0.01
hmax <- 10
fnUCV(faithful$eruptions, fnGaussianKernel, hmin, hmax)
```

```
## [1] 0.102756
```

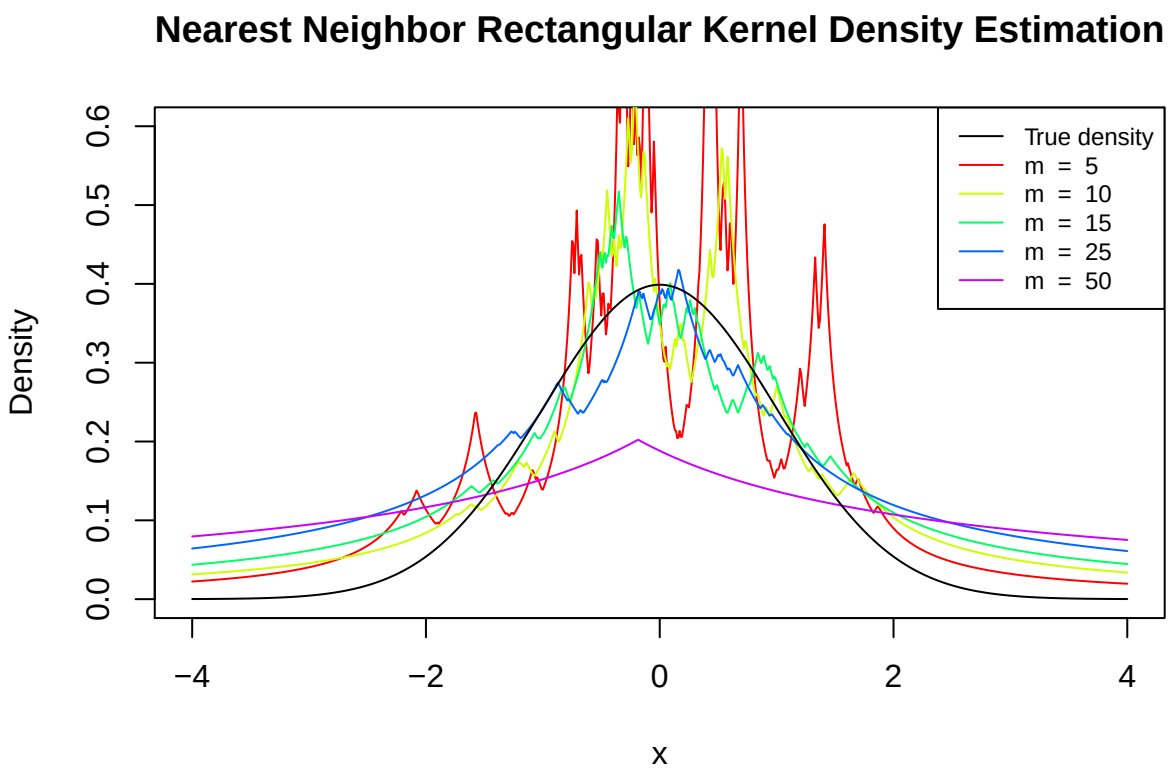
```
bw.ucv(faithful$eruptions)
```

