Solution to Series 5

1. **a)** Read in and sort the data set for further analysis:

```
> diabetes <-
   read.table("http://stat.ethz.ch/Teaching/Datasets/diabetes2.dat",
              header = TRUE)
> reg <- diabetes[, c("Age", "C.Peptide")]
> names(reg) <-     c("x",   "y")
> reg <- reg[sort.list(reg$x), ]
```

We use a utility function for leave-one-out (LOO) cross-validation:

```
> ##' Calculates the LOO CV score for given data and regression prediction function
> ##'
> ##' @param reg.data: regression data; data.frame with columns 'x', 'y'
> ##' @param reg.fcn:  regr.prediction function; arguments:
> ##'                    reg.x: regression x-values
> ##'                    reg.y: regression y-values
> ##'                    x:     x-value(s) of evaluation point(s)
> ##'                  value: prediction at point(s) x
> ##' @return LOOCV score
> loocv <- function(reg.data, reg.fcn)
 {
   ## Help function to calculate leave-one-out regression values
   loo.reg.value <- function(i, reg.data, reg.fcn)
     return(reg.fcn(reg.data$x[-i], reg.data$y[-i], reg.data$x[i]))

   ## Calculate LOO regression values using the help function above
   n <- nrow(reg.data)
   loo.values <- sapply(1:n, loo.reg.value, reg.data, reg.fcn)

   ## Calculate and return MSE
   mean((reg.data$y - loo.values)^2)
 }
```

We first plot the data to guess a good bandwidth ($h = 4$; plot not shown here, it is the same as in Figure 3.1 of the lecture notes), then define a regression function that can be used with `loocv` defined above.

```
> plot(reg$x, reg$y)
> h <- 4
> reg.fcn.nw <- function(reg.x, reg.y, x)
   ksmooth(reg.x, reg.y, x.point = x, kernel = "normal", bandwidth = h)$y
> (cv.nw <- loocv(reg, reg.fcn.nw))
```

```
[1] 0.3905108
```

We calculate the hat matrix "manually" in order to calculate the degrees of freedom; this is the smoothing parameter used for other regression estimators:

```
> n <- nrow(reg)
> Id <- diag(n)
> S.nw <- matrix(0, n, n)
> for (j in 1:n)
   S.nw[, j] <- reg.fcn.nw(reg$x, Id[, j], reg$x)
> (df.nw <- sum(diag(S.nw)))
```

```
[1] 4.45845
```

```
>
```

We also do the calculation of the CV value with the hat matrix:

```
> y.fit.nw <- reg.fcn.nw(reg$x, reg$y, reg$x)
> (cv.nw.hat <- mean(((reg$y - y.fit.nw)/(1 - diag(S.nw)))^2))
```

```
[1] 0.3905108
```

Moreover, we can also simply use `hatMat` from the package `sfsmisc`:

```
> library(sfsmisc)
> #degrees of freedom
> hatMat(reg$x,trace=TRUE,pred.sm=reg.fcn.nw,x=reg$x)
```

```
[1] 4.45845
```

```
> #CV value
> S.nw.hatMat <- hatMat(reg$x,trace=FALSE,pred.sm=reg.fcn.nw,x=reg$x)
> (cv.nw.hatMat <- mean(((reg$y - y.fit.nw)/(1 - diag(S.nw.hatMat)))^2))
```

```
[1] 0.3905108
```

b) Local polynomial ("lp") regression from `loess`:

```
> reg.fcn.lp <- function(reg.x, reg.y, x) {
    lp.reg <- loess(reg.y ~ reg.x, enp.target = df.nw, surface = "direct")
    predict(lp.reg, x)
  }
> (cv.lp <- loocv(reg, reg.fcn.lp))
```

```
[1] 0.3849359
```

Again, we also calculate the CV value with the hat matrix constructed "manually":

```
> n <- nrow(reg)
> Id <- diag(n)
> S.lp <- matrix(0, n, n)
> for (j in 1:n)
    S.lp[, j] <- reg.fcn.lp(reg$x, Id[, j], reg$x)
> y.fit.lp <- reg.fcn.lp(reg$x, reg$y, reg$x)
> (cv.lp.hat <- mean(((reg$y - y.fit.lp)/(1 - diag(S.lp)))^2))
```

```
[1] 0.3849359
```

