Clusternomics

Stephen Coleman

17/10/2019

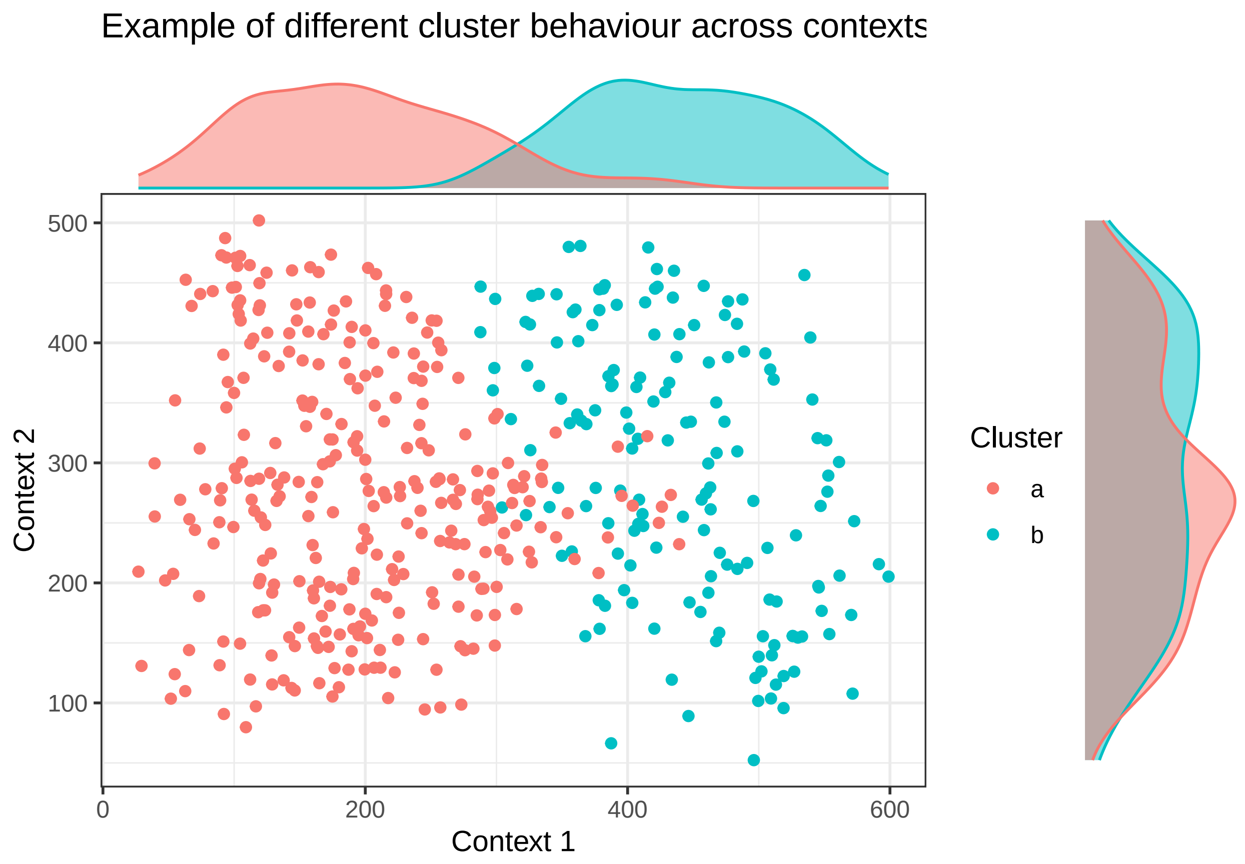
Introduction

Principal aim: to model both **global** (across dataset) and **local** (dataset-specific) clustering structure. Specifically, Clusternomics (Gabasova, Reid, and Wernisch 2017) can allow for clusters merging and separating in different datasets (also referred to as **contexts**). A motivating example is shown below for two 1D datasets.

```
library(ggplot2) # ubiquitous
library(ggExtra) # for ggMarginal
library
```

```
## function (package, help, pos = 2, lib.loc = NULL, character.only = FALSE,
##     logical.return = FALSE, warn.conflicts, quietly = FALSE,
##     verbose = getOption("verbose"), mask.ok, exclude, include.only,
##     attach.required = missing(include.only))
## {
##     conf.ctrl <- getOption("conflicts.policy")
##     if (is.character(conf.ctrl))
##         conf.ctrl <- switch(conf.ctrl, strict = list(error = TRUE,
##             warn = FALSE), depends.ok = list(error = TRUE, generics.ok = TRUE,
##             can.mask = c("base", "methods", "utils", "grDevices",
##                 "graphics", "stats"), depends.ok = TRUE), warning(gettextf("unknown conflict policy:
##             sQuote(conf.ctrl)), call. = FALSE, domain = NA))
##     if (!is.list(conf.ctrl))
##         conf.ctrl <- NULL
##     stopOnConflict <- isTRUE(conf.ctrl$error)
##     if (missing(warn.conflicts))
##         warn.conflicts <- if (isFALSE(conf.ctrl$warn))
##             FALSE
##         else TRUE
##     if ((!missing(include.only)) && (!missing(exclude)))
##         stop(gettext("only one of 'include.only' and 'exclude' can be used"),
##             call. = FALSE, domain = NA)
##     testRversion <- function(pkgInfo, pkgname, pkgpath) {
##         if (is.null(built <- pkgInfo$Built))
##             stop(gettextf("package %s has not been installed properly\n",
##                 sQuote(pkgname)), call. = FALSE, domain = NA)
##         R_version_built_under <- as.numeric_version(built$R)
##         if (R_version_built_under < "3.0.0")
##             stop(gettextf("package %s was built before R 3.0.0: please re-install it",
##                 sQuote(pkgname)), call. = FALSE, domain = NA)
##         current <- getRversion()
##         if (length(Rdeps <- pkgInfo$Rdepends2)) {
##             for (dep in Rdeps) if (length(dep) > 1L) {
##                 target <- dep$version
##                 res <- if (is.character(target)) {
##                   do.call(dep$op, list(as.numeric(R.version[["svn rev"]]),
##                     as.numeric(sub("^r", "", dep$version))))
##                 }
##                 else {
##                   do.call(dep$op, list(current, as.numeric_version(target)))
##                 }
##                 if (!res)
##                   stop(gettextf("This is R %s, package %s needs %s %s",
##                     current, sQuote(pkgname), dep$op, target),
##                     call. = FALSE, domain = NA)
##             }
##         }
##         if (R_version_built_under > current)
##             warning(gettextf("package %s was built under R version %s",
##                 sQuote(pkgname), as.character(built$R)), call. = FALSE,
##                 domain = NA)
##         platform <- built$Platform
##         r_arch <- .Platform$r_arch
##         if (.Platform$OS.type == "unix") {
##         }
##         else {
##             if (nzchar(platform) && !grepl("mingw", platform))
##                 stop(gettextf("package %s was built for %s",
##                   sQuote(pkgname), platform), call. = FALSE,
##                   domain = NA)
##         }
##         if (nzchar(r_arch) && file.exists(file.path(pkgpath,
##             "libs")) && !file.exists(file.path(pkgpath, "libs",
##             r_arch)))
##             stop(gettextf("package %s is not installed for 'arch = %s'",
##                 sQuote(pkgname), r_arch), call. = FALSE, domain = NA)
##     }
##     checkNoGenerics <- function(env, pkg) {
##         nenv <- env
##         ns <- .getNamespace(as.name(pkg))
##         if (!is.null(ns))
##             nenv <- asNamespace(ns)
##         if (exists(".noGenerics", envir = nenv, inherits = FALSE))
##             TRUE
##         else {
##             !any(startsWith(names(env), ".__T"))
##         }
##     }
##     checkConflicts <- function(package, pkgname, pkgpath, nogenerics,
##         env) {
##         dont.mind <- c("last.dump", "last.warning", ".Last.value",
##             ".Random.seed", ".Last.lib", ".onDetach", ".packageName",
##             ".noGenerics", ".required", ".no_S3_generics", ".Depends",
##             ".requireCachedGenerics")
##         sp <- search()
##         lib.pos <- which(sp == pkgname)
##         ob <- names(as.environment(lib.pos))
##         if (!nogenerics) {
##             these <- ob[startsWith(ob, ".__T__")]
##             gen <- gsub(".__T__(.*):([^:]+)", "\\1", these)
##             from <- gsub(".__T__(.*):([^:]+)", "\\2", these)
##             gen <- gen[from != package]
##             ob <- ob[!(ob %in% gen)]
##         }
##         ipos <- seq_along(sp)[-c(lib.pos, match(c("Autoloads",
##             "CheckExEnv"), sp, 0L))]
##         cpos <- NULL
##         conflicts <- vector("list", 0)
##         for (i in ipos) {
##             obj.same <- match(names(as.environment(i)), ob, nomatch = 0L)
##             if (any(obj.same > 0)) {
##                 same <- ob[obj.same]
##                 same <- same[!(same %in% dont.mind)]
##                 Classobjs <- which(startsWith(same, ".