```
## [1] 0.1019193
```

2.3 m-nächste Nachbarn

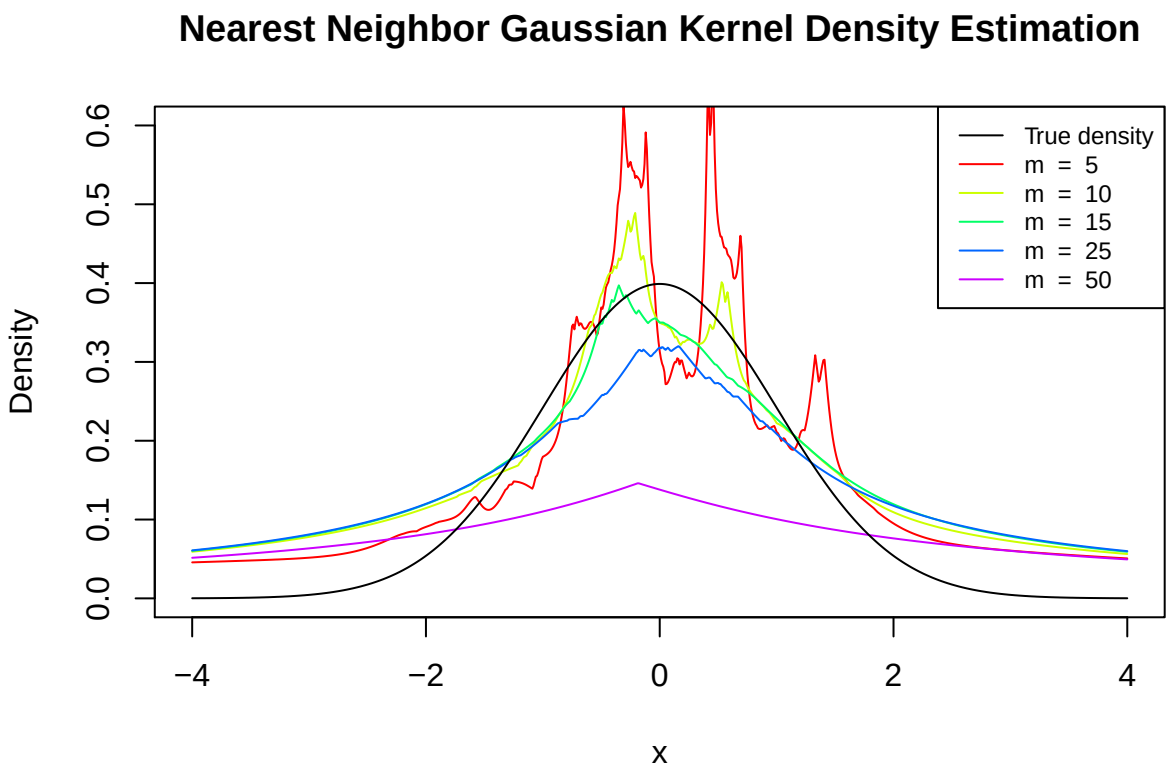
```
fnMNearestNeighbors <- function(x, X, K, m) {
  # This function computes the kernel density estimates for a given sample X and
  # kernel K with vector bandwidths h for every point, which is estimated by
  # nearest neighbor approach
  #
  # Args:
  #   x: Points for which the kernel density estimates are computed
  #   X: Data sample on which the kernel density estimation is fitted
  #   K: Kernel function to be used for smoothing
  #   m: Neighbor position
  #
  # Returns:
  #   A vector of the nearest neighbor kernel density estimates at points x
```

```
  n <- length(X)

  # Compute bandwidth for every point x
  h <- apply(abs(outer(X, x, "-")), 2, sort)[m,]

  # Compute kernel density estimation values using outer
  mKernels <- K((1/t(matrix(rep(h, n), ncol = n))) * outer(X, x, FUN = "-"))
  return((1/(n*h)) * colSums(mKernels))
}

