Supplementary Material for

Biclustering Models for Structured Microarray Data

Turner, H.L., Bailey, T.C., Krzanowski, W.J. and Hemingway, C.A.

IEEE/ACM Trans. Comp. Biol. Bioinf., 2005, 2(4), 316-329

This supplement provides the R[1] code that was used to produce the analyses presented in the associated paper.

The code includes two user functions, `plaid()` and `summary()`, to fit plaid models in R and summarize the results. The remaining code is for a number of internal functions called by `plaid()`, to find starting values, fit individual biclusters, perform back-fitting, etc. The code has been annotated to assist the user and the final page gives some examples of how the code can be used to fit models of the type presented in the paper.

[1] `http://www.r-project.org/`

```
## R 2.1.0 (http://www.r-project.org/)
## Heather Turner
## 27/04/2005

##### USER FUNCTIONS #####

## 'plaid' to fit plaid model
## 'summary' to print summary of fitted plaid model

## Define plaid result object and corresponding summary method
setClass("PlaidResult", representation("list"))

setGeneric("summary")

setMethod("summary","PlaidResult",
function(object)
{
    nr <- apply(object$rows, 2, sum)
    nk <- apply(object$cols, 2, sum)
    SS <- sapply(object$fits, function(x) sum(x^2))
    print(matrix(c(nr, nk, object$layer.df, round(SS, 2),
                   round(SS/object$layer.df, 2), object$convergence,
                   object$rows.released, object$cols.released), ncol = 8,
                 dim = list(Layer = 1:length(nr) - object$background,
                 c("Rows", "Cols", "Df", "SS", "MS", "Convergence",
                   "Rows Released", "Cols Released"))), digits = 15)
})

## Plaid model wrapper function - calls update function to fit layer
plaid <- function(
Z,                            # array (matrix) to be clustered
row.classes = NULL,           # optional class factor for rows (variable)
col.classes = NULL,           # optional class factor for columns (variable)
cluster = "b",                # "r", "c" or "b",  to cluster rows, columns or both
fit.model = y ~ m + a + b,    # model to fit to each layer (formula)
search.model = NULL,          # optional model to base search on (formula)
background = TRUE,            # logical  - whether or not to fit a background layer
row.release = NULL,           # row release criterion (scalar in range [0, 1])
col.release = NULL,           # column release criterion (scalar in range [0, 1])
shuffle = 3,                  # no. of permuted layers to use in permutation test
back.fit = 0,                 # no. of times to back fit after each layer
max.layers = 20,              # max no. of layers to include in the model
fix.layers = NULL,            # fixed no. of layers to include in the model
start.method = "convert",     # or "average" for starting values in supervised/3-way
iter.startup = NULL,          # no. of iterations to find starting values
iter.layer = NULL,            # no. of iterations to find a layer
iter.supervised = NULL,       # optional no. of supervised iterations
verbose = TRUE)               # if "TRUE", prints extra information on progress
{
    if (is.null(iter.startup) | is.null(iter.layer))
        stop(message = "Please provide values for iter.startup and iter.layer.")
    if ((!is.null(row.classes) | !is.null(col.classes))
        & is.null(iter.supervised))
        stop(message = "Please provide value for iter.supervised.")