__"))
##                 if (length(Classobjs))
##                   same <- same[-Classobjs]
##                 same.isFn <- function(where) vapply(same, exists,
##                   NA, where = where, mode = "function", inherits = FALSE)
##                 same <- same[same.isFn(i) == same.isFn(lib.pos)]
##                 not.Ident <- function(ch, TRAFO = identity, ...) vapply(ch,
##                   function(.) !identical(TRAFO(get(., i)), TRAFO(get(.,
##                     lib.pos)), ...), NA)
##                 if (length(same))
##                   same <- same[not.Ident(same)]
##                 if (length(same) && identical(sp[i], "package:base"))
##                   same <- same[not.Ident(same, ignore.environment = TRUE)]
##                 if (length(same)) {
##                   conflicts[[sp[i]]] <- same
##                   cpos[sp[i]] <- i
##                 }
##             }
##         }
##         if (length(conflicts)) {
##             if (stopOnConflict) {
##                 emsg <- ""
##                 pkg <- names(conflicts)
##                 notOK <- vector("list", 0)
##                 for (i in seq_along(conflicts)) {
##                   pkgname <- sub("^package:", "", pkg[i])
##                   if (pkgname %in% canMaskEnv$canMask)
##                     next
##                   same <- conflicts[[i]]
##                   if (is.list(mask.ok))
##                     myMaskOK <- mask.ok[[pkgname]]
##                   else myMaskOK <- mask.ok
##                   if (isTRUE(myMaskOK))
##                     same <- NULL
##                   else if (is.character(myMaskOK))
##                     same <- setdiff(same, myMaskOK)
##                   if (length(same)) {
##                     notOK[[pkg[i]]] <- same
##                     msg <- .maskedMsg(sort(same), pkg = sQuote(pkg[i]),
##                       by = cpos[i] < lib.pos)
##                     emsg <- paste(emsg, msg, sep = "\n")
##                   }
##                 }
##                 if (length(notOK)) {
##                   msg <- gettextf("Conflicts attaching package %s:\n%s",
##                     sQuote(package), emsg)
##                   stop(errorCondition(msg, package = package,
##                     conflicts = conflicts, class = "packageConflictError"))
##                 }
##             }
##             if (warn.conflicts) {
##                 packageStartupMessage(gettextf("\nAttaching package: %s\n",
##                   sQuote(package)), domain = NA)
##                 pkg <- names(conflicts)
##                 for (i in seq_along(conflicts)) {
##                   msg <- .maskedMsg(sort(conflicts[[i]]), pkg = sQuote(pkg[i]),
##                     by = cpos[i] < lib.pos)
##                   packageStartupMessage(msg, domain = NA)
##                 }
##             }
##         }
##     }
##     if (verbose && quietly)
##         message("'verbose' and 'quietly' are both true; being verbose then ..")
##     if (!missing(package)) {
##         if (is.null(lib.loc))
##             lib.loc <- .libPaths()
##         lib.loc <- lib.loc[dir.exists(lib.loc)]
##         if (!character.only)
##             package <- as.character(substitute(package))
##         if (length(package) != 1L)
##             stop("'package' must be of length 1")
##         if (is.na(package) || (package == ""))
##             stop("invalid package name")
##         pkgname <- paste0("package:", package)
##         newpackage <- is.na(match(pkgname, search()))
##         if (newpackage) {
##             pkgpath <- find.package(package, lib.loc, quiet = TRUE,
##                 verbose = verbose)
##             if (length(pkgpath) == 0L) {
##                 if (length(lib.loc) && !logical.return)
##                   stop(packageNotFoundError(package, lib.loc,
##                     sys.call()))
##                 txt <- if (length(lib.loc))
##                   gettextf("there is no package called %s", sQuote(package))
##                 else gettext("no library trees found in 'lib.loc'")
##                 if (logical.return) {
##                   warning(txt, domain = NA)
##                   return(FALSE)
##                 }
##                 else stop(txt, domain = NA)
##             }
##             which.lib.loc <- normalizePath(dirname(pkgpath),
##                 "/", TRUE)
##             pfile <- system.file("Meta", "package.rds", package = package,
##                 lib.loc = which.lib.loc)
##             if (!nzchar(pfile))
##                 stop(gettextf("%s is not a valid installed package",
##                   sQuote(package)), domain = NA)
##             pkgInfo <- readRDS(pfile)
##             testRversion(pkgInfo, package, pkgpath)
##             if (is.character(pos)) {
##                 npos <- match(pos, search())
##                 if (is.na(npos)) {
##                   warning(gettextf("%s not found on search path, using pos = 2",
##                     sQuote(pos)), domain = NA)
##                   pos <- 2
##                 }
##                 else pos <- npos
##             }
##             deps <- unique(names(pkgInfo$Depends))
##             depsOK <- isTRUE(conf.ctrl$depends.ok)
##             if (depsOK) {
##                 canMaskEnv <- dynGet("__library_can_mask__",
##                   NULL)
##                 if (is.null(canMaskEnv)) {
##                   canMaskEnv <- new.env()
##                   canMaskEnv$canMask <- union("base", conf.ctrl$can.mask)
##                   "__library_can_mask__" <- canMaskEnv
##                 }
##                 canMaskEnv$canMask <- unique(c(package, deps,
##                   canMaskEnv$canMask))
##             }
##             else canMaskEnv <- NULL
##             if (attach.required)
##                 .getRequiredPackages2(pkgInfo, quietly = quietly)
##             cr <- conflictRules(package)
##             if (missing(mask.ok))
##                 mask.ok <- cr$mask.ok
##             if (missing(exclude))
##                 exclude <- cr$exclude
##             if (packageHasNamespace(package, which.lib.loc)) {
##                 if (isNamespaceLoaded(package)) {
##                   newversion <- as.numeric_version(pkgInfo$DESCRIPTION["Version"])
##                   oldversion <- as.numeric_version(getNamespaceVersion(package))
##                   if (newversion != oldversion) {
##                     tryCatch(unloadNamespace(package), error = function(e) {
##                       P <- if (!is.null(cc <- conditionCall(e)))
##                         paste("Error in", deparse(cc)[1L], ": ")
##                       else "Error : "
##                       stop(gettextf("Package %s version %s cannot be unloaded:\n %s",
##                         sQuote(package), oldversion, paste0(P,
##                           conditionMessage(e), "\n")), domain = NA)
##                     })
##                   }
##                 }
##                 tt <- tryCatch({
##                   attr(package, "LibPath") <- which.lib.loc
##                   ns <- loadNamespace(package, lib.loc)
##                   env <- attachNamespace(ns, pos = pos, deps,
##                     exclude, include.only)
##                 }, error = function(e) {
##                   P <- if (!is.null(cc <- conditionCall(e)))
##                     paste(" in", deparse(cc)[1L])
##                   else ""
##                   