# Generate sample
set.seed(42)
n <- 50
X <- rnorm(n)

# Define points to estimate kernel density on
x <- seq(-4, 4, 0.01)

# Define vector of neighbor positions m
m <- c(5,10,15,25,50)

# Nearest Neighbor Rectangular kernel density estimation
fnKernelPlot(x, X, fnRectangularKernel, vm = m, fnEstimation = fnMNearestNeighbors,
             title = "Nearest Neighbor Rectangular Kernel Density Estimation",
             trueDensity = dnorm, ylim = c(0, 0.6))
```

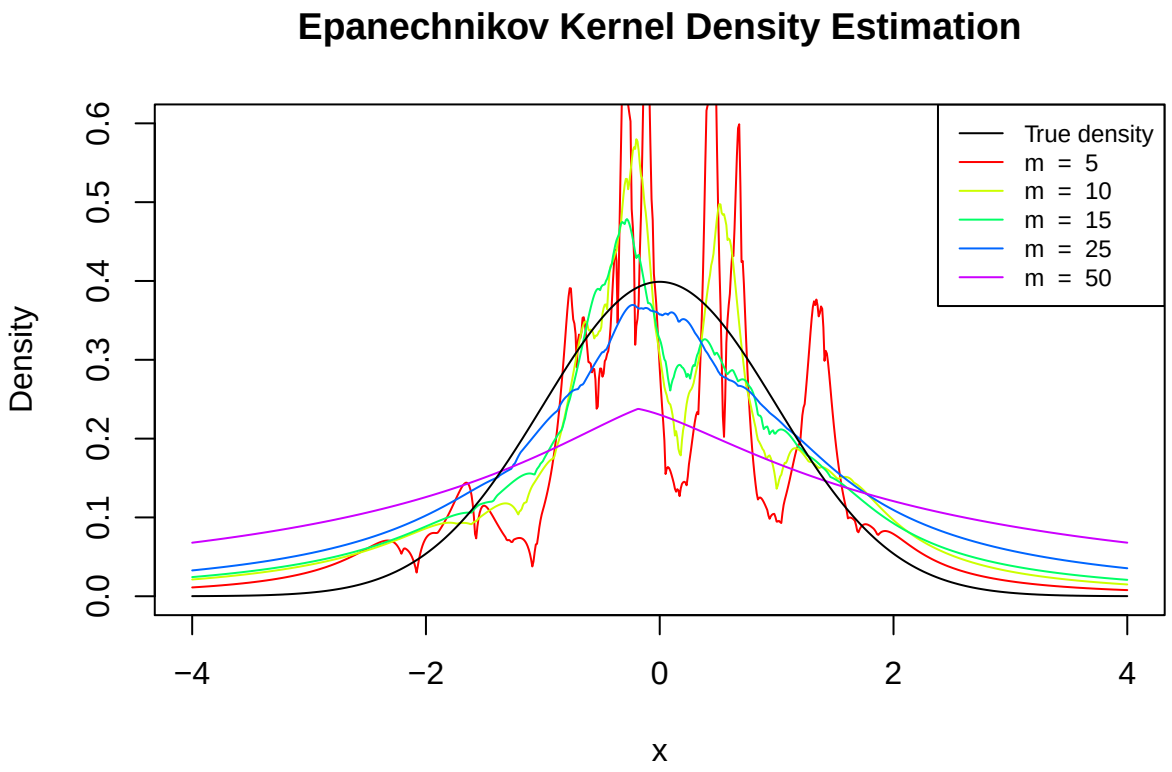



```
# Nearest Neighbor Gaussian kernel density estimation
fnKernelPlot(x, X, fnGaussianKernel, vm = m, fnEstimation = fnMNearestNeighbors,
```

```
            title = "Nearest Neighbor Gaussian Kernel Density Estimation",
            trueDensity = dnorm, ylim = c(0, 0.6))
```




```
# Nearest Neighbor Epanechnikov kernel density estimation
fnKernelPlot(x, X, fnEpanechnikovKernel, vm = m, fnEstimation = fnMNearestNeighbors,
            title = "Epanechnikov Kernel Density Estimation",
            trueDensity = dnorm, ylim = c(0, 0.6))
```



3 Bildentrauschen

```
# Load package
library(EBImage)

# Load image from parent directory
imgLenaColor <- readImage("../lena.png")

# Change image to grayscale
imgLenaGray <- channel(imgLenaColor, "gray")

# Display images
par(mfrow = c(1,2))
display(imgLenaColor, method = "raster")
display(imgLenaGray, method = "raster")
```

```
fnAddNoise <- function(img, rnoise, ...) {
  # This function adds noise specified by rnoise to a grayscale image. If the
  # image is not grayscale, it gets converted to grayscale first.
  #
  # Args:
  #   img:    Image
  #   rnoise: Random noise generation function (e.g. rnorm for Gaussian noise)
  #   ...:    Further arguments passed to or from other methods
  #
  # Returns:
  #   Grayscale image with added noise