And once more, we also compute the CV value using `hatMat`:

```
> S.lp.hatMat <- hatMat(reg$x,trace=FALSE,pred.sm=reg.fcn.lp,x=reg$x)
> (cv.lp.hatMat <- mean(((reg$y - y.fit.lp)/(1 - diag(S.lp.hatMat)))^2))
```

```
[1] 0.3849359
```

Note that for both the kernel and the local polynomial regression, the alternative calculation using the hat matrix also gives the right result here. However, it is not known whether this is always the case for kernel or local polynomial regression.

c) Smoothing spline ("ss") regression from `smooth.spline` with fixed degrees of freedom. We begin by looking at the internally calculated CV value:

```
> est.ss <- smooth.spline(reg$x, reg$y, cv = TRUE, df = df.nw)
> est.ss$cv.crit
```

```
[1] 0.3886042
```

We then use the same smoothing parameter `spar` for our own calculations of the CV value:

```
> reg.fcn.ss <- function(reg.x, reg.y, x)
  {
    ss.reg <- smooth.spline(reg.x, reg.y, spar = est.ss$spar)
    predict(ss.reg, x)$y
  }
> (cv.ss <- loocv(reg, reg.fcn.ss))
```

```
[1] 0.3880219
```

Alternative calculation using the hat-matrix computed "manually":

```
> n <- nrow(reg)
> Id <- diag(n)
> S.ss <- matrix(0, n, n)
> for (j in 1:n)
   S.ss[, j] <- reg.fcn.ss(reg$x, Id[, j], reg$x)
> y.fit.ss <- reg.fcn.ss(reg$x, reg$y, reg$x)
> (cv.ss.hat <- mean(((reg$y - y.fit.ss)/(1 - diag(S.ss)))^2))
[1] 0.3886042
```

And alternative calculation using `hatMat`:

```
> S.ss.hatMat <- hatMat(reg$x,trace=FALSE,pred.sm=reg.fcn.ss,x=reg$x)
> (cv.ss.hatMat <- mean(((reg$y - y.fit.ss)/(1 - diag(S.ss.hatMat)))^2))
[1] 0.3886042
```

Note that there is a slight discrepancy between the CV score computed using `loocv` and the rest of the CV scores. In theory, all these values should be identical. However, the difference is caused by the way in which the degrees of freedoms are specified in `smooth.spline`, i.e., the `spar` parameter is internally converted to the λ value and this conversion depends on the data. In each run of the CV (in `loocv`), the training data is slightly different, which means that a different λ is used each time.

d) Smoothing spline regression with optimized degrees of freedom:

```
> cv.ssopt<-smooth.spline(reg$x, reg$y, cv = TRUE)$cv.crit
> (cv.ssopt)
[1] 0.3828729
```

e) Constant fit ("CF"):

```
> reg.fcn.cf <- function(reg.x, reg.y, x) mean(reg.y)
> (cv.cf <- loocv(reg, reg.fcn.cf))
[1] 0.531576
```

f)

```
> sort(c(nw=cv.nw,lp=cv.lp,ss=cv.ss,ssopt=cv.ssopt,cf=cv.cf))
    ssopt        lp        ss        nw        cf
0.3828729 0.3849359 0.3880219 0.3905108 0.5315760
```

The optimized smoothing splines method achieves the best score. However, we should remark that if the quality of a method is judged by cross-validation, the cross-validation mimics the error which the method is expected to produce on new, independent data. We leave out a point and apply the method independent of this point to predict its response. This is no longer true, if the method includes an optimal choice of a parameter by optimization of the cross-validation score (as method no. 4 does), because then the outcome depends on *all* cross-validations, and therefore it is no longer independent on the point left out at the moment. Thus it can be expected, that the resulting cross-validation score is over-optimistic. For this reason we would conclude that local polynomial regression is most adequate in this task although differences to smoothing splines and kernel regression are small. Our constant fit, however, performed worst.