msg <- gettextf("package or namespace load failed for %s%s:\n %s",
##                     sQuote(package), P, conditionMessage(e))
##                   if (logical.return)
##                     message(paste("Error:", msg), domain = NA)
##                   else stop(msg, call. = FALSE, domain = NA)
##                 })
##                 if (logical.return && is.null(tt))
##                   return(FALSE)
##                 attr(package, "LibPath") <- NULL
##                 {
##                   on.exit(detach(pos = pos))
##                   nogenerics <- !.isMethodsDispatchOn() || checkNoGenerics(env,
##                     package)
##                   if (isFALSE(conf.ctrl$generics.ok) || (stopOnConflict &&
##                     !isTRUE(conf.ctrl$generics.ok)))
##                     nogenerics <- TRUE
##                   if (stopOnConflict || (warn.conflicts && !exists(".conflicts.OK",
##                     envir = env, inherits = FALSE)))
##                     checkConflicts(package, pkgname, pkgpath,
##                       nogenerics, ns)
##                   on.exit()
##                   if (logical.return)
##                     return(TRUE)
##                   else return(invisible(.packages()))
##                 }
##             }
##             else stop(gettextf("package %s does not have a namespace and should be re-installed",
##                 sQuote(package)), domain = NA)
##         }
##         if (verbose && !newpackage)
##             warning(gettextf("package %s already present in search()",
##                 sQuote(package)), domain = NA)
##     }
##     else if (!missing(help)) {
##         if (!character.only)
##             help <- as.character(substitute(help))
##         pkgName <- help[1L]
##         pkgPath <- find.package(pkgName, lib.loc, verbose = verbose)
##         docFiles <- c(file.path(pkgPath, "Meta", "package.rds"),
##             file.path(pkgPath, "INDEX"))
##         if (file.exists(vignetteIndexRDS <- file.path(pkgPath,
##             "Meta", "vignette.rds")))
##             docFiles <- c(docFiles, vignetteIndexRDS)
##         pkgInfo <- vector("list", 3L)
##         readDocFile <- function(f) {
##             if (basename(f) %in% "package.rds") {
##                 txt <- readRDS(f)$DESCRIPTION
##                 if ("Encoding" %in% names(txt)) {
##                   to <- if (Sys.getlocale("LC_CTYPE") == "C")
##                      "ASCII//TRANSLIT"
##                    else ""
##                    tmp <- try(iconv(txt, from = txt["Encoding"],
##                      to = to))
##                    if (!inherits(tmp, "try-error"))
##                      txt <- tmp
##                    else warning("'DESCRIPTION' has an 'Encoding' field and re-encoding is not possible
##                      call. = FALSE)
##                  }
##                  nm <- paste0(names(txt), ":")
##                  formatDL(nm, txt, indent = max(nchar(nm, "w")) +
##                    3L)
##              }
##              else if (basename(f) %in% "vignette.rds") {
##                  txt <- readRDS(f)
##                  if (is.data.frame(txt) && nrow(txt))
##                    cbind(basename(gsub("\\.[[:alpha:]]+$", "",
##                      txt$File)), paste(txt$Title, paste0(rep.int("(source",
##                      NROW(txt)), ifelse(nzchar(txt$PDF), ", pdf",
##                      ""), ")")))
##                  else NULL
##              }
##              else readLines(f)
##          }
##          for (i in which(file.exists(docFiles))) pkgInfo[[i]] <- readDocFile(docFiles[i])
##          y <- list(name = pkgName, path = pkgPath, info = pkgInfo)
##          class(y) <- "packageInfo"
##          return(y)
##      }
##      else {
##          if (is.null(lib.loc))
##              lib.loc <- .libPaths()
##          db <- matrix(character(), nrow = 0L, ncol = 3L)
##          nopkgs <- character()
##          for (lib in lib.loc) {
##              a <- .packages(all.available = TRUE, lib.loc = lib)
##              for (i in sort(a)) {
##                  file <- system.file("Meta", "package.rds", package = i,
##                    lib.loc = lib)
##                  title <- if (nzchar(file)) {
##                    txt <- readRDS(file)
##                    if (is.list(txt))
##                      txt <- txt$DESCRIPTION
##                    if ("Encoding" %in% names(txt)) {
##                      to <- if (Sys.getlocale("LC_CTYPE") == "C")
##                        "ASCII//TRANSLIT"
##                      else ""
##                      tmp <- try(iconv(txt, txt["Encoding"], to,
##                        "?"))
##                      if (!inherits(tmp, "try-error"))
##                        txt <- tmp
##                      else warning("'DESCRIPTION' has an 'Encoding' field and re-encoding is not possil
##                        call. = FALSE)
##                    }
##                     txt["Title"]
##                   }
##                   else NA
##                   if (is.na(title))
##                     title <- " ** No title available ** "
##                   db <- rbind(db, cbind(i, lib, title))
##               }
##               if (length(a) == 0L)
##                   nopkgs <- c(nopkgs, lib)
##           }
##           dimnames(db) <- list(NULL, c("Package", "LibPath", "Title"))
##           if (length(nopkgs) && !missing(lib.loc)) {
##               pkglist <- paste(sQuote(nopkgs), collapse = ", ")
##               msg <- sprintf(ngettext(length(nopkgs), "library %s contains no packages",
##                   "libraries %s contain no packages"), pkglist)
##               warning(msg, domain = NA)
##           }
##           y <- list(header = NULL, results = db, footer = NULL)
##           class(y) <- "libraryIQR"
##           return(y)
##       }
##       if (logical.return)
##           TRUE
##       else invisible(.packages())
## }
## <bytecode: 0x563c769e0a30>
## <environment: namespace:base>
```

```r
# Personal preference
theme_set(theme_bw())

# Data generated at http://drawdata.xyz/
my_data <- read.csv("./example_data.csv")

# Plot the data
p1 <- ggplot(data = my_data, mapping = aes(x = x, y = y, colour = z)) +
  geom_point() +
  labs(
    title = "Example of different cluster behaviour across contexts",
    x = "Context 1",
    y = "Context 2",
    colour = "Cluster"
  )

# Add marginal density plots by grouping
ggMarginal(p1, groupColour = T, groupFill = T)
```