    ## sort out the input Z
    Z <- unname(Z)
    if (length(dim(Z)) == 2) Z <- array(Z, c(dim(Z), 1))
    n <- dim(Z)[1]
    p <- dim(Z)[2]
```

```
t <- dim(Z)[3]
## make sure supervisory settings are consistent
if (is.null(row.classes) & is.null(col.classes)) iter.supervised <- 0
if (!is.null(row.classes))
{
    if (length(row.classes) != n)
        stop(message =
             "Length of row.classes must equal the number of rows")
    ## ensure classes encoded by consecutive numbers
    row.classes <- as.vector(unclass(as.factor(row.classes)))
    row.grouped <- tabulate(row.classes)
}
if (!is.null(col.classes))
{
    if (length(col.classes) != p)
        stop(message =
             "Length of col.classes must equal the number of columns")
    ## ensure classes encoded by consecutive numbers
    col.classes <- as.vector(unclass(as.factor(col.classes)))
    col.grouped <- tabulate(col.classes)
}
## sort out which models are being used
fit.model <- labels(terms(fit.model))
if (is.null(search.model)) search.model <- fit.model
else search.model <- labels(terms(search.model))
## check requirements on number of layers
if (!is.null(fix.layers)) shuffle <- 0
length <- ifelse(!is.null(fix.layers), fix.layers, max.layers) + background
## set up objects to hold results
SS <- layer.df <- status <- rows.released <- cols.released <-
    rep(NA, length)
r <- matrix(c(rep(TRUE, n), rep(FALSE, n * (length - 1))), nrow = n)
k <- matrix(c(rep(TRUE, p), rep(FALSE, p * (length - 1))), nrow = p)
fits <- vector(mode = "list", length = length)
## START FITTING MODEL
## background layer
if (background)
{
    fits[[1]] <- fitLayer(Z, r[,1], k[,1], fit.model)
    Z <- Z - fits[[1]]
    SS[1] <- sum(fits[[1]]^2)
    layer.df[1] <- 1 + is.element("a", fit.model) * (n - 1) +
        is.element("b", fit.model) * (p - 1) +
            is.element("c", fit.model) * (t - 1)
    if (verbose == TRUE) cat("layer: 0 \n ", SS[[1]], "\n", sep = "")
}
layer <- as.numeric(background)
## bicluster layers
while (layer < min(fix.layers, max.layers, na.rm = TRUE) + background)
{
    if (verbose == TRUE) cat("layer:", layer, "\n")
    u <- updatePlaid(Z, n, p, t, row.classes, row.grouped, col.classes,
                     col.grouped, cluster, fit.model, search.model,
                     revised.consistency, row.release, col.release,
                     shuffle, start.method, iter.startup, iter.layer,
                     iter.supervised, verbose)
    ## stop if no cluster found
    if (u[[1]] == 0) break
    ## otherwise extract results and calculate new residual matrix
```

```
        layer <- layer + 1
        distributeList(u, ind = layer, margin = 2)
        Z[r[,layer], k[,layer],] <- Z[r[,layer], k[,layer], , drop = FALSE] -
            fits[[layer]]
        ## back fit if desired
        if (back.fit > 0 & layer > 1)
        {
            if (verbose == TRUE) cat("back fitting", back.fit, "times\n")
            distributeList(backFit(layer, back.fit, fits, r, k, Z, fit.model))
        }
    }
    ## Create table summarising results - as in "summary" method
    if (!is.null(fix.layers)) layer <- fix.layers + background
    if (layer == background) print("No clusters have been found")
    else
    {
        nr <- colSums(r[,1:layer, drop = FALSE])
        nk <- colSums(k[,1:layer, drop = FALSE])
        print(matrix(c(nr, nk, layer.df[1:layer], round(SS[1:layer], 2),
                       round(SS[1:layer]/layer.df[1:layer], 2), status[1:layer],
                       rows.released[1:layer], cols.released[1:layer]),
                     ncol = 8, dim = list(Layer = 1:layer - background,
                                c("Rows", "Cols", "Df", "SS", "MS",
                                  "Convergence", "Rows Released",
                                  "Cols Released"))), digits = 15)
    }
    if (layer > background)
        new("PlaidResult",
            list(residuals = drop(Z),fits = lapply(fits[1:layer], drop),
                 layer.df = layer.df[1:layer], rows = r[,1:layer],
                 cols = k[,1:layer], convergence = status[1:layer],