  # Check if grayscale and convert if necessary
  if(colorMode(img) != 0) {img <- channel(img, "gray")}

  # Add noise
  m <- dim(img)[1]
  p <- dim(img)[2]
  imgNoise <- Image(imageData(img) + matrix(rnoise(m*p, ...), m, p))

  # Adjust values below 0 and above 1
  imgNoise[imgNoise < 0] <- 0
  imgNoise[imgNoise > 1] <- 1

  return(imgNoise)
}
```

```
imgLenaNoise1 <- fnAddNoise(imgLenaGray, rnorm, sd = 0.1)
imgLenaNoise2 <- fnAddNoise(imgLenaGray, rnorm, sd = 0.25)
imgLenaNoise3 <- fnAddNoise(imgLenaGray, rnorm, sd = 0.5)

par(mfrow = c(1,3))
display(imgLenaNoise1, method = "raster")
```

```r
display(imgLenaNoise2, method = "raster")
display(imgLenaNoise3, method = "raster")
```

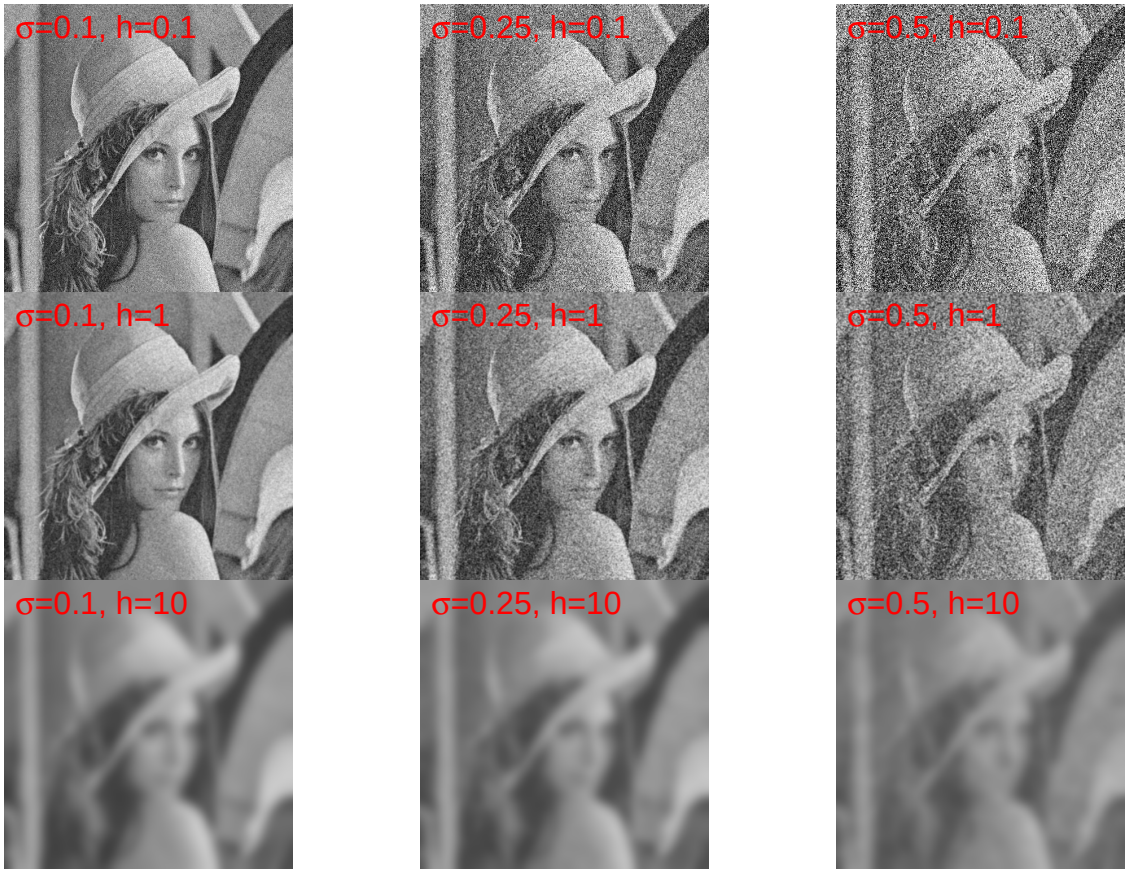
```r
fnNadarayaWatson <- function(img, K, h) {
  # The Nadaraya-Watson-estimator with weights defined by kernel K and bandwidth
  # h for denoising/smoothing an image
  #
  # Args:
  #   img:  Image to be denoised
  #   K:    Smoothing kernel used in weights
  #   h:    Bandwidth used in weights
  #
  # Returns:
  #   Denoised/smoothed image

