

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/einseitigen t -Test und einen zweiseitigen/einseitigen Wilcoxon-Vorzeichen-Test, jeweils zum Signifikanzniveau $\alpha = 0.05$, durch.

```
# Enter data
x <- c(82,69,73,43,58,56,76,65)
y <- c(63,42,74,37,51,43,80,62)
testType <- c('two-sided', 'one-sided')
pValueT <- c(0,0); pValueWilcox <- c(0,0)

# t-test
pValueT[1] <- t.test(x, y, alternative = "two.sided", mu = 0, conf.level = 0.95, paired = TRUE)$p.value
pValueT[2] <- t.test(x, y, alternative = "greater", mu = 0, conf.level = 0.95, paired = TRUE)$p.value

# Wilcoxon signed rank test
pValueWilcox[1] <- wilcox.test(x, y, alternative = "two.sided", mu = 0, conf.level = 0.95,
                                 paired = TRUE, conf.int = TRUE)$p.value
pValueWilcox[2] <- wilcox.test(x, y, alternative = "greater", mu = 0, conf.level = 0.95,
                                 paired = TRUE, conf.int = TRUE)$p.value

knitr::kable(data.frame(testType, pValueT, pValueWilcox), caption = 'p-Werte')
```

Table 1: p-Werte

testType	pValueT	pValueWilcox
two-sided	0.0489476	0.0546875
one-sided	0.0244738	0.0273438

1.1.0.1 Zweiseitige Tests Wir können sehen, dass der t -Test die Nullhypothese ablehnt ($p = 0.04895 < \alpha$) und somit einen signifikanten Einfluss feststellt.

Der Wilcoxon-Vorzeichen-Test hingegen ($p = 0.0546875 < \alpha$) lehnt die Nullhypothese nicht zum gegebenen Signifikanzniveau nicht ab.

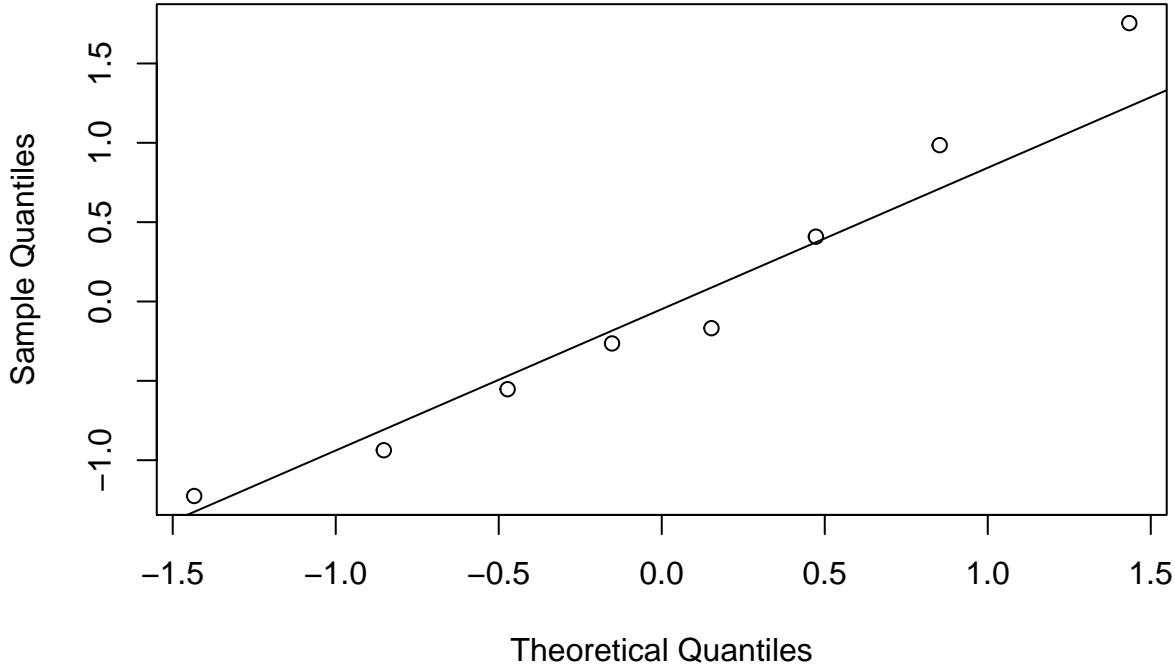
1.1.0.2 Einseitige Tests Im Fall zweiseitiger Tests unter der gleichen Nullhypothese, und der Alternative dass die sozialen Fähigkeiten signifikant höher sind, lehnen nun beide Tests die Nullhypothese ab.

Der Grund liegt darin, dass hier die p -Werte nur eine Ablehnregion in die Richtung “größer” abdecken müssen und aufgrund der Symmetrie der Tests deshalb nur halb so groß sind wie bei den zweiseitigen Tests. Beide

Tests indizieren nun einen signifikanten Anstieg der sozialen Fähigkeiten durch den Kindergartenbesuch.

1.1.0.3 Testannahmen Schließlich lohnt es, einen Blick auf die Testannahmen zu richten. Während der Wilcoxon-Test lediglich eine symmetrische Verteilung voraussetzt, nimmt der t-Test konkret die Normalverteilung, $X_i - Y_i \sim N(0, \sigma^2)$, der Daten an. Da diese stärkere Annahme ihm auch zu seiner größeren Power verhilft, sollte man ihre Validität überprüfen, hier geschieht dies mit Hilfe eines QQ-Plot:

Normal Q-Q Plot



Wie besonders am rechten Tail zu erkennen, kann die Normalverteilungsannahme durchaus in Frage gestellt werden.

Doch auch die Annahme symmetrischer Daten ist angesichts der ‘‘Schiefe’’ (2.25) der Daten anzuzweifeln, aber sie ist deutlich schwächer als die zusätzliche Annahme einer (bis auf Parameter) exakten Verteilung.

Das Hauptproblem dieser Stichprobe ist jedoch ihre geringe Größe von $N = 8$, jedwede Art von Statistik besitzt nur sehr geringe Aussagekraft.

1.2 t-Test vs. Wilcoxon-Vorzeichen-Test

Als nächstes schätzen wir die Power beider Tests in verschiedenen Settings ab. Dafür wird zunächst der wahre Wert $\theta \in [0, 0.5, 1]$ mit Fehlern verrauscht, die der Normal-/Cauchy-/ und Gleichverteilung folgen.

Blablabla

```
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
#   $WilcoxonProb: 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$TProb <- mean(vecPvalT < alpha)
result$WilcoxonProb <- mean(vecPvalWilcox < alpha)
return(result)
}

```

```

# Set seed
set.seed(432)

# Normal errors
normalMean = c(0, 0.5, 1)
normalSD = c(1, 1, 1)
normalPowerT <- list()
normalPowerW <- list()

for (i in 1:3) {
  res <- fnTestPowerMC(fnError = rnorm, nSim = 10^4,
                        mean = normalMean[i], sd=normalSD[i])
  normalPowerT[i] <- res[1]
  normalPowerW[i] <- res[2]
}
# Cauchy errors (t-distribution with df = 1)
cauchyPowerT <- list()
cauchyPowerW <- list()
cauchyLoc = c(0, 0.5, 1)

```

```

cauchyScale = c(1,1,1)
for (i in 1:3) {
  res <- fnTestPowerMC(fnError = rcauchy, nSim = 10^4,
                        location = cauchyLoc[i], scale=cauchyScale[i])
  cauchyPowerT[i] <- res[1]
  cauchyPowerW[i] <- res[2]
}

# Uniform errors
uniMin <- c(-1, -0.5, 0)
uniMax <- c(1, 1.5, 2)
uniPowerT <- list()
uniPowerW <- list()

for (i in 1:3) {
  res <- fnTestPowerMC(fnError = runif, nSim = 10^4,
                        min = uniMin[i], max = uniMax[i])
  uniPowerT[i] <- res[1]
  uniPowerW[i] <- res[2]
}

```

Table 2: normalverteiltes Rauschen

mean	sd	power	T-test	power	Wilcox-test
0.0	1		0.0492		0.0506
0.5	1		0.7528		0.7387
1.0	1		0.9992		0.9990

Table 3: uniformes Rauschen

mean	+- interval	power	T-test	power	Wilcox-test
0.0		1	0.0504		0.0495
0.5		1	0.9973		0.9871
1.0		1	1.0000		1.0000

Table 4: cauchyverteiltes Rauschen

location	scale	power	T-test	power	Wilcox-test
0.0	1		0.0209		0.0527
0.5	1		0.0712		0.2928
1.0	1		0.2022		0.6927

1.2.0.4 Normalverteiltes Rauschen Wie zu erwarten gibt es beim normalverteilten Rauschen kaum merkliche Unterschiede.

1.2.0.5 Uniformes Rauschen t-test Überraschend gut im Vergleich zu Normal, quasi identisch. Liegt vermutlich an der ...

1.2.0.6 Cauchy-Rauschen Deutlich schlechter als Wilcox, Cauchy-Verteilung stärker konzentriert als Normalverteilung, t-Test überschätzt Tails.