                 rows.released = rows.released[1:layer],
                 cols.released = cols.released[1:layer],
                 background = background))
}

##### INTERNAL FUNCTIONS #####

## Function to find k means starting values for rows (cols if transpose = TRUE)
kmeansStart <- function(Z, iter.startup, transpose = FALSE)
{
    ## split rows into two groups, catch error if kmeans fails
    old.options <- options("error")
    options(error = NULL)
    if (transpose) km <- try(kmeans(t(Z), 2, iter.startup), TRUE)
    else km <- try(kmeans(Z, 2, iter.startup), TRUE)
    options(old.options)
    if (class(km) == "try-error") x <- rep(FALSE, dim(Z)[1 + transpose])
    ## if kmeans successful, use smaller cluster for starting rows (columns)
    else
    {
        if (km$size[[1]] < km$size[[2]])
            x <- ifelse(km$cluster == 1, TRUE, FALSE)
        else
            x <- ifelse(km$cluster == 2, TRUE, FALSE)
    }
    x
}
```

```
## Function to find k means starting values averaged over supervisory classes
kmeansClassStart <- function(Z, classes, iter.startup, transpose = FALSE)
{
    if (length(unique(classes)) > 2)
    {
        ## average over classes
        if (transpose) average.Z <- rowsum(t(Z), classes)/tabulate(classes)
        else average.Z <- rowsum(Z, classes)/tabulate(classes)
        ## proceed as if single rows/columns
        old.options <- options("error")
        options(error = NULL)
        km <- kmeans(average.Z, 2, iter.startup)
        options(old.options)
        if (class(km) == "try-error") x <- rep(FALSE, dim(Z)[1 + transpose])
        else
        {
            names(km$cluster) <- dimnames(average.Z)[[1]]
            if (km$size[[1]] < km$size[[2]])
                x <- ifelse(km$cluster == 1, TRUE, FALSE)[classes]
            else
                x <- ifelse(km$cluster == 2, TRUE, FALSE)[classes]
        }
    }
    ##If only two classes, start with smaller class
    else x <- (order(tabulate(classes)) == 1)[classes]
    x
}

## Function to fit single bicluster
updatePlaid <- function(Z, n, p, t, row.classes, row.grouped, col.classes,
                        col.grouped, cluster, fit.model, search.model,
                        revised.consistency, row.release, col.release, shuffle,
                        start.method, iter.startup, iter.layer, iter.supervised,
                        verbose)
{
    ## set number of release iterations equal to number of layer iterations
    if (!is.null(row.release) | !is.null(col.release))
        extra <- round(iter.layer/2) * 2
    else extra <- 0
    ## set up objects required
    cluster.SS <- numeric(length = shuffle + 1)
    status <- 0
    i <- 1
    for (i in 1:(shuffle + 1))
    {
        a <- numeric(n)
        b <- numeric(p)
        c <- numeric(t)
        r.check <- k.check <- list(1, 1)
        model <- search.model
        if (i > 1)
        {
            ## permute genes and samples, within each time point separately
            Z <- array(apply(Z, 3, function(x) x[sample(1:(n * p))]), dim(Z))
        }
        if (is.element(cluster, c("r", "b")))
        {
            ## get starting values for rows
            if (start.method == "average")
```

```
    {
        if (is.null(row.classes))
            r <- kmeansStart(rowMeans(Z, dims = 2), iter.startup)
        else
            r <- kmeansClassStart(rowMeans(Z, dims = 2), row.classes,
                                  iter.startup)
    }
    if (start.method == "convert")
    {
        r <- rowMeans(apply(Z, 3, kmeansStart, iter.startup)) >= 0.5
        if (!is.null(row.classes))
        {
            temp.r <- (tabulate(r * row.classes,
                                nbin = max(row.classes)) >=
                       0.5 * row.grouped)[row.classes]
            if (sum(temp.r) != 0) r <- temp.r
            else
            {
                row.classes <- NULL
                if (i == 1)
                    cat("Row starting values all converted to zero.",
                        "\nReverting to unsupervised iterations.")
