

Projektaufgaben Block 2

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

06 Dezember 2016

1 Nichtparametrisches Testen

1.1 Zwillingstudie

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:

```
# 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
```

```
# 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.

1.2 t -Test vs. Wilcoxon-Vorzeichen-Test

```
fnTestPowerMC <- function(fnError, n = 30, theta = 0, 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
  #   theta:   True parameter
  #   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(theta + fnError(n*nSim, ...), ncol = n)

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

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

  # 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, theta = 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 the points x

# Get number of points and sample size
nPts <- length(x)
n <- length(X)

# Compute the estimated kernel density values for every bandwidth
kernels <- list()
for(i in 1:length(h)) {
  matKernel <- K(matrix(rep(X, nPts), ncol = n, byrow = TRUE) - x) / h[i])
  kernels <- append(kernels, list((1/(n*h[i])) * apply(matKernel, 1, sum)))
}

# If h is scalar return just a matrix with the kernels
# If h is a vector return a list with multiple kernels, their x and h values
if(length(h) == 1) {
  return(kernels[[1]])
} else {
  return(list(x = x, h = h, kernels = kernels))
}
}

```

```

fnKernelPlot <- function(listKernels, title = NULL, data = NULL) {
  # This function plots the kernel density estimates for a given kernel list
  #
  # Args:
  #   listKernels: list of kernels calculated with fnKernelDensityEst
  #   title: main title of plot
  #
  # Returns:
  #   -

  # Save number of kernels (each with a different bandwidth)
  nKernels <- length(listKernels$kernels)

  # Create palette depending on size of list
  cols <- rainbow(nKernels, alpha = 1)

  # Plot theoretical density of histogram (if data is available)
  if(is.null(data)) {
    plot(listKernels$x, dnorm(listKernels$x), type = "l",
        main = title, xlab = "x", ylab = "Density", ylim = c(0,0.6))
  } else {
    plot(density(data), main = title, xlab = "x",
        ylab = "Density", ylim = c(0,0.6))
  }