2 Dichteschätzung

In dieser Aufgabe ist es unser Ziel eine unbekannte Dichte f mittels Sample $X_1, \dots, X_n \stackrel{iid}{\sim} \mathbb{P}_f$ zu approximieren. Für die Schätzung untersuchen und verwenden wir eine nichtparametrische Methode - den Kerndichteschätzer. Wir werden sehen, dass die „Güte“ der Approximation von der Wahl des Kerns K sowie von der Bandweitenwahl h abhängt und werden versuchen zu gegebenem Kern K mittels Kreuzvalidierung eine „optimale“ Bandweite zu wählen.

2.1 Kerndichteschätzer

Für gegebene Daten X_1, \dots, X_n können wir einen allgemeinen Kerndichteschätzer \hat{f}_n für einen Kern K und Bandweite h wie folgt für einen Vektor von Stellen x implementieren:

```
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))
}
```

Bevor wir die Methode für Zufallszahlen verschiedener Verteilungen testen, implementieren wir noch drei Kerne (Rechteckskern, Gaußkern, Epanechnikov-Kern) und schreiben eine allgemeine Funktion zur Erstellung von Plots (welche bereits für die spätere Anwendung der m -nächste Nachbarn Methode angepasst wurde):

```
# 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))
}
```

```

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.
  #
  # 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 neighbor positions
  #   fnEstimation: Type of estimation used (kernel density or knn)
  #   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
    strPar <- "h"
  } else if(!is.null(vm)) {
    vPar <- vm
    strPar <- "m"
  } else {
    stop("You need to set bandwidths vh OR neighbor ranks vm")
  }

  nPar <- length(vPar)

  # Create color palette for number of hyperparameters
  cols <- rainbow(nPar, 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 (nPar > 1) {
    for (i in 2:nPar) {
      lines(x, fnEstimation(x, X, K, vPar[i]), col = cols[i])
    }
  }

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

```

2.1.1 Anwendung auf standardnormalverteilte Zufallszahlen

```

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

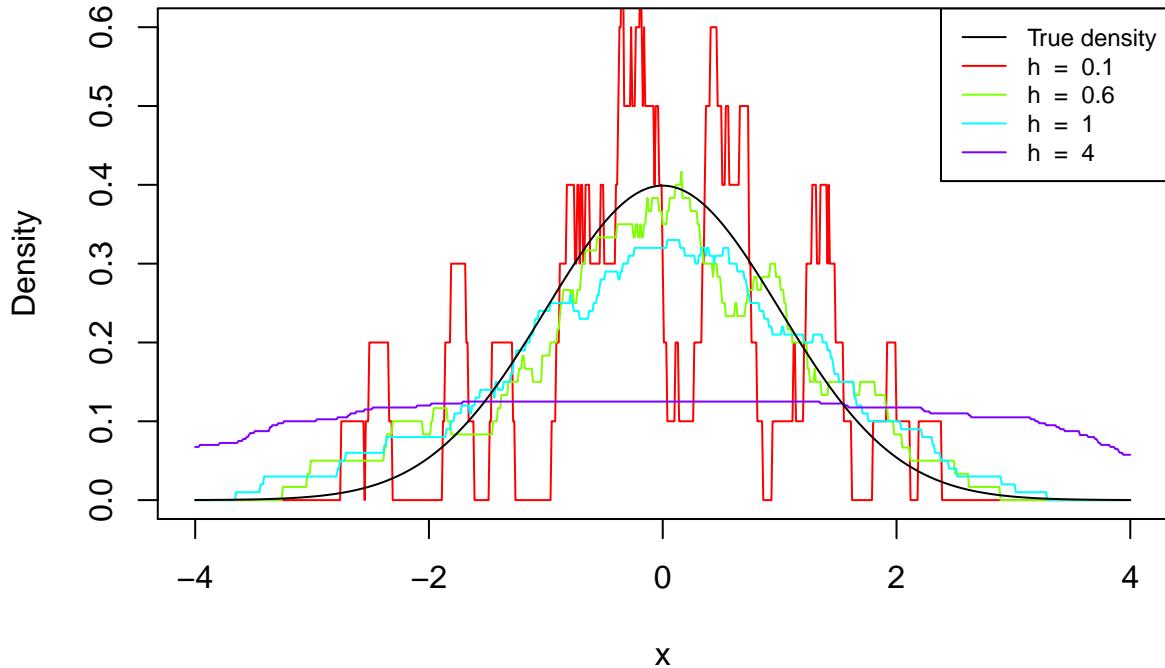
# Define points to estimate kernel density on
x <- seq(-4, 4, 0.01)

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

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

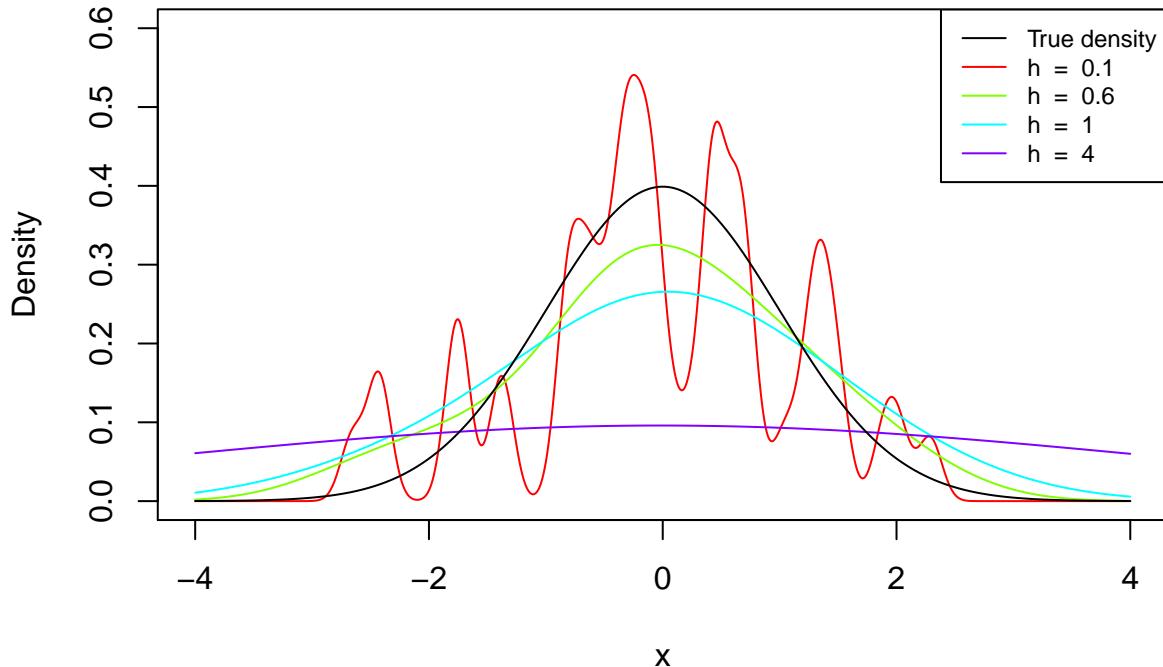
```