            }
        }
        if (sum(r) > n/2) r <- !r
    }
}
else r <- rep(TRUE, n)
if (is.element(cluster, c("c", "b")))
{
    ## get starting values for columns
    if (start.method == "average"){
        if (is.null(col.classes))
            k <- kmeansStart(rowMeans(Z, dims = 2), iter.startup, TRUE)
        else
            k <- kmeansClassStart(rowMeans(Z, dims = 2), col.classes,
                                  iter.startup, TRUE)
    }
    if (start.method == "convert"){
        k <- rowMeans(apply(Z, 3, kmeansStart, iter.startup,
                            TRUE)) >= 0.5
        if (!is.null(col.classes))
        {
            temp.k <- (tabulate(k * col.classes,
                                nbin = max(col.classes)) >=
                       0.5 * col.grouped)[col.classes]
            if (sum(temp.k) != 0) k <- temp.k
            else
            {
                col.classes <- NULL
                if (i == 1)
                    cat("Column starting values all converted to zero.",
                        "\nReverting to unsupervised iterations.")
            }
        }
        if (sum(k) > p/2) k <- !k
    }
}
else k <- rep(TRUE, p)
```

```
j <- 0
while (j <= iter.layer + extra)
{
    if (0 < j & j <= iter.layer)
    {
        ## update cluster membership parameters, r and k
        if (is.element(cluster, c("r", "b")))
        {
            if (j <= iter.supervised & !is.null(row.classes))
            {
                ## supervised update
                r <- (rowsum(rowSums((Z[, k, , drop = FALSE] -
                                      makeLayer(m, a, b[k], c))^2),
                             row.classes) <
                      rowsum(rowSums(Z[, k, , drop = FALSE]^2),
                             row.classes))[row.classes]
                if (sum(r) == 0)
                {
                    row.classes <- NULL
                    cat("Layer ", i, ": no rows clustered",
                        " - reverting to unsupervised")
                }
            }
            if (j > iter.supervised | is.null(row.classes))
            {
                r <- rowSums((Z[, k, , drop = FALSE] -
                              makeLayer(m, a, b[k], c))^2) <
                                  rowSums(Z[, k, , drop = FALSE]^2)
            }
        }
        n2 <- sum(r)
        if (n2 == 0) break
        if (is.element(cluster, c("c", "b")))
        {
            if (j <= iter.supervised & !is.null(col.classes))
            {
                ## supervised update
                k <- (rowsum(rowSums(aperm((Z[r, , , drop = FALSE] -
                                            makeLayer(m, a[r], b, c)
                                            )^2, c(2, 1, 3))),
                             col.classes) <
                      rowsum(rowSums(aperm(Z[r, , , drop = FALSE]^2,
                                           c(2, 1, 3))),
                             col.classes))[col.classes]
                if (sum(k) == 0)
                {
                    col.classes <- NULL
                    cat("Layer ", i, ": no columns clustered",
                        " - reverting to unsupervised")
                }
            }
            if (j > iter.supervised | is.null(col.classes))
            {
                k <- rowSums(aperm((Z[r, , , drop = FALSE] -
                                    makeLayer(m, a[r], b, c))^2,
                                   c(2, 1, 3))
                             ) < rowSums(aperm(Z[r, , , drop = FALSE]^2,
                                               c(2, 1, 3)))
            }
```

```
    }
}
if (j >= iter.layer + 1 & !is.null(row.release) &
    (j - iter.layer) %% 2 == 1)
{
    ## row release
    if (resdf == 0) r <- rep(0, n)
    else
    {
        r[r] <- (1/resdf) *
            rowSums((Z[r, k, , drop = FALSE] -
                     makeLayer(m, a[r], b[k], c))^2
                    ) < (1 - row.release)/totdf *
                        rowSums(Z[r, k, , drop = FALSE]^2)