  # Print kernels
  for(i in 1:nKernels) {
    lines(listKernels$x, listKernels$kernels[[i]], col = cols[i])
  }

  # Create and add legend

```

```

dlegend <- ifelse(is.null(data), "True density", "R density()")
hlegend <- sapply(listKernels$h, function(x) {x <- paste("h = ", x)})
legend("topright", legend = c(dlegend, hlegend),
      col = c("black", 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 Daten

```

# Sample size and generate sample
set.seed(42)
n <- 30
X <- rnorm(n)

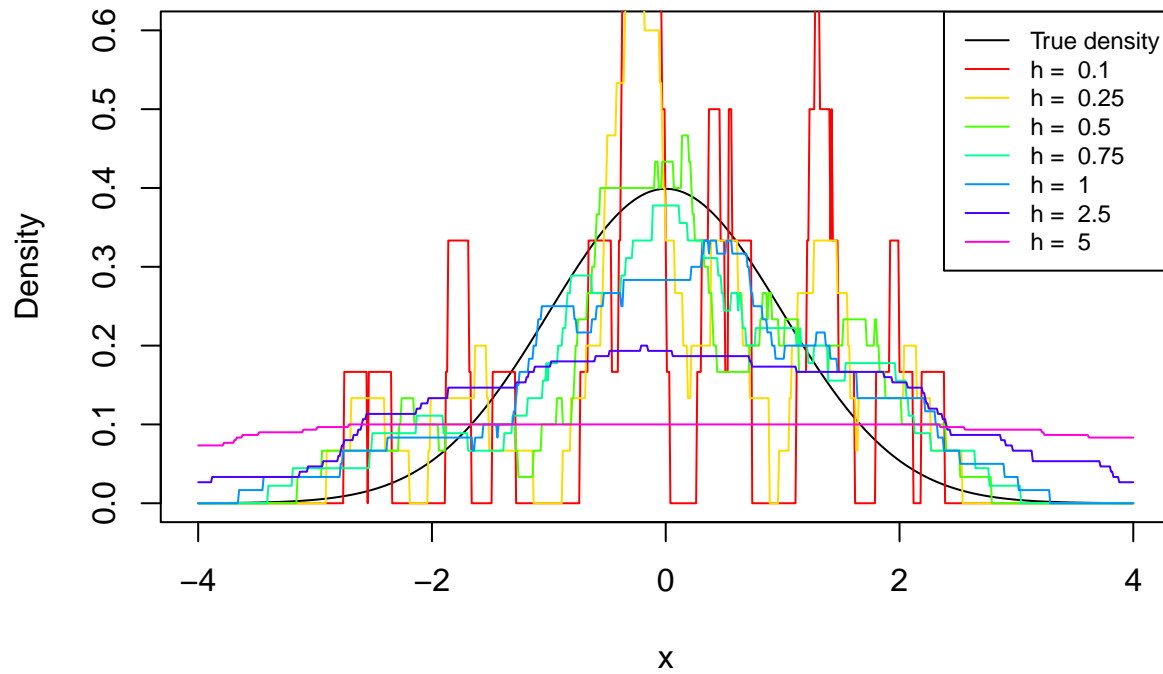
# Define points to estimate kernel density on
x <- seq(-4, 4, 0.01)

# Set different bandwidths
#h <- c(0.1, 1, 10)
h <- c(0.1, 0.25, 0.5, 0.75, 1, 2.5, 5)

# Rectangular kernel density estimation
fnKernelPlot(fnKernelDensityEst(x, X, K = fnRectangularKernel, h),
  title = "Rectangular Kernel Density Estimation")

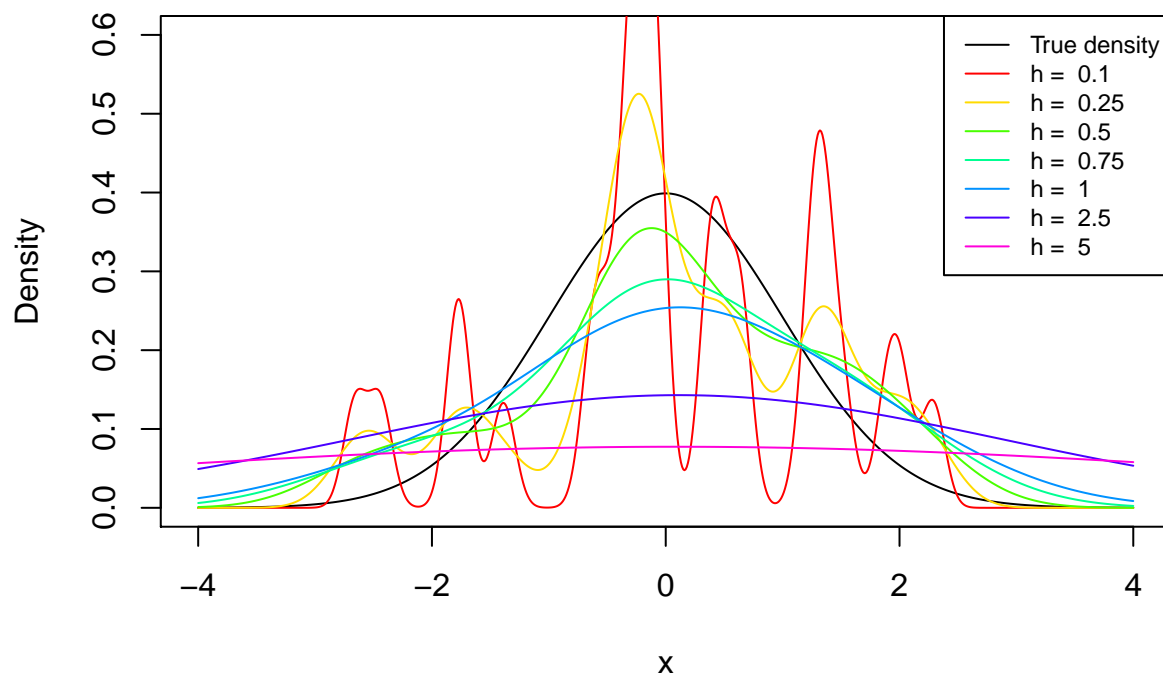
```