Rectangular Kernel Density Estimation



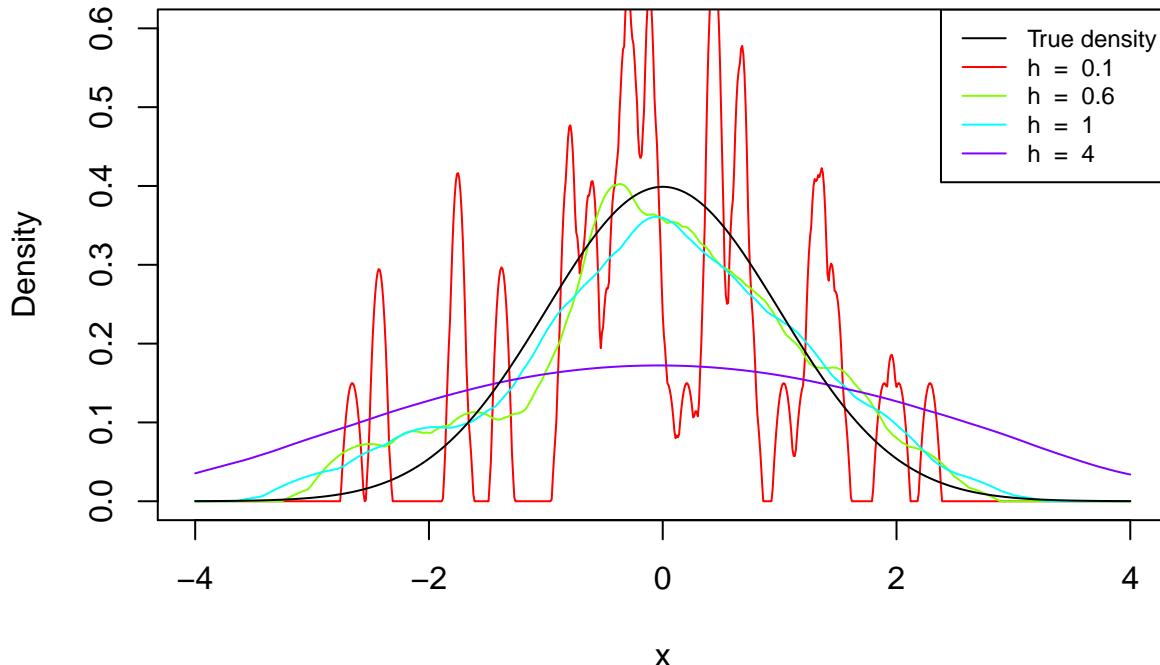
```
# Gaussian kernel density estimation
fnKernelPlot(x, X, fnGaussianKernel, h,
             title = "Gaussian Kernel Density Estimation",
             trueDensity = dnorm, ylim = c(0, 0.6))
```

Gaussian Kernel Density Estimation



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

Epanechnikov Kernel Density Estimation



Wir können sehen, dass die Wahl des Kerns die Glattheit der geschätzten Dichtefunktion beeinflusst. Da der Rechteckskern sowie der Epanechnikov-Kern den Träger $[-1, 1]$ (bzw. $(-1, 1)$) besitzen, finden sich Sprünge in den geschätzten Dichten. Da (mit h skalierte) Differenzen in $[-1, 1]$ für den Rechtseckskern gleich gewichtet werden, sind die geschätzten Dichten nochmals sprunghafter im Vergleich zum Epanechnikov-Kern, welcher symmetrisch um 0 monoton abfällt, was eine Glättung zur Folge hat. Dabei nimmt die Sprunghöhe beim Rechtseckskern mit steigender Bandbreite h ab. Im Vergleich dazu hat der Gaußkern ganz \mathbb{R} als Träger und ist ebenso symmetrisch um 0 monoton fallend. D.h. für die Schätzung $\hat{f}_n(x)$ an der Stelle x hat jeder Datenpunkt $X_i, i = 1, \dots, n$, einen (mit wachsendem Abstand geringeren) Einfluss auf die Schätzung. Dadurch resultieren glatte Kerndichteschätzer \hat{f}_n .

Weiter können wir in den Beispielen schön das Bias-Varianz-Dilemma der Bandweitenwahl beobachten. Für kleineres h wird die lokale Umgebung, die für die Schätzung an der Stelle x herangezogen wird, ebenso kleiner. Dies hat eine stärkere Oszillation der Schätzung \hat{f}_n zur Folge. In diesem Fall ist der Bias klein (lokale Schätzung), die Varianz jedoch hoch und das Resultat eine Unterglättung. Umgekehrt folgt für eine größere Bandweite h ein größerer Bias (globale Schätzung), jedoch eine geringere Varianz. In diesem Fall sprechen wir von Überglättung. Die Aufgabe einer optimalen Bandweitenwahl ist es also einen Ausgleich zwischen Bias und Varianz zu finden, so dass der insgesamt resultierende Fehler möglichst minimal wird.

2.1.2 Anwendung auf exponentialverteilte Zufallszahlen

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

# Define points to estimate kernel density on
x <- seq(0, 6, 0.01)
```

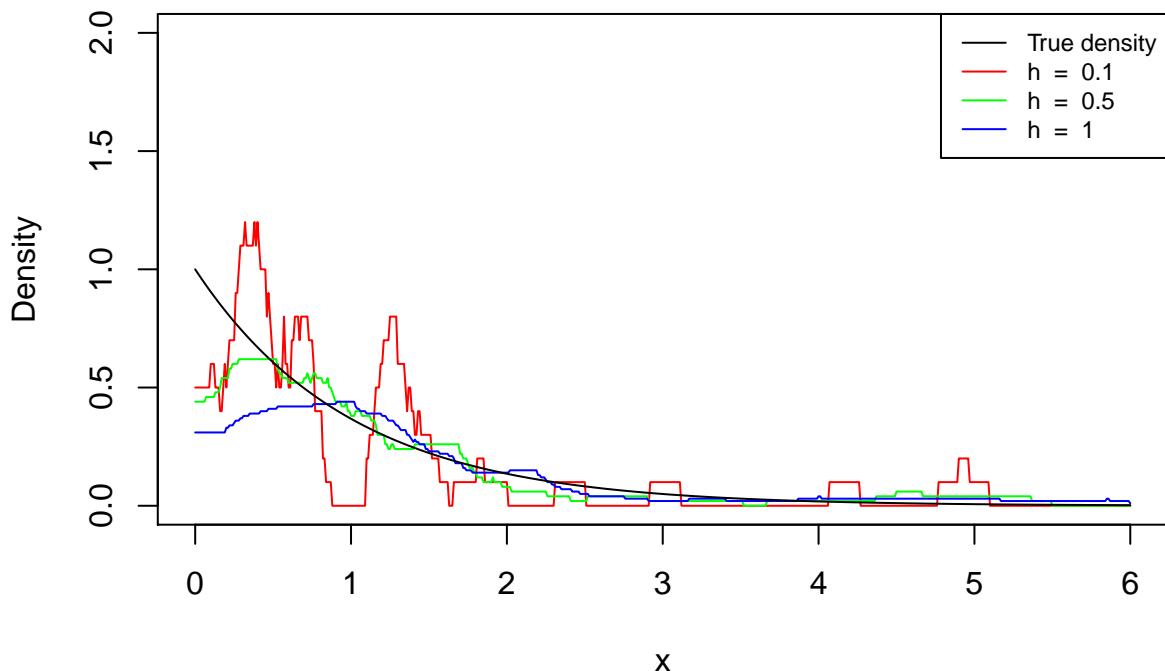
```

# Set different bandwidths
h <- c(0.1, 0.5, 1)

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

```

Rectangular Kernel Density Estimation

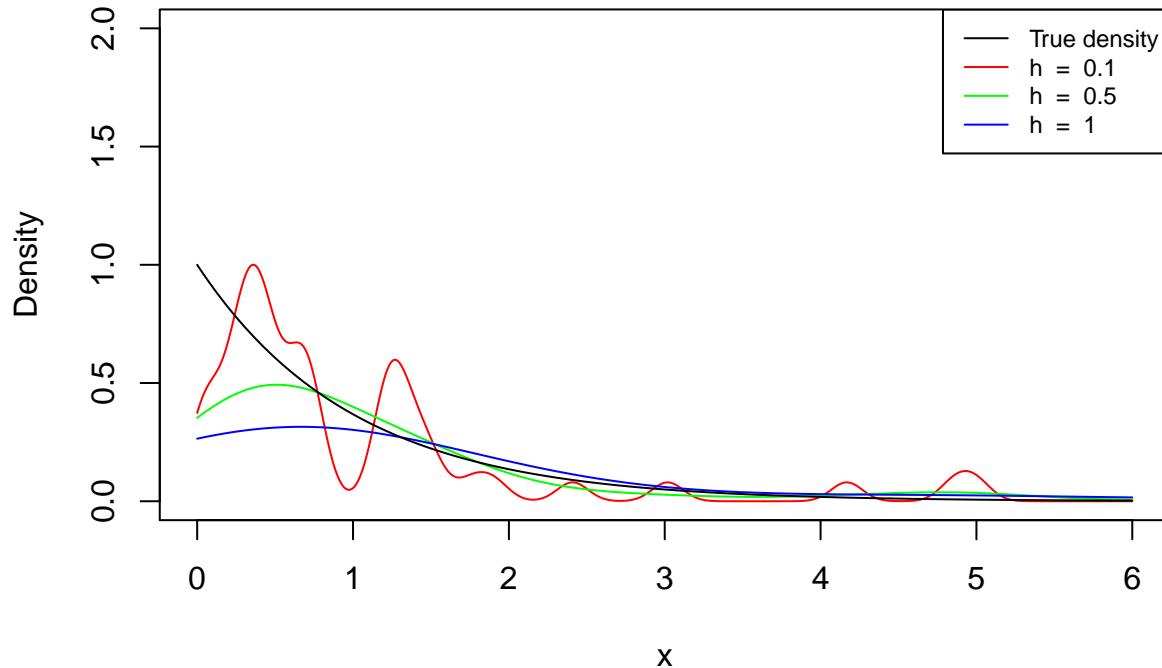