Here we have two separate subpopulations. In Context 1 the sub-populations are discernible as two local clusters. However this not the case in Contex 2. Clusternomics allows for such disagreement in local models while conveying the complexity of clustering structure (in this case that there really is two sub-populations present) to the global model.

The model

Clusternomics, which is embedded in the Bayesian clustering framework, uses a hierarchical Dirichlet mixture model to identify structure on both the local and the global level. The model is fit to the data via Gibbs sampling. The model does not assume that cluster behaviour will be consistent across heterogeneous datasets. This is not to assume that the clustering structure uncovered in one dataset should not inform the clustering in another dataset. This can be summarised as so:

1. Clustering structure in one dataset should inform the clustering in another dataset. If two points are clustered together in one context they should be more inclined to cluster together in other contexts.
2. Different degrees of dependence should be allowed between clusters across contexts. The model should work when datasets have the same underlying structure and also when each dataset is effecitvely independent of all others. Fundamental to this is allowing datasets to have different numbers of clusters.

To enable these modelling aims, Clusternomics explicitly represents the local clusters (i.e. dataset specific) and the global structure that emerges when considering the combination of the datasets. The global clusters are defined by combinations of local clusters. Consider the case where 3 clusters emerge in Context 1 (denoted by labels $\{1, 2, 3\}$) and 2 clusters emerge in Context 2 (denoted by labels $\{A, B\}$). In this case our global structure has the possibile form:

$$\{(1, A), (2, A), (3, A), (1, B), (2, B), (3, B)\}$$

Thus if a point is assigned a label of 1 in Context 1 and a label of A in Context 2 it increases the probability of cluster $(1, A)$ becoming populated at the global level. However, it is possible that some of the possible global clusters described above are not realised as some local clusters overlap across datasets. Consider the case that labellings 1 and 2 from the first context are captured entirely by label A in the second context with a albel of 3 corresponding perfectly to label B. In this case our global structure would take the form:

$$\{(1, A), (2, A), (3, B)\}$$

In this way the local structure informs the global structure.

The original paper introduces two models that are "asymptotically equivalent". The first is easier to develop an intuition of, but it is the second that is implmented as it is more computationally efficient.

Notation

Let us denote the number of datasets by L and all observed data by X. Let

$$X = (X_1, \ldots, X_L),$$
$$X_l = (X_{l1}, \ldots, X_{ln})$$

where X_l is the data of the lth context.

It is assumed that it is the same n individuals in each dataset in the same order. Therefore we have L membership vectors denoting cluster membership:

$$C = (C_1, \ldots, C_L),$$
$$C_l = (c_{l1}, \ldots, c_{ln}).$$

Basic model

The basis of the integrative model is a finite approximation of a Dirichlet process known as a Dirichlet-Multinomial Allocation mixture model (Green and Richardson 2001). There is a nice explanation of this model in Savage et al. (2013).

In this case we model the latent structure in the lth dataset using a mixture of K_l components. This means that the full model density is the weighted sum of the probability density functions associated with each component where the weights, π_{lk}, are the proportion of the total population assigned to the kth components:

$$p(X_{li} | c_{li} = k) = \pi_{lk} f(X_{li} | \theta_{lk}),$$
$$p(X_{li}) = \sum_{k=1}^{K_l} \pi_{lk} f_l(X_{li} | \theta_{lk}).$$

Here K_l is the total number of clusters present and θ_{lk} are the parameters defining the kth distribution in the lth dataset.

The weights, $\pi_l = (\pi_{l1}, \ldots, \pi_{lK_l})$, follow a Dirichlet distribution with concentration parameter α_0.

The distributions in the mixture model for each dataset are:

$$\begin{aligned}
\pi_l &\sim \text{Dirichlet}\left(\frac{\alpha_0}{K_l}, \ldots, \frac{\alpha_0}{K_l}\right) \\
c_{li} &\sim \text{Categorical}(\pi_l) \\
\theta_{lk} &\sim h_l \\
X_{li} | c_{li} = k &\sim f_l(X_{li} | \theta_{lk})
\end{aligned}$$

where H_l is some prior distribution for parameteters for each mixture component; F_l is a probability distribution for samples given the parameters θ_{lk}. Note that each context may have different distributions and hyperparameters.

First formulation: the intuitive model

For ease of understanding, let $L = 2$. Then each context has its own mixture weights with symmetric Dirichlet priors:

$$\begin{aligned}
\pi_1 &\sim \text{Dirichlet}\left(\frac{\alpha_1}{K_1}, \ldots, \frac{\alpha_1}{K_1}\right) \\
\pi_2 &\sim \text{Dirichlet}\left(\frac{\alpha_2}{K_2}, \ldots, \frac{\alpha_2}{K_2}\right)
\end{aligned}$$

These weights form the basis of the local clustering within each dataset. A third mixure distribution is used to link the two local clusters. This has a Dirichlet prior over the global mixture weights, ρ, which is defined over the outer product of the local weights:

$$\rho \sim \text{Dirichlet}\left(\gamma \ \text{vec}(\pi_1 \otimes \pi_2)\right).$$

The outer product is defined:

$$\begin{aligned}
\pi_1 \otimes \pi_2 &= \pi_1 \pi_2^T \\
&= \begin{pmatrix}
\pi_{11}\pi_{21} & \pi_{11}\pi_{22} & \cdots & \pi_{11}\pi_{2K_2} \\
\pi_{12}\pi_{21} & \pi_{12}\pi_{22} & \cdots & \pi_{12}\pi_{2K_2} \\
\vdots & \vdots & \ddots & \vdots \\
\pi_{1K_1}\pi_{21} & \pi_{1K_1}\pi_{22} & \cdots & \pi_{1K_1}\pi_{2K_2}
\end{pmatrix}
\end{aligned}$$

Then the vector function takes the column vectors of the matrix and places them in one vector:

$$\text{vec}(\pi_1 \otimes \pi_2) = \begin{pmatrix}
\pi_{11}\pi_{21} \\ \vdots \\ \pi_{1K_1}\pi_{21} \\ \pi_{11}\pi_{22} \\ \vdots \\ \pi_{1K_1}\pi_{22} \\ \pi_{11}\pi_{23} \\ \vdots \\ \pi_{1K_1}\pi_{2K_2}
\end{pmatrix}$$

This is the basis for the non-symmetric concentration parameter of the Dirichlet distribution over the global mixture weights, ρ.