  # Check if grayscale and convert if necessary
  if(colorMode(img) != 0) {img <- channel(img, "gray")}

  # Get Y and dimensions
  Y <- imageData(img)
  m <- dim(img)[1]
  p <- dim(img)[2]

  # Generate M and N
  M <- K((1/h) * outer(1:m, 1:m, FUN = "-"))
  M <- M / rowSums(M)

  N <- K((1/h) * outer(1:p, 1:p, FUN = "-"))
  N <- N / rowSums(N)

  return(Image(M %*% Y %*% t(N)))
}
```

```r
fnEvalNWGaussianNoise <- function(img, vsigma, vh) {
  # This function is a wrapper to compare the denoising results of images with
  # different levels of Gaussian noise for the Nadaraya-Watson-estimator with
  # weights defined by the Gaussian kernel and rectangular kernel respectively
  # for different bandwidths h.
  #
```

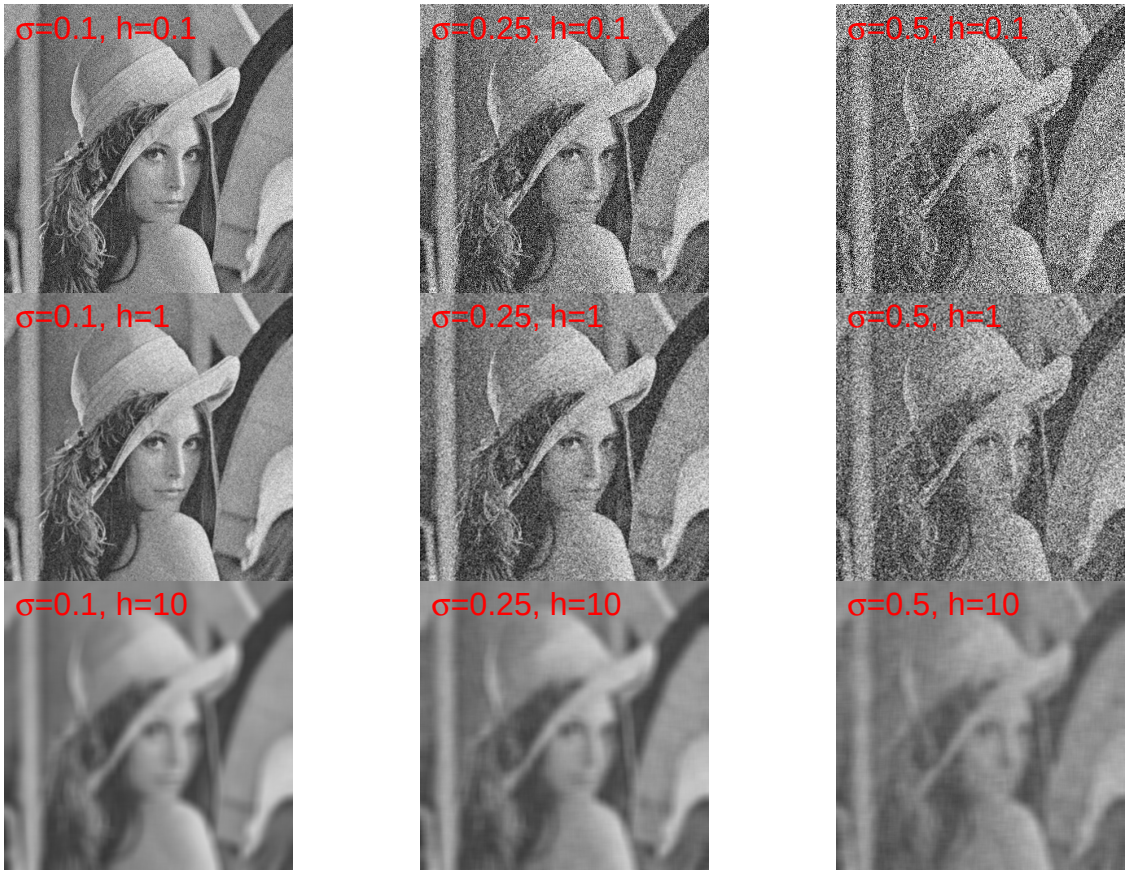
```
  # Args:
  #   img:    Image
  #   sigma:  Vector of standard deviations used to add noise
  #   h:      Vector of bandwidths used in weights of the
  #           Nadaraya-Watson-estimator
  #
  # Returns:
  #   -