```

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

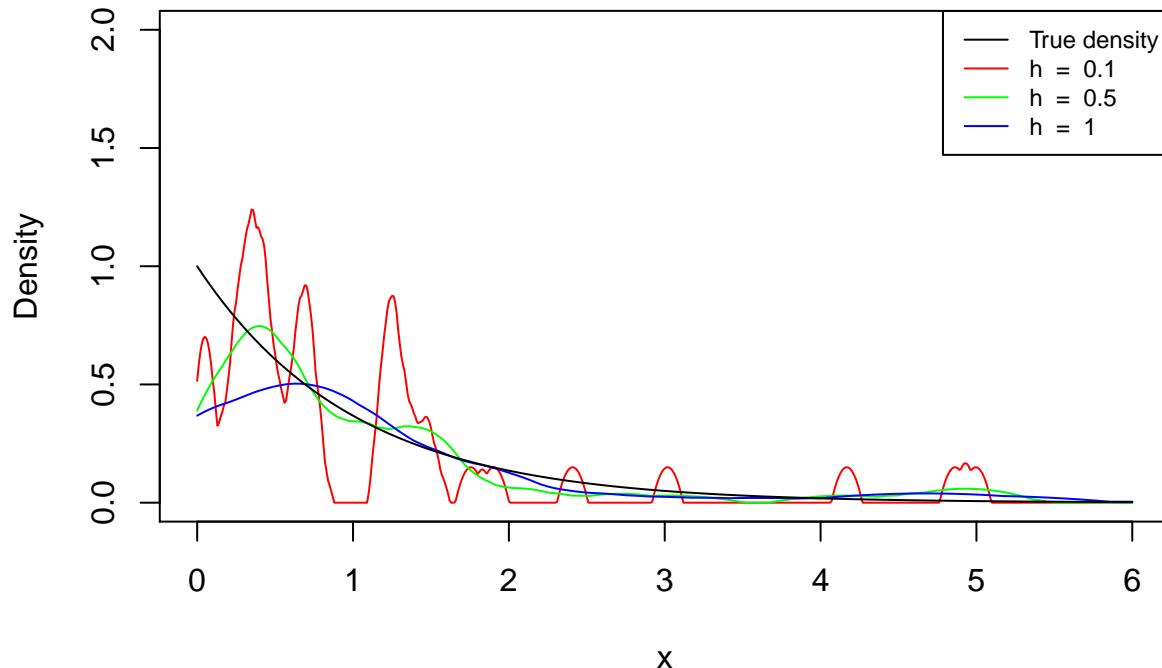
```

Gaussian Kernel Density Estimation



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

Epanechnikov Kernel Density Estimation



Bei den Kerndichteschätzungen für exponentialverteilte Zufallszahlen können wir ebenso den Einfluss der unterschiedlichen Kerne auf die Glattheit der Schätzungen sowie das Bias-Varianz-Dilemma beobachten. Insgesamt deuten die Kerndichteschätzungen auf eine rechtsschiefe Verteilung hin, was in der Anwendung rechtsschiefe parametrische Likelihood-Modelle (Exponentialverteilung, Log-Normalverteilung, Chi-Quadrat-Verteilung, etc.) für die weitere Modellierung motivieren könnte.

2.1.3 Anwendung auf Cauchy-verteilte Zufallszahlen

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

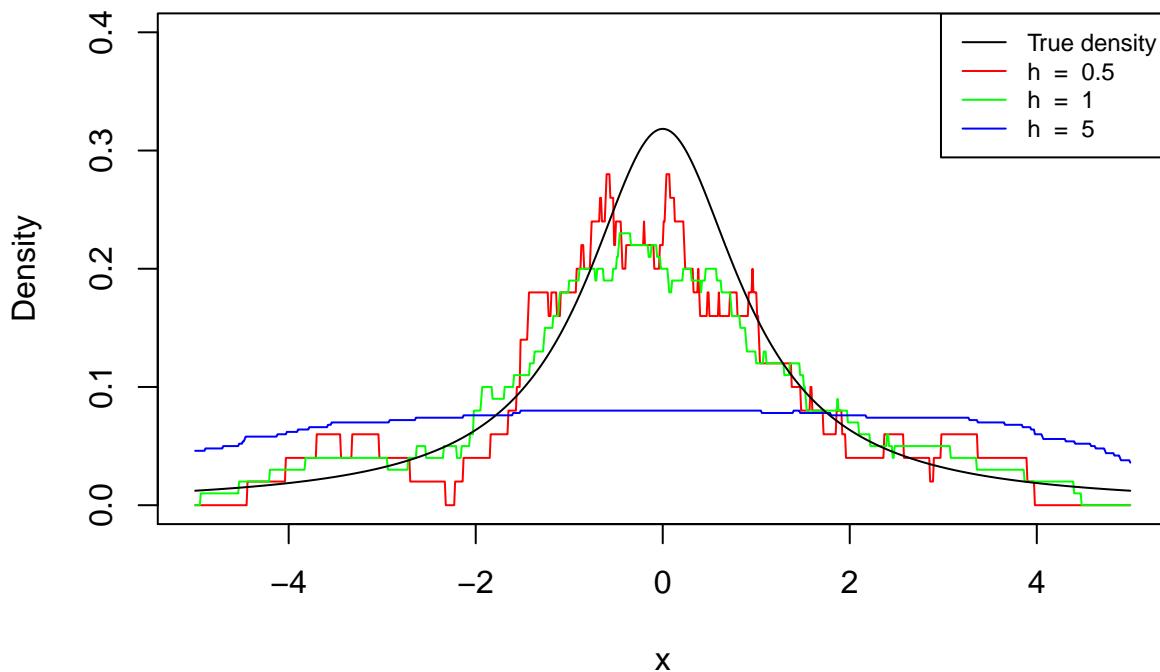
# Define points to estimate kernel density on
x <- seq(-5, 5, 0.01)

# Set different bandwidths
h <- c(0.5, 1, 5)

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

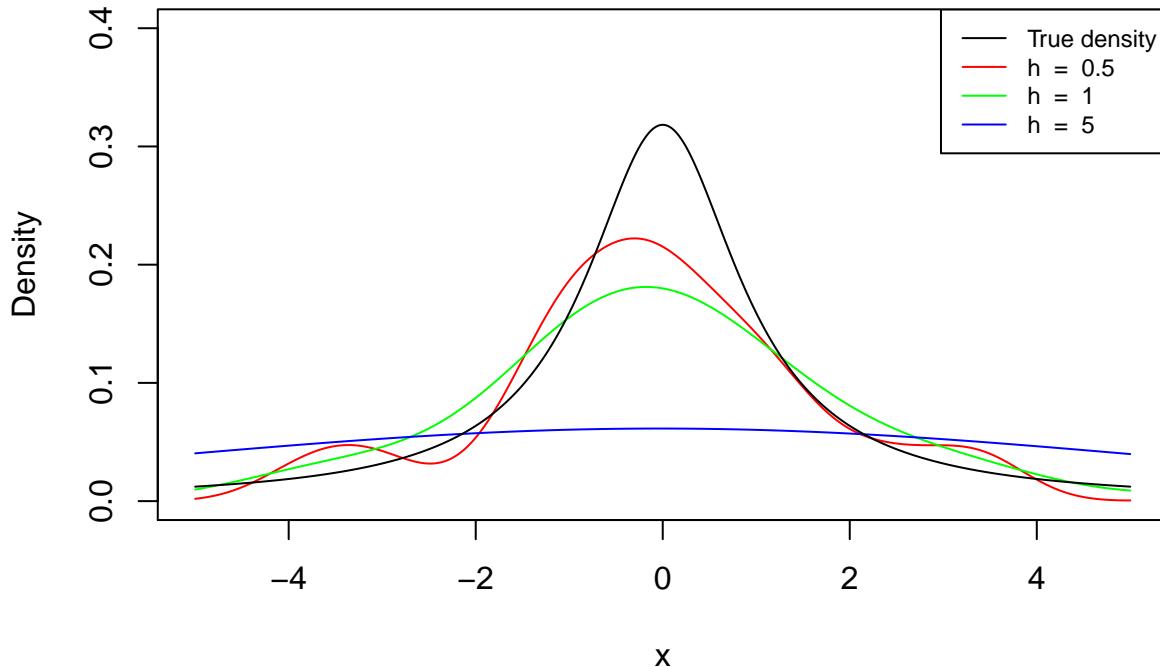
# Rectangular kernel density estimation
fnKernelPlot(x, X, fnRectangularKernel, h,
              title = "Rectangular Kernel Density Estimation",
              trueDensity = dcauchy, ylim = c(0, 0.4))
```