Rectangular Kernel Density Estimation



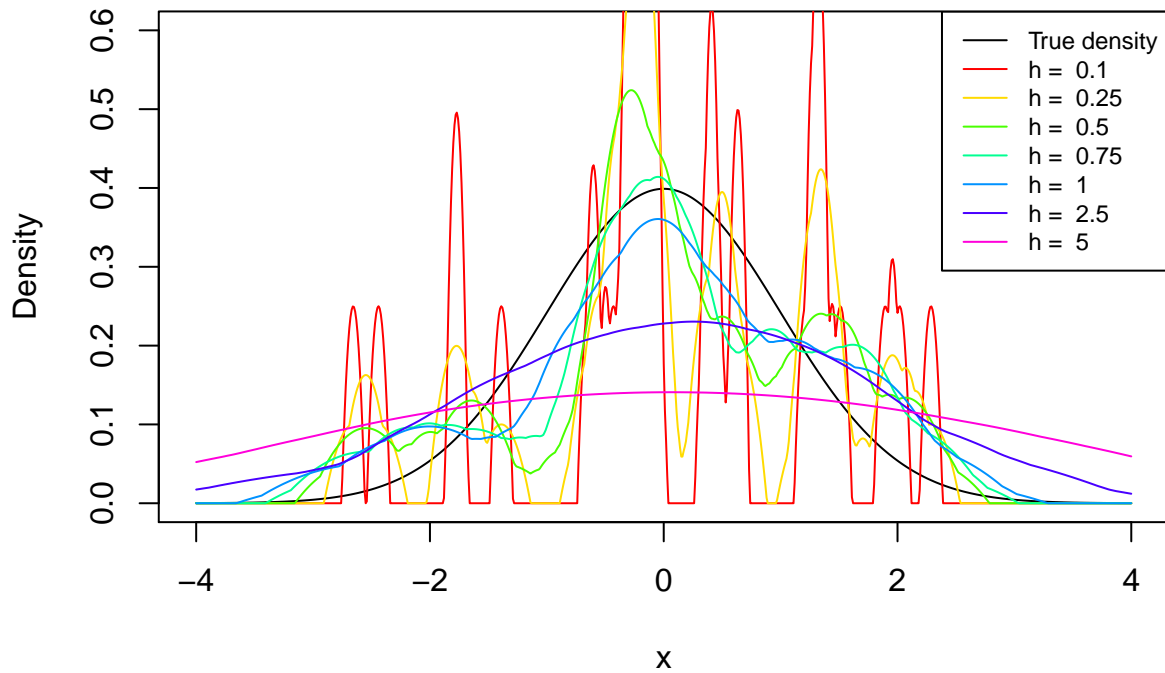
```
# Gaussian kernel density estimation
fnKernelPlot(fnKernelDensityEst(x, X, K = fnGaussianKernel, h),
             title = "Gaussian Kernel Density Estimation")
```

Gaussian Kernel Density Estimation



```
# Epanechnikov kernel density estimation
fnKernelPlot(fnKernelDensityEst(x, X, K = fnEpanechnikovKernel, h),
             title = "Epanechnikov Kernel Density Estimation")
```