  # Define helper function for label implementation
  fnLabel <- function(label) {
    text(x = 20, y = 20, adj = c(0,1), col = "red", cex = 1.5, label = label)
  }

  # Set size of frame
  par(mfrow = c(1, length(vsigma)))

  # Initalize noise image list
  imagesNoise <- list()

  for (i in 1:length(vsigma)) {
    # Add noise to image and plot
    imgNoise <- fnAddNoise(img, rnorm, sd = vsigma[i])
    imagesNoise[i] <- list(imgNoise)
    display(imgNoise, method = "raster")
    fnLabel(substitute(paste(sigma, "=", sd), list(sd=vsigma[i])))
  }

  # Set size of frame for plots below
  par(mfrow = c(length(vh), length(vsigma)))

  cat('Gaussian kernel')
  for (h in vh) {
    # NW-Denoising with Gaussian kernel
    for (i in 1:length(vsigma)) {
      display(fnNadarayaWatson(imagesNoise[[i]], fnGaussianKernel, h),
              method = "raster")
      fnLabel(substitute(paste(sigma, "=", sd, ", h=", h),
                         list(sd=vsigma[i], h=h)))
    }
  }

  cat('Rectangular kernel')
  for (h in vh) {
    # NW-Denoising with rectangular kernel
    for (i in 1:length(vsigma)) {
      display(fnNadarayaWatson(imagesNoise[[i]], fnRectangularKernel, h),
              method = "raster")
      fnLabel(substitute(paste(sigma, "=", sd, ", h=", h),
                         list(sd=vsigma[i], h=h)))
    }
  }
}
```

```
sigma <- c(0.1, 0.25, 0.5)
h <- c(0.1, 1, 10)

# Lena
fnEvalNWGaussianNoise(imgLenaGray, sigma, h)
```

Gaussian kernel



Rectangular kernel




```
# Porsche
imgPorsche <- readImage("../porsche.jpeg")
fnEvalNWGaussianNoise(imgPorsche, sigma, h)
```


```
## Gaussian kernel
```

Rectangular kernel


```
# 7. The weighted median is more robust than the mean
```

3.1 Weitere Methoden zur Bildentrauschung

3.1.1 Rangordnungsfilter

Bei Rangordnungsfilter wird eine bestimmte Anzahl von Grauwerten in einer Umgebung eines Pixels betrachtet. Die so erfassten Grauwerte werden dem Rang nach in einer Liste sortiert, also nach Größe des Grauwertes. Der aktuell betrachtete Pixel wird durch einen Grauwert aus der Liste ersetzt. Dabei kann für die Wahl der Position ein beliebiges Verfahren eingesetzt werden, z.B. Minimumfilter (minimaler Wert aus der Liste), Maximumfilter (maximaler Wert aus der Liste), Medianfilter (der Grauwert in der Mitte der Liste), etc.

Rangordnungsfilter eignen sich für Bilder mit Ausreißern, also z.B. Kratzer oder einzelne deutlich abweichende Pixel.

3.1.2 Frequenzraumfilter

Ein Bild kann sowohl im Ortsraum, als auch im Freuqenzraum beschrieben werden. Zur Transformation aus dem Ortsraum in den Frequenzraum wird eine diskrete Fouriertransformation durchgeführt. Anschließend können verschiedene Filter verwendet werden, z.B. Hochpass- oder Tiefpassfilter. Zufälliges Rauschen kann als hochfrequent angenommen werden, weshalb sich hier ein Tiefpassfilter eignen würde (die niedrigen Frequenzen bleiben unverändert). Dabei kann der Filter “hart” abschneiden oder einen Übergangsbereich definieren. Nach der Filterung werden die Daten wieder in den Ortsraum zurück transformiert.

Anwednungsfälle für den Frequenzraumfilter ergeben sich je nach Fragestellung. Für verschiedene Fälle eignen sich entsprechende Filter im Frequenzraum.