Rectangular Kernel Density Estimation



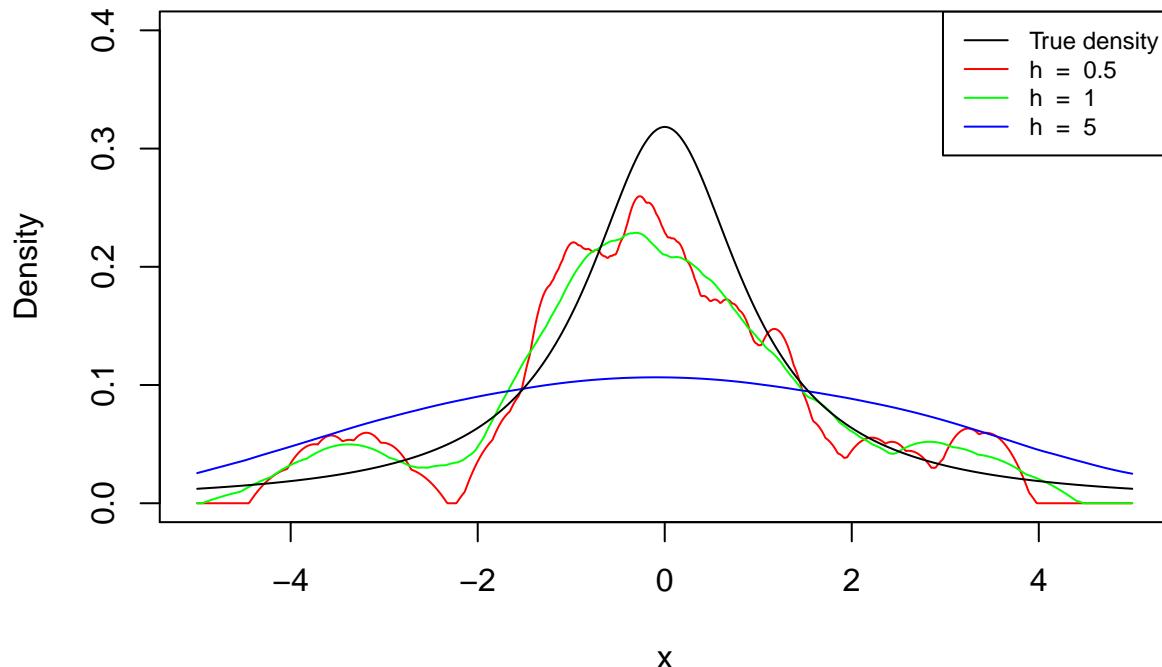
```
# Gaussian kernel density estimation
fnKernelPlot(x, X, fnGaussianKernel, h,
             title = "Gaussian Kernel Density Estimation",
             trueDensity = dcauchy, ylim = c(0, 0.4))
```

Gaussian Kernel Density Estimation



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

Epanechnikov Kernel Density Estimation



Auch hier können wir den Einfluss der unterschiedlichen Kerne auf die Glattheit der Schätzungen sowie das Bias-Varianz-Dilemma beobachten.

2.1.4 Anwendung auf faithful Datensatz

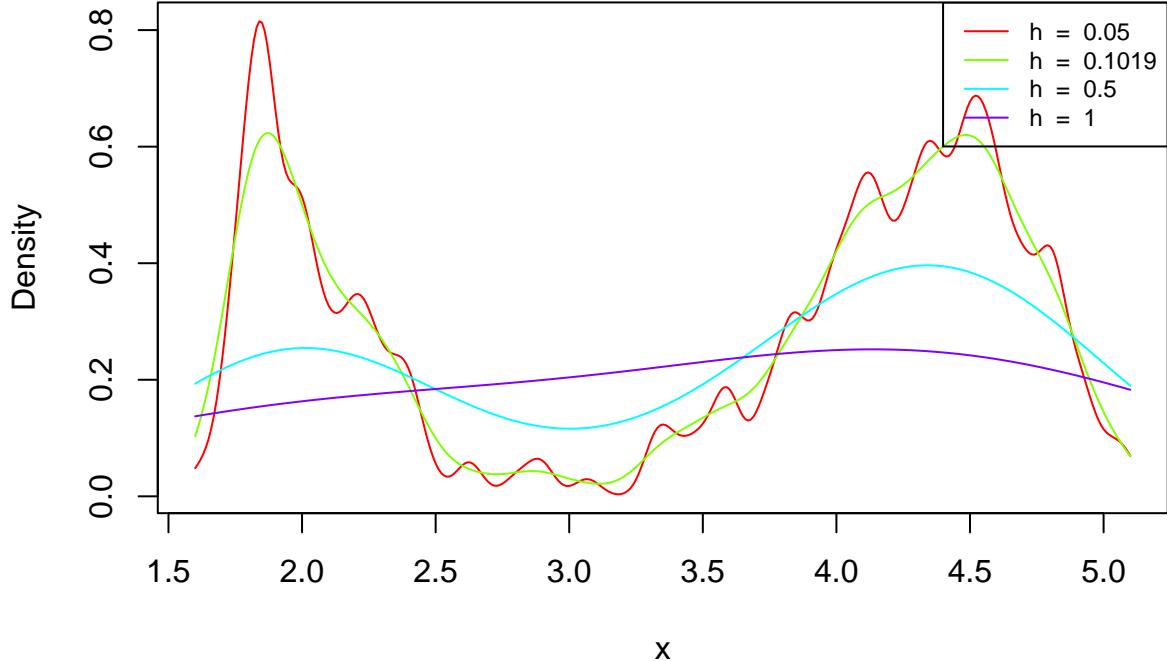
```
# Faithful kernel density estimation
X <- faithful$eruptions

# 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)
```

Gaussian Kernel Density Estimation for faithful eruption data



Wir können sehen, dass die Anwendung des Kerndichteschätzers mit Gaußkern für verschiedene Bandweiten eine bimodale Verteilung der Eruptionsdauern im Datensatz `faithful` aufzeigt, was auf zwei unterschiedliche Gruppen von Ausstößen hindeutet.

2.2 Kreuzvalidierung zur Bandweitenwahl

Die Idee der Kreuzvalidierung ist es die Bandweite h so zu wählen, dass der Mean Integrated Squared Error (MISE) minimal wird. Es gilt:

$$\begin{aligned} \operatorname{argmin}_h \text{MISE}(h) &= \operatorname{argmin}_h \mathbb{E} \left[\int (\hat{f}_n(x) - f(x))^2 dx \right] \\ &= \operatorname{argmin}_h \mathbb{E} \left[\int \hat{f}_n^2(x) dx - 2 \int \hat{f}_n(x)f(x) dx \right] + \int f^2(x) dx \\ &= \operatorname{argmin}_h \mathbb{E} \left[\int \hat{f}_n^2(x) dx - 2 \int \hat{f}_n(x)f(x) dx \right] = \operatorname{argmin}_h J(h) \end{aligned}$$

mit J definiert durch

$$J(h) := \mathbb{E} \left[\int \hat{f}_n^2(x) dx - 2 \int \hat{f}_n(x)f(x) dx \right], \quad h > 0.$$

2.2.1 Erwartungstreue des Kreuzvalidierungskriteriums $\hat{J}(h)$

Wir zeigen nun, dass das Kreuzvalidierungskriterium $\hat{J}(h)$ definiert durch

$$\hat{J}(h) := \int \hat{f}_n^2(x) dx - 2\hat{G}, \quad h > 0,$$

mit

$$\hat{G} := \frac{1}{n} \sum_{j=1}^n \hat{f}_{n,-j}(X_j), \quad \hat{f}_{n,-j}(x) := \frac{1}{(n-1)h} \sum_{k \neq j} K\left(\frac{X_k - x}{h}\right)$$

ein erwartungstreuer Schätzer für $J(h)$ ist.