2. **a)** Approximate the true value

```
> set.seed(3)
> (true.par <- mean(rgamma(100000000, shape = 2, rate = 2), trim = 0.1))
[1] 0.9103737
```

b)

```
> set.seed(1)
> sample40 <- rgamma(n = 40, shape = 2, rate = 2)
> mean(sample40, trim = 0.1)
[1] 0.8824166
```

c)

```
> require("boot")
> tm <- function(x, ind) {mean(x[ind], trim = 0.1)}
> res.boot <- boot(data = sample40, statistic = tm, R = 10000,
                   sim = "ordinary")
> boot.ci(res.boot, conf = 0.95, type = c("basic", "norm", "perc"))
```

```
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 10000 bootstrap replicates

CALL :
boot.ci(boot.out = res.boot, conf = 0.95, type = c("basic", "norm",
    "perc"))

Intervals :
Level      Normal              Basic              Percentile
95%   ( 0.7008,  1.0542 )   ( 0.6972,  1.0512 )   ( 0.7136,  1.0677 )
Calculations and Intervals on Original Scale
```

d) We first define two functions before running the simulation.

```
> ##' Checks if a confidence interval contains the true parameter (separately
> ##' for the lower and the upper end)
> ##'
> ##' @param ci: Output of the function boot.ci which contains CIs
> ##' @param ty: Type of confidence interval
> ##' @param true.par: True parameter
> ##'
> ##' @return Vector with two elements where first one corresponds to the lower
> ##'         end and the second to the upper end of the confidence interval.
> ##'         If the CI is [CI_l, CI_u], the first element is 1 if theta < CI_l
> ##'         and 0 otherwise. The second element is 1 if theta > CI_u and 0
> ##'         otherwise.
> check_ci <- function(ci, ty, true.par) {
   # Get confidence interval of type ty from object ci
   lower.upper <- switch (ty,
     "norm" = ci[["normal"]][2:3],
     "perc" = ci[["percent"]][4:5],
     "basic" = ci[["basic"]][4:5]
   )

   res <- if (true.par < lower.upper[1]) {
       c(1, 0)
     } else if (true.par > lower.upper[2]) {
       c(0, 1)
     } else {
       c(0, 0)
     }
   names(res) <- c("lower", "upper")

   return(res)
 }
> ##' Runs one simulation run, i.e. creates new data set, calculates bootstrap
> ##' CIs, and checks if true parameter is contained.
> ##'
> ##' @param n: Size of sample
> ##' @param true.par: True parameter
> ##' @param R: Number of bootstrap replicates
> ##' @param type: Type of bootstrap CIs, see function boot.ci
> ##'
> ##' @return A vector containing the result of the function check_ci for each
> ##'         of the confidence intervals
> do_sim <- function(n, true.par, R = 1000,
                     type = c("basic", "norm", "perc")) {
   # Generate the data
   x <- rgamma(n = n, shape = 2, rate = 2)
   # Construct the CIs for the trimmed mean
   res.boot <- boot(data = x, statistic = tm, R = R, sim = "ordinary",
```