A global membership variable, c, is drawn from a Catgeorical distribution with concentration parameter ρ:

$$\begin{aligned} c_i &= (c_{1i}, c_{2i}) \\ c &\sim \text{Categorical}(\rho) \\ X_{li} | c_{li} = k &\sim f_l(X_{li} | \theta_{lk}), l \in \{1, 2\} \end{aligned}$$

In this way the local clusters, c_{li}, in the *lth* context are projections of the global clustering, c_i, onto this space.

The model achieves the property that the local assignment should affect the global assignment posterior probability. This can be seen as local assignments affect the posterior probability of the context-specific weights, π_l, which in turn define the probabilities of the global combinatorial clusters through the hierarchical model (a change in the membership of the *kth* component in the *lth* dataset affects an entire slice of column vectors in the tensors defining the concentration parameter for the global component weights, ρ).

The model also represents different degrees of dependence by allowing any combination of cluster assignments across contexts. When there is a single common cluster structure across the two contexts, the occupied clusters will be concentrated along the diagonal of the probability matrix of ρ.

When $L > 2$ then the outer product of the local componenet weights generalises to a tensor product. In this case the hierarchical model is given by:

$$\begin{aligned} \pi_l | \alpha_0 &\sim \text{Dirichlet}\left(\frac{\alpha_0}{K_l}, \ldots, \frac{\alpha_0}{K_l}\right) \\ \rho | \gamma, \{\pi_1 \ldots, \pi_L\} &\sim \text{Dirichlet}\left(\gamma \left\{ \bigotimes_{l=1}^{L} \pi_l \right\}\right) \\ c_i | \rho &\sim \text{Categorical}(\rho), c_i = (c_{1i}, \ldots, c_{Li}) \\ \theta_{lk} &\sim h_l \\ X_{li} | c_{li} = k &\sim f_l(X_{li} | \theta_{lk}) \end{aligned}$$

The model for L contexts has a scaling issue as each additional dataset brings an additional layer of calculations in the probability tensor. The second model is designed to reduce this cost by avoiding calculations for uninhabited components.

Second formulation: the quick model

This model attempts to reduce the number of combinations required. This requires decoupling the number of local clusters and the number of global clusters. A mixture over S global clusters is defined:

$$\begin{aligned} \rho | \gamma_0 &\sim \text{Dirichlet}\left(\frac{\gamma_0}{S}, \ldots, \frac{\gamma_0}{S}\right) \\ c_i | \rho &\sim \text{Categorical}(\rho) \end{aligned}$$

where ρ is the global mixture weights and c_i is the components assignment indivator variable as before. A variable z_{ls} is then defined that associates the *sth* global cluster with context specific clusters:

$$\begin{aligned} \pi_l | \alpha_0 &\sim \text{Dirichlet}\left(\frac{\alpha_0}{K_l}, \ldots, \frac{\alpha_0}{K_l}\right), \\ z_{ls} | \pi_l &\sim \text{Categorical}(\pi_l). \end{aligned}$$

Here $z_{ls} \in \{1, \ldots, K_l\}$ assignes the *sth* global cluster to the a specific local cluster in the *lth* dataset. One may consider $(z_{sl})_{s=1}^{S}$ as the coordinates of the global clusters in the *lth* dataset. This link means that the c_i maps a point to a global cluster as well as the local clusters.

In the previous model the mapping of global clusters to local clusters was implicit, because each combination of context clusters mapped to a unique global cluster. In this model, the mapping is probabilistic and forms a part of the model. One may now state S, the number of global components, and thus limit the number of computations required in contrast to the preceding model.

This hierarchial model is defined by:

$$\begin{aligned}
\rho|\gamma_0 &\sim \text{Dirichlet}\left(\frac{\gamma_0}{S}, \ldots, \frac{\gamma_0}{S}\right), \\
c_i|\rho &\sim \text{Categorical}(\rho), \\
\pi_l|\alpha_0 &\sim \text{Dirichlet}\left(\frac{\alpha_0}{K_l}, \ldots, \frac{\alpha_0}{K_l}\right), \\
z_{ls}|\pi_l &\sim \text{Categorical}(\pi_l) \\
\theta_{kl}|h_l &\sim h_l \\
X_{li}|c_i, (z_{sl})_{s=1}^{S}, (\theta_{lk})_{k=1}^{K} &\sim f_l(X_{li}|\theta_{lk_{c_i}})
\end{aligned}$$

Inference

The quick, decoupled model is implemented in the R package `clusternomics` and uses Gibbs sampling as the inference algorithm.

R package

Setup

The `clusternomics` package is available on CRAN.

```r
# If not installed, remedy this.
if (!requireNamespace("clusternomics", quietly = TRUE)) {
  devtools::install_github("evelinag/clusternomics")
  # install.packages("clusternomics")
}

# Load the library
library(clusternomics)

# For the pipe
library(magrittr)
library(plyr)
```

Evelina has a vignette showcasing the Clusternomics model on simulated data at https://cran.r-project.org/web/packages/clusternomics/vignettes/using-clusternomics.html.

First, define two local clusters in each of two contexts:

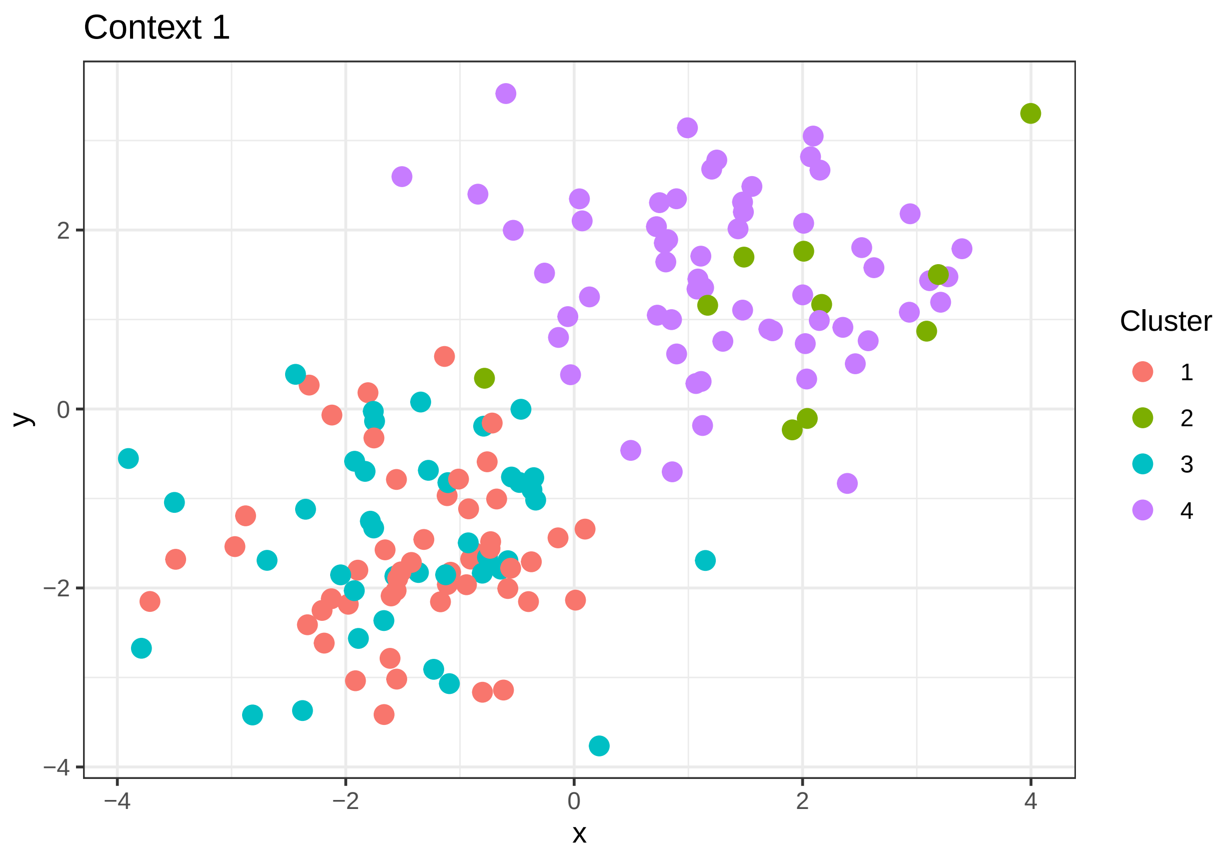
$$\begin{aligned}
\text{Cluster 1: } x &\sim \mathcal{N}\left(\begin{pmatrix}-1.5\\-1.5\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix}\right) \\
\text{Cluster 2: } x &\sim \mathcal{N}\left(\begin{pmatrix}1.5\\1.5\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix}\right)
\end{aligned}$$

