Stat 435 lecture notes 6: coding

FDR and differential gene expression study

Define the BH procedure:

```r
BH <- function(Pvals, FDRlevel) {
    if (any(is.na(Pvals)) | any(is.nan(Pvals))) {
        cat("^^NA or NAN in pvalues... ", "\n")
        print(Pvals[is.na(Pvals)])
    }

    if (any(is.na(FDRlevel)) | any(is.nan(FDRlevel)))
        cat("^^NA or NAN FDRlevel... ", "\n")

    if (is.vector(Pvals))
        lgh3 <- length(Pvals)
    if (is.matrix(Pvals) | is.data.frame(Pvals))
        lgh3 <- dim(Pvals)[1]
    PvalAndIdx <- cbind(Pvals, seq(1:lgh3))
    PvalAndIdxOrd <- PvalAndIdx[order(PvalAndIdx[, 1]), ]

    # cat('^^Dims of PvalAndIdxOrd
    # is:',as.vector(dim(PvalAndIdxOrd)),'\n')

    BHstepups <- seq(1:lgh3) * (FDRlevel/lgh3)
    cmp3 <- PvalAndIdxOrd[, 1] <= BHstepups
    scmp3 <- sum(cmp3)

    # collect rejections if any
    if (scmp3 == 0) {
        print("No rejections made by BH procedure")  # when there are no rejections
        rejAndTresh <- list(matrix(numeric(0), ncol = 2, nrow = 0),
            0)
    } else {
        r <- max(which(cmp3))
        # cat('^^^ Minimax index in BH is:',r,'\n') cat('^^^ Minimax
        # threshold in BH is:',BHstepups[r],'\n')
        if (r == 1) {
            # when r =1, a row is chosen, so should change it into a
            # matrix of two columns
            BHrej = as.matrix(t(PvalAndIdxOrd[1:r, ]))
            # print(BHrej)
        } else {
            BHrej <- PvalAndIdxOrd[1:r, ]
        }
        rejAndTresh <- list(BHrej, BHstepups[r])
    }
    return(rejAndTresh)
}
```

Generate data

```
m = 200
n = 10
m0 = 150
set.seed(123)
miuAll = runif(m, -2, 2)
miuEqual = miuAll[1:m0]
miuUnequal = miuAll[(m0 + 1):m] + runif(m - m0, 0.5, 2)

miuGroup2 = c(miuEqual, miuUnequal)

Data1 = matrix(0, nrow = m, ncol = n)
Data2 = Data1
for (i in 1:m) {
    Data1[i, ] = rnorm(n, mean = miuAll[i], sd = 1)
    Data2[i, ] = rnorm(n, mean = miuGroup2[i], sd = 1)
}

Data = cbind(Data1, Data2)

# for each test, obtain p-value
pVec = double(m)
for (i in 1:m) {
    # two-sample t-test with equal variance; two-sided p-value
    pVec[i] = t.test(Data1[i, ], Data2[i, ], alternative = "two.sided",
        var.equal = TRUE)$p.value
}

# apply BH
Res = BH(pVec, 0.05)
Res
```

```
## [[1]]
##              Pvals
##  [1,] 2.212089e-05 153
##  [2,] 3.413001e-05 152
##  [3,] 4.251729e-05 179
##  [4,] 5.190520e-05 181
##  [5,] 6.854481e-05 196
##  [6,] 1.417621e-04 174
##  [7,] 1.481867e-04 170
##  [8,] 1.513725e-04 198
##  [9,] 2.019002e-04 200
## [10,] 5.194775e-04 192
## [11,] 1.860628e-03 164
## [12,] 2.152617e-03 194
## [13,] 2.567441e-03 180
## [14,] 3.460722e-03 197
##
## [[2]]
## [1] 0.0035
```

FDR and LASSO

```
# start of codes
set.seed(123)
p = 100
s = 20
beta1 = rep(0, p - s)
y1 = rnorm(p - s)
beta2 = runif(s, 0.5, 2)
y2 = beta2 + rnorm(s)
y = c(y1, y2)
beta = c(beta1, beta2)
x = diag(p)
simData = data.frame(cbind(y, x))
## end of codes


## fitting a model

library(glmnet)
grid = 10^seq(10, -2, length = 100)

set.seed(1)
train = sample(1:nrow(x), nrow(x)/2)
test = (-train)
y.test = y[test]


predict.regsubsets = function(object, newdata, id, ...) {
    form = as.formula(object$call[[2]])
    mat = model.matrix(form, newdata)
    coefi = coef(object, id = id)
    mat[, names(coefi)] %*% coefi
}
```

```r
# The Lasso

lasso.mod = glmnet(x[train, ], y[train], alpha = 1, lambda = grid)
cv.out = cv.glmnet(x[train, ], y[train], alpha = 1)

bestlam = cv.out$lambda.min

# fit model on whole data
out = glmnet(x, y, alpha = 1, lambda = grid)
lasso.coef = predict(out, type = "coefficients", s = bestlam)[1:101,
    ]
lasso.coef[lasso.coef != 0]
```

```
## (Intercept)         V87
##   0.2741084   0.2584121
```

```r
which(lasso.coef != 0)
```

```
## (Intercept)         V87
##           1          88
```

```r
beta[87]
```

```
## [1] 1.610502
```

De-sparsified lasso

Compute p-values based on the lasso projection method, also known as the de-sparsified Lasso, using an asymptotic gaussian approximation to the distribution of the estimator.

```r
library(hdi)

outRes = lasso.proj(x, y, family = "gaussian", standardize = TRUE,
    multiplecorr.method = "holm", N = 10000, parallel = FALSE,
    ncores = getOption("mc.cores", 2L), betainit = "cv lasso",
    sigma = NULL, Z = NULL, verbose = FALSE, return.Z = FALSE,
    suppress.grouptesting = FALSE, robust = FALSE, do.ZnZ = FALSE)
```

```
## Nodewise regressions will be computed as no argument Z was provided.

## You can store Z to avoid the majority of the computation next time around.

## Z only depends on the design matrix x.
```

```r
outRes$pval[1:10]
```

```
##  [1] 0.4306208 0.6332345 0.2274631 0.8460924 0.8896317 0.1756989 0.8623115
##  [8] 0.1466565 0.3643517 0.4964782
```

knockoff

Apply the knockoff method with fixed-X knockoff variables

```
library(knockoff)

p = 100
n = 200
k = 15
X = matrix(rnorm(n * p), n)
nonzero = sample(p, k)
beta = 5.5 * (1:p %in% nonzero)

which(beta != 0)
```

```
##  [1] 13 14 15 16 23 32 37 39 62 64 71 83 84 96 97
```

```
y = X %*% beta + rnorm(n)
# Basic usage with default arguments
result = knockoff.filter(X, y, knockoffs = create.fixed)
print(result$selected)
```

```
## integer(0)
```

```
# Advanced usage with custom arguments
knockoffs = function(X) create.fixed(X, method = "equi")
result = knockoff.filter(X, y, knockoffs = knockoffs)
print(result$selected)
```

```
##  [1] 13 14 15 16 23 32 37 39 62 64 71 83 84 93 96 97
```