    }
}
n2 <- sum(r)
if (j >= iter.layer + 1 & !is.null(col.release) &
    (j - iter.layer) %% 2 == 0)
{
    ## column release
    if (totdf == 0 | resdf == 0) p2 <- 0
    else
    {
        k[k] <- (1/resdf) *
            colSums(rowSums((Z[r, k, , drop = FALSE] -
                             makeLayer(m, a[r], b[k], c))^2,
                            dims = 2)
                    ) < (1 - col.release)/totdf *
                        colSums(rowSums(Z[r, k, , drop =
                                          FALSE]^2, dims = 2))
    }
}
p2 <- sum(k)
if (n2 == 0 | p2 == 0)
{
    if (i == 1)
    {
        if (verbose == TRUE) print(c(j, sum(r), sum(k)))
        stopnow <- TRUE
        n.iter <- j + 1
    }
    break
}
## skip to last iteration if already converged
if ((j >= iter.supervised & j <= iter.layer) |
    (j > iter.layer & (j - iter.layer) %% 2 == 0))
{
    r.check <- c(r.check[2], list((1:n)[r]))
    k.check <- c(k.check[2], list((1:p)[k]))
    if ((identical(r.check[[1]], r.check[[2]]) &
         identical(k.check[[1]], k.check[[2]])))
    {
        if (i == 1 & j <= iter.layer)
        {
            n.iter <- j
            status <- 1
        }
        if (j > iter.layer) j <- iter.layer + extra
```

```
            else j <- iter.layer
        }
    }
    if (j == iter.layer)
    {
        ## use fit.model for final model/basis of row & col rel
        model <- fit.model
        ## save no. of rows & cols in order to calc no. released
        n.rows <- n2
        n.cols <- p2
    }
    ## calculate d.f. for row/col release
    if (j >= iter.layer)
    {
        totdf <- n2 * p2 * t
        resdf <- totdf - (1 + is.element("a", model) * (n2 - 1)
                          + is.element("b", model) * (p2 - 1)
                          + is.element("c", model) * (t - 1))
        if (i == 1 & resdf == 0)
            print("Zero residual degrees of freedom")
    }
    ## update layer effects
    m <- mean(Z[r, k, , drop = FALSE])
    if (is.element("a", model))
    {
        a[r] <- rowMeans(Z[r, k, , drop = FALSE] - m, dims = 1)
        a[!r] <- 0
    }
    if (is.element("b", model))
    {
        b[k] <- colMeans(rowMeans(Z[r, k, , drop = FALSE] - m,
                                  dims = 2))
        b[!k] <- 0
    }
    if (is.element("c", model))
        c <- colMeans(Z[r, k, , drop = FALSE] - m, dims = 2)
    if (i == 1 & verbose == TRUE) print(c(j, sum(r), sum(k)))
    j <- j + 1
}
if (n2 == 0 | p2 == 0)
    cluster.SS[i] <- 0 # strictly NA, put 0 for comparison
else cluster.SS[i] <- sum((makeLayer(m, a[r], b[k], c))^2)
if (i == 1)
{
    ## save results for candidate layer only
    if (!exists("n.iter")) n.iter <- j - 1
    if (verbose == TRUE) print(n.iter)
    id <- list(drop(r), drop(k))
    if (n2 == 0 | p2 == 0)
        fits <- layer.df <- rows.released <- cols.released <- NA
    else
    {
        fits <- makeLayer(m, a[r], b[k], c)
        layer.df <- 1 + is.element("a", model) * (n2 - 1) +
            is.element("b", model) * (p2 - 1) +
                is.element("c", model) * (t - 1)
        if (!is.null(row.release) | cluster == "c")
            rows.released <- n.rows - n2
        else rows.released <- NA
```

```
                if (!is.null(col.release) | cluster == "r")
                    cols.released <- n.cols - p2
                else cols.released <- NA
            }
        }
        if (exists("stopnow")) break
    }
    if (verbose == TRUE) print(cluster.SS)
    if (shuffle > 0)
        cluster.SS <- ifelse(cluster.SS[1] > max(cluster.SS[-1]),
                             cluster.SS[1], 0)
    list(SS = cluster.SS, r = id[[1]], k = id[[2]], fits = fits,
         layer.df = layer.df, status = status,
         rows.released = rows.released, cols.released = cols.released)
}

# backFitting function
backFit <- function(
n.layers, # no. of layers fitted so far (including background)
back.fit, # no. of back fits required
fits,     # list of current fitted values for each layer