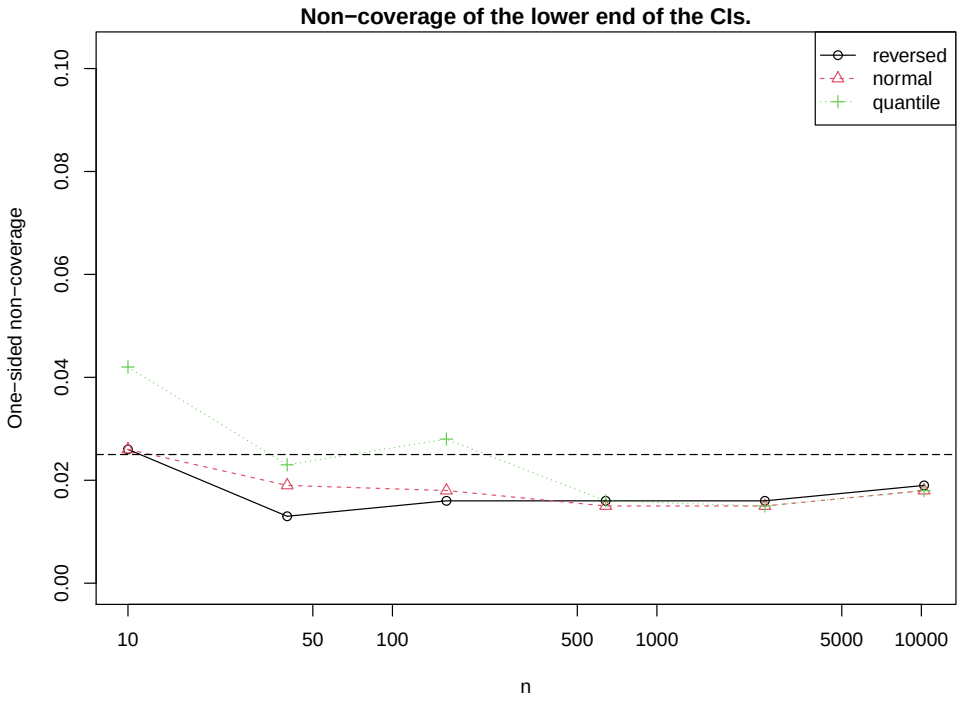
```
                    parallel = "multicore", ncpus = 20)
  res.ci <- boot.ci(res.boot, conf = 0.95, type = type)

  # Check if CIs contain true.par
  res <- vector(mode = "integer", length = 0)
  for (ty in type) {
    res <- c(res, check_ci(ci = res.ci, ty = ty, true.par = true.par))
    names(res)[(length(res) - 1):length(res)] <-
      paste(c(ty, ty), c("lower", "upper"), sep = "_") #add names in the format
                                                       #'type_lower' and 'type_upper'
  }
  # Alternatively, one could use a function of the apply family, e.g. sapply.

  return(res)
}


> ############################
> ### Run simulation     ###
> ############################
> set.seed(22)
> require("boot")
> sample.size <- c(10, 40, 160, 640, 2560, 10240)
> n.sim <- 1000
> type <- c("basic", "norm", "perc")
> # The object RES stores the results, i.e. each row corresponds
> # to the non-coverage rate for the lower and upper ends of the
> # confidence intervals, i.e. the percentage of times that theta < CI_l
> # and the percentage of times that theta > CI_u, if the CI is
> # denoted by (CI_l, CI_u). The last column of RES corresponds to
> # the number of observations.
> RES <- matrix(NA, nrow = length(sample.size), ncol = length(type) * 2 + 1)
> colnames(RES) <- c(paste(rep(type, each = 2),
                           rep(c("lower", "upper"), times = length(type)), sep = "_"),
                     "n")
> for (j in 1:length(sample.size)) {
  n <- sample.size[j]
  # The object res.sim stores the results, i.e. each row corresponds
  # to the output of the function do_sim. This means that each row contains 0
  # and 1 encoding whether the true parameter was inside the CI or outside.
  # Also see the function check_ci.
  res.sim <- matrix(NA, nrow = n.sim, ncol = length(type) * 2)
  for (i in 1:n.sim) {
    # Compute CIs and check if true.par is contained
    res.sim[i, ] <- do_sim(n = n, true.par = true.par, type = type, R = 2000)
  }
  # Compute the upper and lower non-coverage rate
  RES[j, ] <- c(apply(res.sim, 2, mean), n)
}
```

Note that the above code runs in parallel on 20 cores. We chose larger values for some parameters than you were asked to do on the exercise sheet, i.e., `R = 2000`, `n.sim = 1000`, and `sample.size = c(10, 40, 160, 640, 2560, 10240)`.