Here the first row of each vector is the paramters for the normal distribution in the first context and the second row is the value of the parameters in the second context.

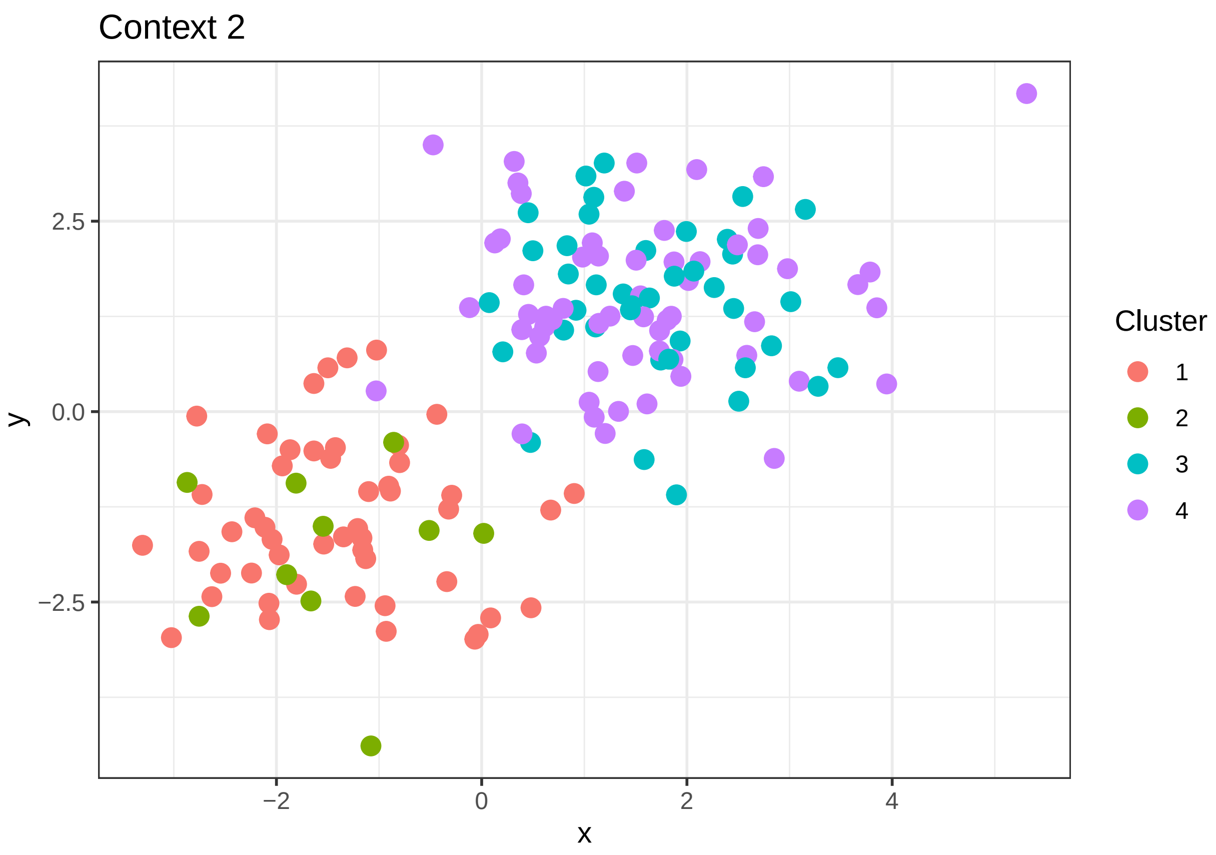
	Cluster 1 (Context 2)	Cluster 2 (Context 2)
Cluster 1 (Context 1)	50	10
Cluster 2 (Context 1)	40	60

There are 160 data points in total. There are two local clusters within each context. At the same time, there are four clusters on the global level that appear when we combine observations from the local contexts.

```r
# For consistent results
set.seed(1)

# Number of elements in each cluster, follows the table given above
groupCounts <- c(50, 10, 40, 60)
# Centers of clusters
means <- c(-1.5, 1.5)
# Helper function to generate test data
testData <- generateTestData_2D(groupCounts, means)
datasets <- testData$data

# Inspect the data in each context
# Context 1
qplot(datasets[[1]][, 1], datasets[[1]][, 2], col = factor(testData$groups)) +
  geom_point(size = 3) +
  ggtitle("Context 1") + xlab("x") + ylab("y") +
  scale_color_discrete(name = "Cluster")
```



```
# Context 2
qplot(datasets[[2]][, 1], datasets[[2]][, 2], col = factor(testData$groups)) +
  geom_point(size = 3) +
  ggtitle("Context 2") + xlab("x") + ylab("y") +
  scale_color_discrete(name = "Cluster")
```



Note that the following assumptions work awfully well in this case as the data is simulated. In real life the number of MCMC iterations would have to be signigicantly larger to see convergence-like behaviour. We can also use relatively low upper bounds on the number of clusters as we know the ground truth. This also decreases the time required within each individual iteration.

```
# Setup of the algorithm
dataDistributions <- "diagNormal"

# Pre-specify number of clusters
clusterCounts <- list(global = 10, context = c(3, 3))

# Set number of iterations
# The following is ONLY FOR SIMULATION PURPOSES
# Use larger number of iterations for real-life data
maxIter <- 300
burnin <- 200
lag <- 2 # Thinning of samples

# Run context-dependent clustering
results <- contextCluster(datasets, clusterCounts,
  maxIter = maxIter,
  burnin = burnin,
  lag = lag,
  dataDistributions = "diagNormal",
  verbose = F
)

# Extract resulting cluster assignments
```