r,    # matrix of row membership parameters (rows by layers)
k,    # matrix of column membership parameters (columns by layers)
Z,    # array of current residuals
fit.model)
{
    SS <- numeric(length = length(fits))
    for (b in 1:back.fit)
    {
        for (i in 1:n.layers)
        {
            ## "undo" fit for layer i by adding fitted values to residuals
            Z[r[,i], k[,i], ] <- Z[r[,i], k[,i], , drop = FALSE] + fits[[i]]
            ## re-fit, then recalulate SS and residuals
            fits[[i]] <- fitLayer(Z, r[,i], k[,i], fit.model)
            SS[i] <- sum(fits[[i]]^2)
            Z[r[,i], k[,i], ] <- Z[r[,i], k[,i], , drop = FALSE] - fits[[i]]
        }
    }
    list(SS = SS, fits = fits, Z = Z)
}

## Function to fit layer, given residuals from all other layers in the model
fitLayer <- function(Z, r, k, model)
{
    Z <- Z[r, k, , drop = FALSE]
    m <- mean(Z)
    if (is.element("a", model)) a <- rowMeans(Z - m, dims = 1)
    else a <- numeric(dim(Z)[1])
    if (is.element("b", model)) b <- colMeans(rowMeans(Z - m, dims = 2))
    else b <- numeric(dim(Z)[2])
    if (is.element("c", model)) c <- colMeans(Z - m, dims = 2)
    else c <- numeric(dim(Z)[3])
    makeLayer(m, a, b, c)
}

## Function to construct layer given the fitted effects
makeLayer <- function(m, a, b, c) outer(outer(m + a, b, "+"), c, "+")
```

```
## Function to distribute the elements of a list into objects with the same name
distributeList <- function(x, ind = NULL, margin = NULL)
{
    for (i in seq(along = x))
    {
        if (is.null(ind))
            ## if no index specified, create/overwrite complete object
            assign(names(x)[i], x[[i]], parent.frame())
        else
        {
            ## otherwise identify type of object and put in appropriate part
            temp <- get(names(x)[i], parent.frame())
            ## for existing matrices...
            if (is.array(temp))
            {
                if (is.null(margin))
                    stop(message = "Must specify array margin as well as index")
                ## put in ind'th position in appropriate margin
                indString <- character(length(dim(temp)))
                indString[margin] <- paste("c(", toString(ind), ")")
                eval(parse(text = paste("temp[",toString(indString),
                           "] <- x[[i]]")))
            }
            ## else put in default ind'th position of object
            else temp[[ind]] <- x[[i]]
            assign(names(x)[i], temp, parent.frame())
        }
    }
}
```

```
##### Example calls to use plaid functions #####

## To start, source code for all user and internal functions into R
## See comments in code for plaid() for more detail on its arguments

##### Two-way Analysis #####

## Requires data matrix 'M' and factor 'group' for supervised analysis

## Unsupervised analysis
set.seed(1)
unsup <- plaid(M, back.fit = 2, shuffle = 3, fit.model = ~m + a + b,
               search.model = ~m, row.release = 0.7, col.release = 0.7,
               verbose = TRUE, max.layers = 10, iter.startup = 5,
               iter.layer = 30)

## Supervised analysis
set.seed(1)
sup <- plaid(M, col.classes = group, back.fit = 2, shuffle = 3,
             fit.model = ~m + a + b, search.model = ~m, row.release = 0.7,
             col.release = 0.7, verbose = TRUE, max.layers = 10,
             iter.startup = 5, iter.supervised = 5, iter.layer = 30,
             start = "convert")

##### Three-way Analysis #####

## Requires three-dimensional array 'M':
## the third dimension is assumed to represent repeated measures

set.seed(1)
three.way <- plaid(M, back.fit = 2, shuffle = 3, fit.model = ~m + a + b + c,
                   search.model = ~m + c, row.release = 0.7, col.release = 0.7,
                   start = "convert", verbose = TRUE, max.layers = 5,
                   iter.startup = 5, iter.layer = 30)


##### Summary Method #####

## To print the summary which is printed when a model is fitted

summary(plaid.object)
```