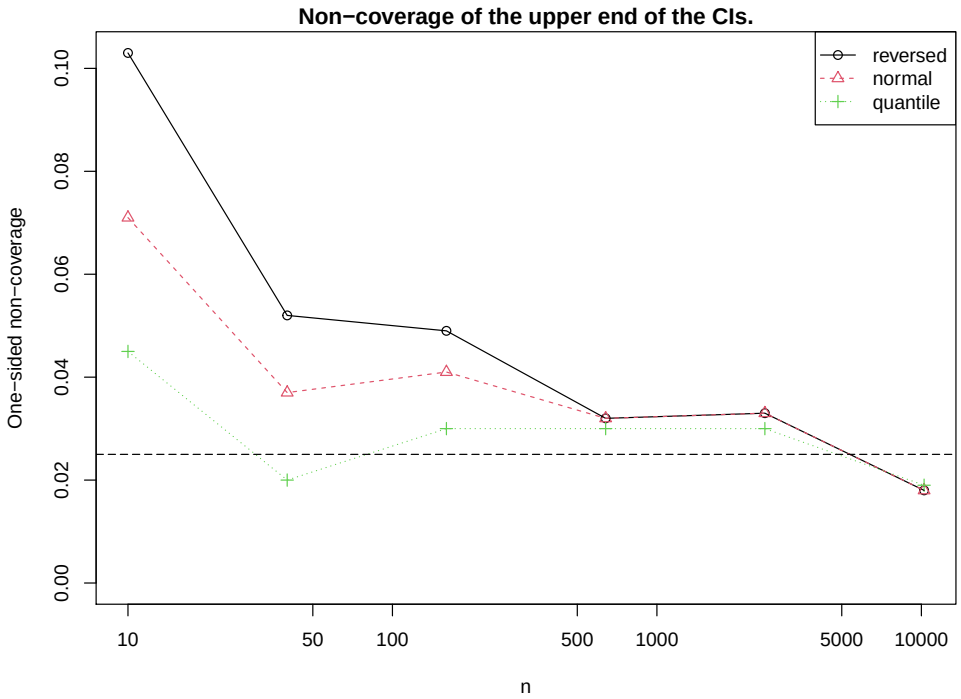
The plots have the same limits on the y-axis and we use log-scale for the x-axis.

```
> y.lim <- max(RES[, -ncol(RES)])
> # Plot of lower non-coverage
> plot(basic_lower ~ n, data = RES, col = 1, pch = 1, ylim = c(0, y.lim),
       log = "x", ylab = "One-sided non-coverage",
       main = "Non-coverage of the lower end of the CIs.")
> points(norm_lower ~ n, data = RES, col = 2, pch = 2, xlog = TRUE)
```

```
> points(perc_lower ~ n, data = RES, col = 3, pch = 3, xlog = TRUE)
> lines(basic_lower ~ n, data = RES, col = 1, lty = 1, xlog = TRUE)
> lines(norm_lower ~ n, data = RES, col = 2, lty = 2, xlog = TRUE)
> lines(perc_lower ~ n, data = RES, col = 3, lty = 3, xlog = TRUE)
> abline(h = 0.025, lty = 5)
> legend("topright", legend = c("reversed", "normal", "quantile"),
       pch = 1:3, lty = 1:3, col = 1:3)
```




```
> # Plot of upper non-coverage
> plot(basic_upper ~ n, data = RES, col = 1, pch = 1, ylim = c(0, y.lim),
     log = "x", ylab = "One-sided non-coverage",
     main = "Non-coverage of the upper end of the CIs.")
> points(norm_upper ~ n, data = RES, col = 2, pch = 2, xlog = TRUE)
> points(perc_upper ~ n, data = RES, col = 3, pch = 3, xlog = TRUE)
> lines(basic_upper ~ n, data = RES, col = 1, lty = 1, xlog = TRUE)
> lines(norm_upper ~ n, data = RES, col = 2, lty = 2, xlog = TRUE)
> lines(perc_upper ~ n, data = RES, col = 3, lty = 3, xlog = TRUE)
> abline(h = 0.025, lty = 5)
> legend("topright", legend = c("reversed", "normal", "quantile"),
       pch = 1:3, lty = 1:3, col = 1:3)
```



In this setting, the reversed bootstrap CI and the normal approximation CI are biased in the sense that they estimate a too small upper end of the CI for small sample sizes. There are only small differences between the CIs for large sample sizes.