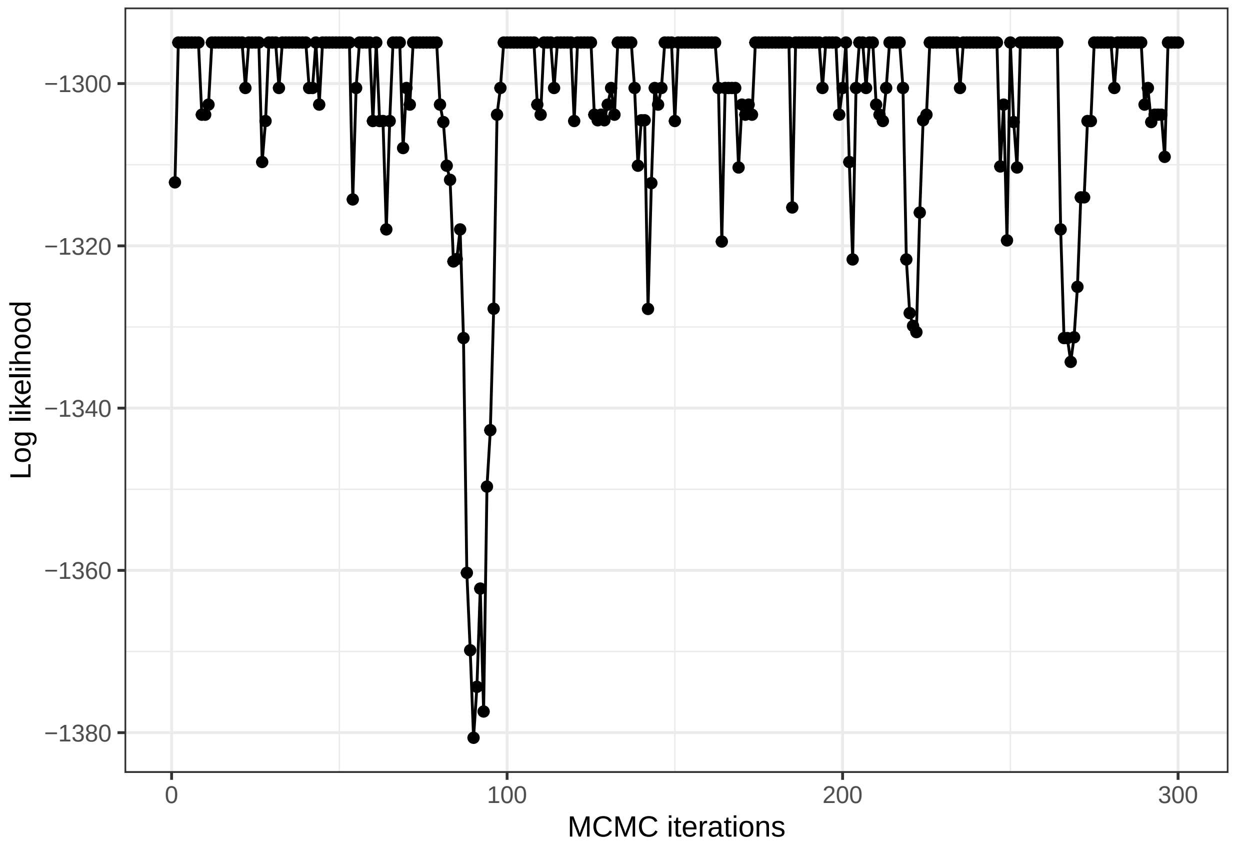
```
samples <- results$samples

# Extract global cluster assignments for each MCMC sample
clusters <- laply(1:length(samples), function(i) samples[[i]]$Global)
# clusters <- sapply(1:length(samples), function(i) samples[[i]]$Global) %>%
# t()
```

Model analysis

Convergence

The defualt for checking convergence, trace plots. In this case, Gabasova, Reid, and Wernisch (2017) recommend log-likelihood as the quantity to measure.

```
logliks <- results$logliks

qplot(1:maxIter, logliks) +
  geom_line() +
  xlab("MCMC iterations") +
  ylab("Log likelihood")
```



The below is copy and pasted from https://cran.r-project.org/web/packages/clusternomics/vignettes/using-clusternomics.html as I don't think there's much to say here.

Choosing number of clusters

As part of the training, we also compute the Deviance Information Criterion (DIC). The DIC is a Bayesian model selection method that can be used to select the number of clusters. The DIC value is returned in results$DIC. Models that better fit the data will result in lower values of DIC than worse models. For example, if we fit a model with number of clusters that is too small, we get higher DIC value than for the original result.

```r
wrongClusterCounts <- list(global = 2, context = c(2, 1))
worseResults <- contextCluster(datasets, wrongClusterCounts,
  maxIter = maxIter,
  burnin = burnin,
  lag = lag,
  dataDistributions = "diagNormal",
  verbose = F
)

print(paste("Original model has lower (better) DIC:", results$DIC))
```

```
## [1] "Original model has lower (better) DIC: 2604.21806172573"
```

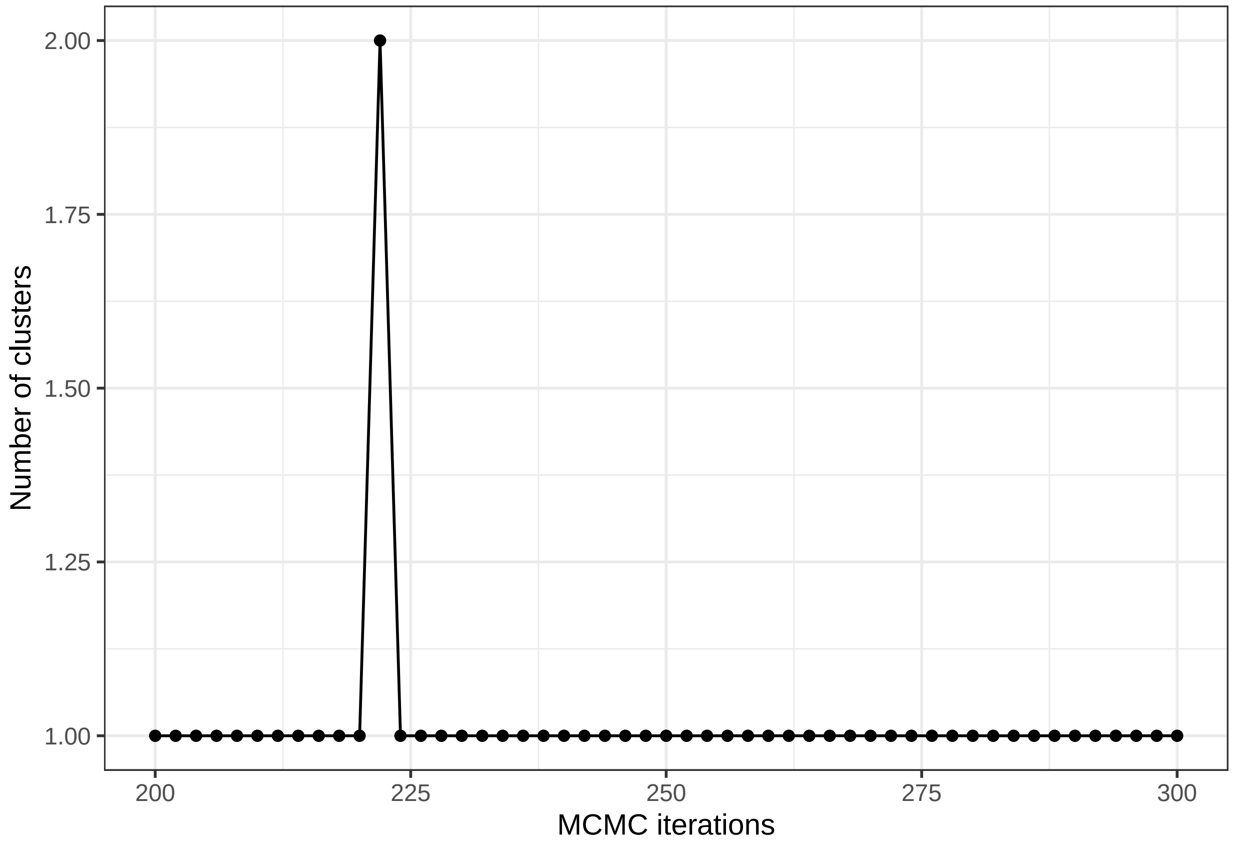
```r
print(paste("Worse model has higher (worse) DIC:", worseResults$DIC))
```

```
## [1] "Worse model has higher (worse) DIC: 2584.25139051911"
```