Seien $X_1, \dots, X_n \stackrel{iid}{\sim} \mathbb{P}_f$ und betrachte für $h > 0$ und $j \in \{1, \dots, n\}$

$$\begin{aligned} \mathbb{E}[\hat{f}_{n,-j}(X_j)] &= \mathbb{E}\left[\frac{1}{(n-1)h} \sum_{k \neq j} K\left(\frac{X_k - X_j}{h}\right)\right] \\ &\stackrel{iid}{=} \frac{1}{(n-1)h} \sum_{k \neq j} \int \int K\left(\frac{y-x}{h}\right) f(y) dy f(x) dx \\ &= \frac{1}{h} \int \int K\left(\frac{y-x}{h}\right) f(y) dy f(x) dx \\ &= \int \int K(y) f(x+hy) dy f(x) dx. \end{aligned}$$

Ebenso gilt

$$\begin{aligned} \mathbb{E}\left[\int \hat{f}_n(x) f(x) dx\right] &= \mathbb{E}\left[\int \frac{1}{nh} \left(\sum_{k=1}^n K\left(\frac{X_k - x}{h}\right)\right) f(x) dx\right] \\ &\stackrel{\text{Fubini}}{=} \frac{1}{nh} \sum_{k=1}^n \int \mathbb{E}\left[K\left(\frac{X_k - x}{h}\right)\right] f(x) dx \\ &\stackrel{iid}{=} \frac{1}{h} \int \mathbb{E}\left[K\left(\frac{X_1 - x}{h}\right)\right] f(x) dx \\ &= \int \int K(y) f(x+hy) dy f(x) dx. \end{aligned}$$

Somit folgt also

$$\begin{aligned} \mathbb{E}[\hat{J}(h)] &= \mathbb{E}\left[\int \hat{f}_n^2(x) dx - 2\hat{G}\right] \\ &= \mathbb{E}\left[\int \hat{f}_n^2(x) dx\right] - 2\mathbb{E}[\hat{G}] \\ &= \mathbb{E}\left[\int \hat{f}_n^2(x) dx\right] - 2\mathbb{E}\left[\frac{1}{n} \sum_{j=1}^n \hat{f}_{n,-j}(X_j)\right] \\ &\stackrel{iid}{=} \mathbb{E}\left[\int \hat{f}_n^2(x) dx\right] - 2\mathbb{E}[\hat{f}_{n,-1}(X_1)] \\ &= \mathbb{E}\left[\int \hat{f}_n^2(x) dx - 2 \int \hat{f}_n(x) f(x) dx\right] = J(h). \end{aligned}$$

Aus dieser Herleitung sind auch die notwendigen Integrationsbedingungen an K und f erkennbar. Damit $\mathbb{E}[\hat{G}] < \infty$, muss für $h > 0$ also $y \mapsto K(y)f(x+hy) \in \mathcal{L}^1(\mathbb{R}, \mathcal{B}(\mathbb{R}), \lambda)$ für jedes $x \in \mathbb{R}$ gelten. Betrachten wir

$$\begin{aligned} \mathbb{E}\left[\int \hat{f}_n^2(x) dx\right] &= \mathbb{E}\left[\frac{1}{(nh)^2} \int \left(\sum_{k=1}^n K\left(\frac{X_k - x}{h}\right)\right)^2 dx\right] \\ &\stackrel{\text{Fubini}}{=} \frac{1}{(nh)^2} \int \mathbb{E}\left[\left(\sum_{k=1}^n K\left(\frac{X_k - x}{h}\right)\right)^2\right] dx \\ &= \frac{1}{(nh)^2} \left\{ \sum_{k=1}^n \int \mathbb{E}\left[K^2\left(\frac{X_k - x}{h}\right)\right] dx + 2 \sum_{k \neq j} \int \mathbb{E}\left[K\left(\frac{X_k - x}{h}\right) K\left(\frac{X_j - x}{h}\right)\right] dx \right\} \\ &\stackrel{iid}{=} \frac{1}{nh} \int K^2(y) f(x+hy) dy dx + \frac{2(n-1)}{n^2} \int \left(\int K(y) f(x+hy) dy\right)^2 dx \end{aligned}$$

so können wir weiter sehen, dass die quadratische Integrationsannahme $y \mapsto K^2(y)f(x+hy) \in \mathcal{L}^1(\mathbb{R}, \mathcal{B}(\mathbb{R}), \lambda)$ für alle $x \in \mathbb{R}$ ebenso erfüllt sein muss, damit $\mathbb{E}[\hat{J}(h)]$ für $h > 0$ existiert.

2.2.2 Implementierung der Kreuzvalidierung zur Bandweitenwahl

Wir implementieren die Kreuzvalidierung in zwei Schritten: wir schreiben erstens eine Funktion `fnJ` die das Kreuzvalidierungskriterium $\hat{J}(h)$ berechnet und zweitens eine Funktion `fnUCV` die dann das Kreuzvalidierungskriterium minimiert. Um das Integral über den quadratischen Kerndichteschätzer zu berechnen nutzen wir die R-Methode `integrate`. Für die Minimierung in `fnUCV` bedienen wir uns der R-Methode `optimize`.

```

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(h0pt$minimum)
}

```

Wir berechnen nun mit unserer Implementierung der Kreuzvalidierung die optimale Bandweite h^* für die bereits zuvor betrachteten Eruptionsdauern im Datensatz `faithful`. Für den Kerndichteschätzer wählen wir einen Gaußkern:

```

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

## [1] 0.102756

```

Vergleichen wir dieses Ergebnis mit der optimalen Bandweite gemäß der R-Methode `bw.ucv` (bandwidth unbiased cross validation), so erhalten wir ein sehr ähnliches Ergebnis:

```

bw.ucv(faithful$eruptions)

```

```

## [1] 0.1019193

```

Es ist anzunehmen, dass die Implementierung der erwartungstreuen Kreuzvalidierung in `bw.ucv` effizienter ist als unsere, da die R-Methode nur für einen Gaußkern implementiert ist, d.h. hierfür ggf. analytische Möglichkeiten ausgenutzt wurden. Im Vergleich dazu besteht in unserer Implementierung die Möglichkeit weitere Kerne anzuwenden.

2.3 m -nächste Nachbarn

Zusätzlich zur Kerndichteschätzung mit fixer Bandweite $h > 0$ wollen wir die m -nächste Nachbarn Methode mit variabler Bandweite implementieren. Für eine Stelle $x \in \mathbb{R}$ ist die Bandweite $h(x, m)$ bei der $knn(m)$ -Methode durch den Abstand von x zum m -nächsten Nachbarn definiert.

```

fnMNearestNeighbors <- function(x, X, K, m) {
  # This function computes the kernel density estimates for a given sample X and
  # kernel K with variable bandwidths h for every point, which are determined by
  # the m-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))
}