Epanechnikov Kernel Density Estimation

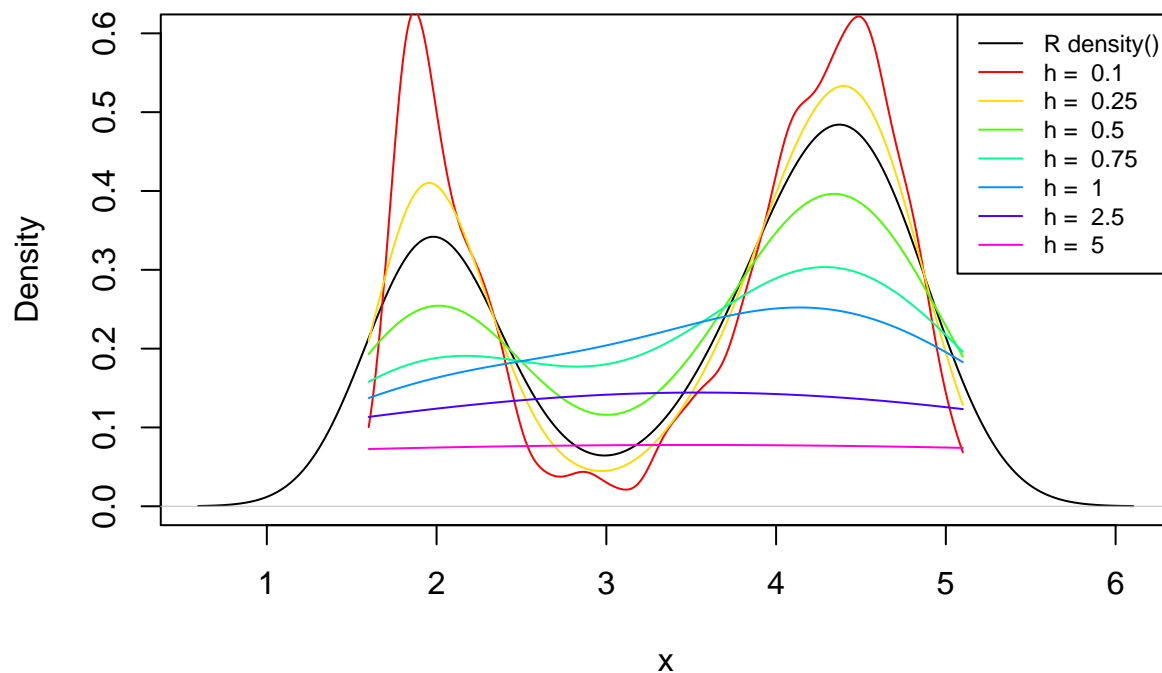


2.1.2 Anwendung auf faithful Datensatz

```
# Faithful kernel density estimation
data <- faithful$eruptions
x <- seq(min(data), max(data), 0.01)

fnKernelPlot(fnKernelDensityEst(x, data, K = fnGaussianKernel, h),
             title = "Gaussian Kernel Density Estimation for faithful eruption data",
             data = data)
```

Gaussian Kernel Density Estimation for faithful eruption data



2.2 Kreuzvalidierung zur Bandweitenwahl

```
fnG <- function(x, X, K, h) {  
  # This function computes the function G for given bandwidth  
  #  
  # 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:  
  #   Result of function G  
  
  # Get number of points and sample size  
  nPts <- length(x)  
  n <- length(X)  
  
  G <- 0  
  for(j in 1:nPts) {  
    fj <- 0  
    for(k in 1:n) {  
      if(k != j) {  
        fj <- fj + K((X[k] - X[j]) / h)  
      }  
    }  
    G <- G + fj/((n-1)*h)  
  }  
}
```



```

    }
    return(G/n)
  }

fnJ <- function(h) {
  # Calculates J for given bandwidth
  G <- fnG(x, data, fnGaussianKernel, h)
  f <- fnKernelDensityEst(x, data, fnGaussianKernel, h)
  return(sum(f^2) - (2*G))
}

vech <- c(seq(0.1,0.9,0.1), seq(1,10,1))

vecJ <- NULL
for(h in vech) {
  vecJ <- c(vecJ, fnJ(h))
}
vech[which.min(vecJ)]

```

```
## [1] 10
```

3 Bildentrauschen