```

Betrachten wir die $knn(m)$ -Schätzungen für standard-normalverteilte Daten wiederum für den Rechteckskern, Gaußkern sowie Epanechnikov-Kern für verschiedene m :

```

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

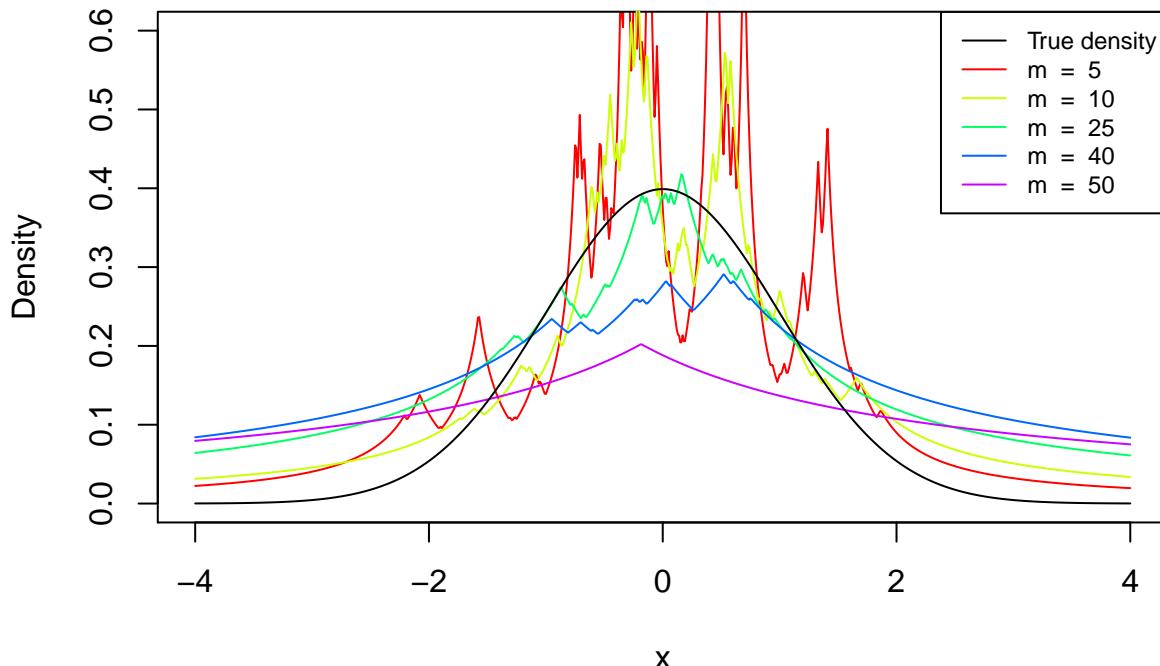
# Define points to estimate kernel density on
x <- seq(-4, 4, 0.01)

# Define vector of neighbor positions m
m <- c(5,10,25,40,50)

# 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 Rectangular Kernel Density Estimation



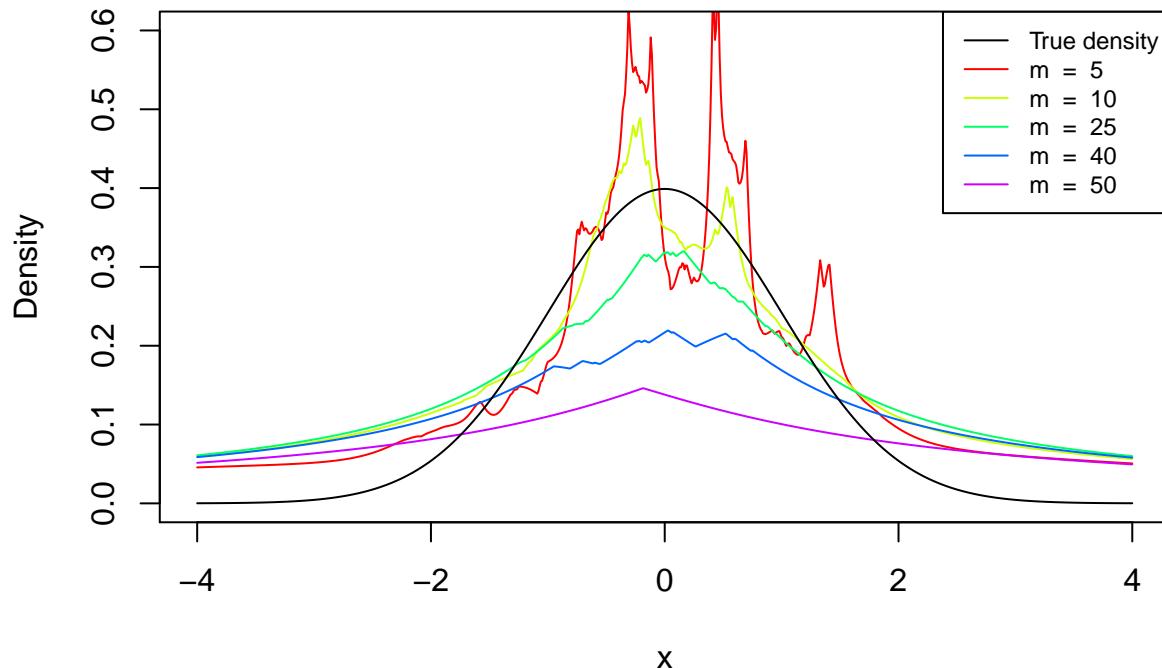
```

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

```

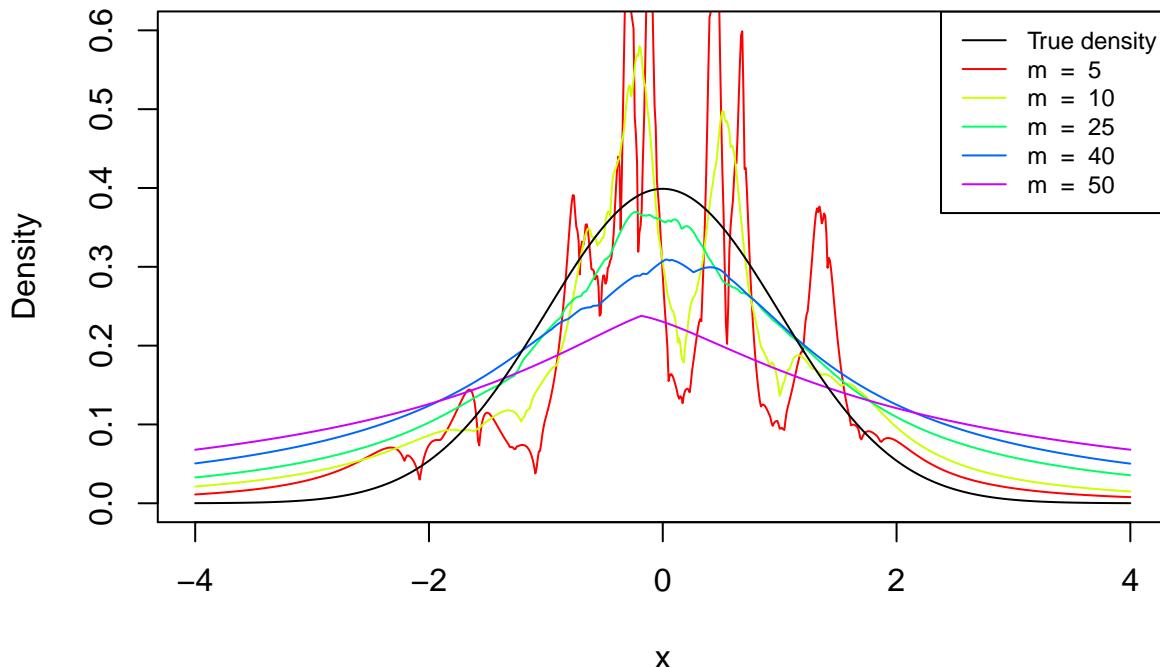
```
title = "Nearest Neighbor Gaussian Kernel Density Estimation",
trueDensity = dnorm, ylim = c(0, 0.6))
```

Nearest Neighbor Gaussian Kernel Density Estimation



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

Nearest Neighbor Epanechnikov Kernel Density Estimation



Im Vergleich zur Kerndichteschätzung mit fixer Bandweite $h > 0$ passt die $knn(m)$ -Methode die Bandweite in Abhängigkeit des Abstands der Stelle x zu den m -nächsten Datenpunkten aus X_1, \dots, X_n variabel an. Dies hat zur Folge, dass die Bandweiten zu x in Umgebungen, in welchen sich Datenpunkte häufen, kleiner gewählt werden und somit lokaler geschätzt wird. Dagegen wird für Stellen x , für welche die Umgebung zum m -nächsten Nachbarn groß ist eine größere Bandweite gewählt, was eine globale Schätzung zur Folge hat. Diese Überlegung wird in den Plots bestätigt. In kleineren Umgebungen um 0, in welchen sich die standard-normalverteilten Zufallszahlen häufen, oszillieren die $knn(m)$ -Schätzungen stärker als in den Tails. Damit wird tendenziell in kleineren Umgebungen um 0 unterglättet und in den Tails überglättet. Weiter lässt sich wie zu erwarten erkennen, dass mit größerem m die Schätzungen glatter werden, da für größeres m die Bandweiten $h(x, m)$ für alle Stellen x verhältnismäßig größer werden.

Insgesamt ist im Vergleich zur Kerndichteschätzung mit fixer Bandweite $h > 0$ tendenziell eine stärkere Oszillation der Schätzungen festzustellen.

3 Bildentrauschen

In dieser Aufgabe wollen wir mit Hilfe des Nadaraya-Watson-Schätzers versuchen Bilder zu entrauschen. Zunächst laden wir das Bild `lena.png` und wandeln es in Graustufen um:

```
# 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")
```



Als nächstes implementieren wir eine Funktion die es uns ermöglicht ein verrauschtes Bild zu erzeugen, wobei die Verteilung des Rauschens sowie das Rauschniveau spezifiziert werden kann:

```
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)
}
```

Betrachten wir drei verrauschte Bilder mit Gaußschem Rauschen für unterschiedliche Rauschniveaus:

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



3.1 Nadaraya-Watson-Schätzer

In diesem Abschnitt wollen wir einen geeigneten Nadaraya-Watson-Schätzer zur Bildentrauschung angeben sowie effizient implementieren.

Dazu zeigen wir zunächst, dass $K_2(x, y) := K(x)K(y)$, $x, y \in \mathbb{R}$ für einen eindimensionalen Kern K einen zweidimensionalen Kern definiert: Da K ein eindimensionaler Kern ist, ist K eine nichtnegative, reelle Funktion mit

$$\int K(y) dy = 1 \quad \text{und} \quad K(x) = K(-x) \quad \text{für alle } x \in \mathbb{R}.$$

Damit ist K_2 nichtnegative, reelle Funktion mit

$$\int K_2(x, y) d(x, y) = \int \int K(x)K(y) dx dy = 1^2 = 1$$

nach Fubini sowie

$$K_2(x, y) = K(x)K(y) = K(-x)K(-y) = K_2(-x, -y)$$

für alle $x, y \in \mathbb{R}$.

Als nächstes geben wir einen geeigneten Nadaraya-Watson-Schätzer an, um die Pixel des ursprünglichen Bildes $(f(i, j))_{i=1, \dots, m, j=1, \dots, p}$ aus den Pixeln des verrauschten Bildes $(Y_{ij})_{i=1, \dots, m, j=1, \dots, p}$ mit $Y_{ij} = f(i, j) + \varepsilon_{ij}$ für $\varepsilon_{ij} \stackrel{iid}{\sim} N(0, \sigma^2)$ mit Rauschniveau $\sigma > 0$ zu schätzen.

Die Idee der Entrauschung ist es, den Pixel des ursprünglichen Bildes mit Pixeln aus einer Umgebung des zu entrauschenden Pixels zu schätzen. Wir betrachten daher die Designpunkte $(i, j) \in \{1, \dots, m\} \times \{1, \dots, p\}$ als Regressoren in der nichtparametrischen Regression, d.h.

$$\hat{f}_{mp}(i, j) := \underset{y \in [0, 1]}{\operatorname{argmin}} \left(\sum_{k=1}^m \sum_{l=1}^p \|Y_{kl} - y\|^2 w_{kl}(i, j) \right) = \sum_{k=1}^m \sum_{l=1}^p w_{kl}(i, j) Y_{kl}.$$

Für den Nadaraya-Watson-Schätzer wählen wir nun den zuvor eingeführten zweidimensionalen Kern $K_2(i, j) = K(i)K(j)$ für einen Designpunkt (i, j) und eindimensionalen Kern K . Damit erhalten wir folgende Gewichte:

$$w_{kl}(i, j) = \frac{K_2\left(\frac{k-i}{h}, \frac{l-j}{h}\right)}{\sum_{s=1}^m \sum_{t=1}^p K_2\left(\frac{s-i}{h}, \frac{t-j}{h}\right)} = \frac{K\left(\frac{k-i}{h}\right)K\left(\frac{l-j}{h}\right)}{\left(\sum_{s=1}^m K\left(\frac{s-i}{h}\right)\right)\left(\sum_{t=1}^p K\left(\frac{t-j}{h}\right)\right)}$$

für $(i, j), (k, l) \in \{1, \dots, m\} \times \{1, \dots, p\}$ und Bandweite $h > 0$. Da der zweidimensionale Kern K_2 als Produkt eines eindimensionalen Kerns K definiert ist, bedeutet dies für Kerne K , die symmetrisch um 0 monoton fallen (z.B. Gaußkern), dass Pixel, die diagonal zum entrauschenden Pixel liegen, weniger für die Entrauschung gewichtet werden als Pixel, die horizontal oder vertikal zum entrauschenden Pixel liegen, obwohl sie in beiden Fällen nur einen Pixel vom Zentrum entfernt sind. Für den auf $[-1, 1]$ konstanten Rechteckskern findet dagegen eine gleiche Gewichtung aller Pixel eines „angrenzenden Pixelringes“ statt, d.h. diagonale, horizontale und vertikale Abstände (gemessen in Pixeln) werden gleich gewichtet.

```
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)))
}
```

```
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)))

# Initialize 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)))
  }
}

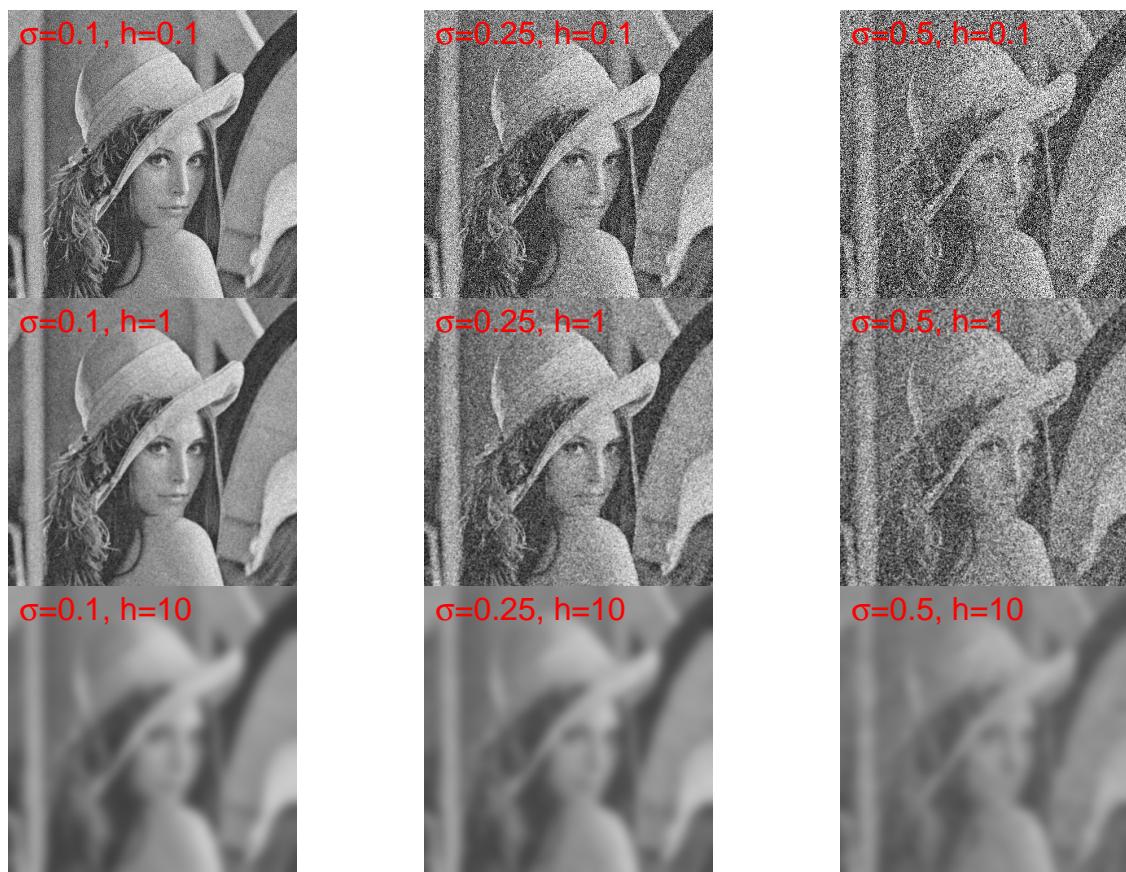
sigma <- c(0.1, 0.25, 0.5)
h <- c(0.1, 1, 10)

```

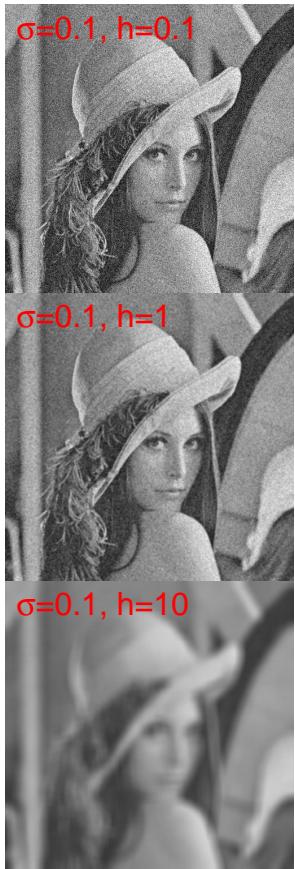
```
# Lena  
fnEvalNWGaussianNoise(imgLenaGray, sigma, h)
```



```
## 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
```

3.2 Weitere Methoden zur Bildentrauschung

3.2.1 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.

3.2.2 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.

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