Posterior number of clusters

We can also look at the number of global clusters that were identified in the datasets. The plot below shows the number of global clusters across MCMC samples. This is the number of actually occupied global clusters, which can be smaller than the number of global clusters specified when running the contextCluster function.

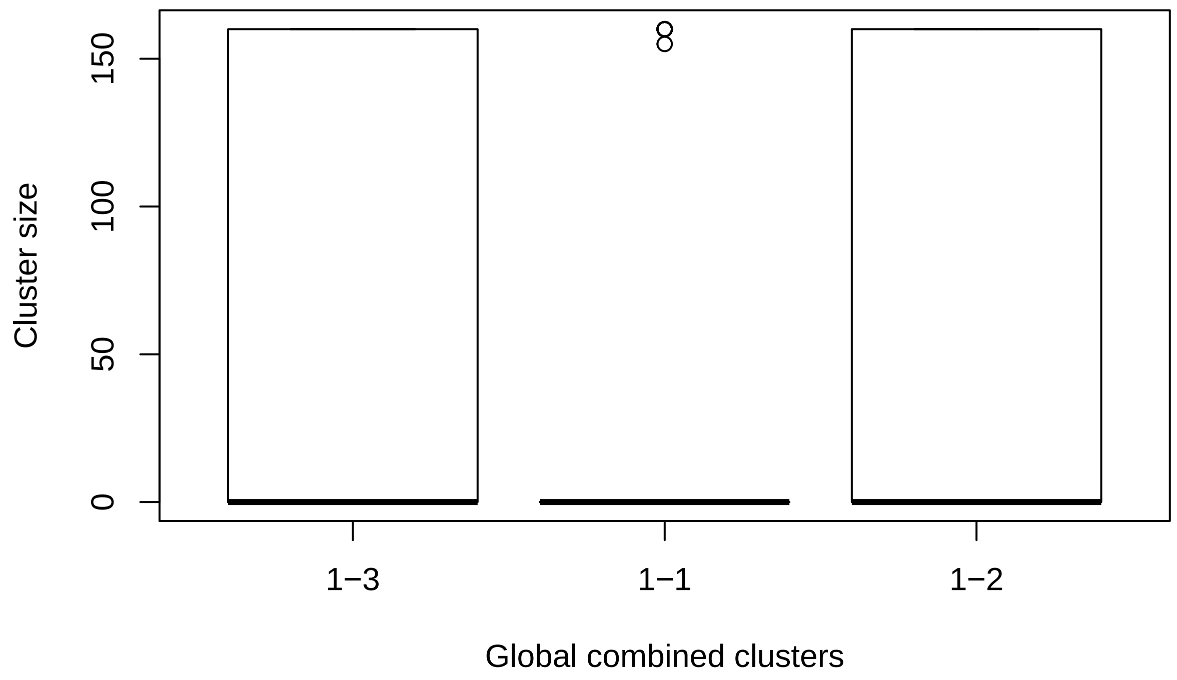
```r
# This does not run as the number of clusters found is precisely 1.
cc <- numberOfClusters(clusters) # This function does not seem to work
# cc <- lapply(1:length(clusters), function(i) unique(clusters[[i]]) %>% length )

qplot(seq(from = burnin, to = maxIter, by = lag), cc) +
  geom_line() +
  xlab("MCMC iterations") +
  ylab("Number of clusters")
```



Sizes of clusters

Here we look at the posterior sizes of the individual global clusters across the MCMC iterations and then we show a box plot with the estimated sizes. The labels of global clusters represent the corresponding combinations of local clusters.

```r
# There's some issues in the code - I suspect that certain outputs where changed
# and the input of other functions never updated.

# This issue was dplyr::laply. Now I use that.

# Find all the cluster labels in the retained MCMC samples
clusterLabels <- unique(clusters %>% as.vector())

sizes <- matrix(nrow = nrow(clusters), ncol = length(clusterLabels))
for (ci in 1:length(clusterLabels)) {
  sizes[, ci] <- rowSums(clusters == clusterLabels[ci])
}
sizes <- sizes %>% as.data.frame()
colnames(sizes) <- clusterLabels

boxplot(sizes, xlab = "Global combined clusters", ylab = "Cluster size")
```



```r
# # Create a matrix to store the cluster membership count in each sample
# sizes <- matrix(nrow = length(clusters), ncol = length(clusterLabels))
#
# for(ii in 1:length(clusters)){
#   for (ci in 1:length(clusterLabels)) {
#     sizes[ii, ci] <- sum(clusters[[ii]] == clusterLabels[[ci]])
#   }
# }
# sizes <- sizes %>% as.data.frame()
# colnames(sizes) <- clusterLabels
#
# # Plot
# boxplot(sizes, xlab = "Global combined clusters", ylab = "Cluster size")
```

Obtaining global clusters

There are several approaches to estimate hard cluster assignments from MCMC samples. If the MCMC chain converged to a stable results, it is possible to use one of the samples as the resulting cluster assignment.

```r
clusteringResult <- samples[[length(samples)]]
```

Co-clustering matrix

A more principled approach is to look at which data points were assigned into the same cluster across the MCMC samples. We can explore this using the posterior co-clustering matrix, which estimates the posterior probability that two samples belong to the same cluster.

```r
# Compute the co-clustering matrix from global cluster assignments
coclust <- coclusteringMatrix(clusters)
```

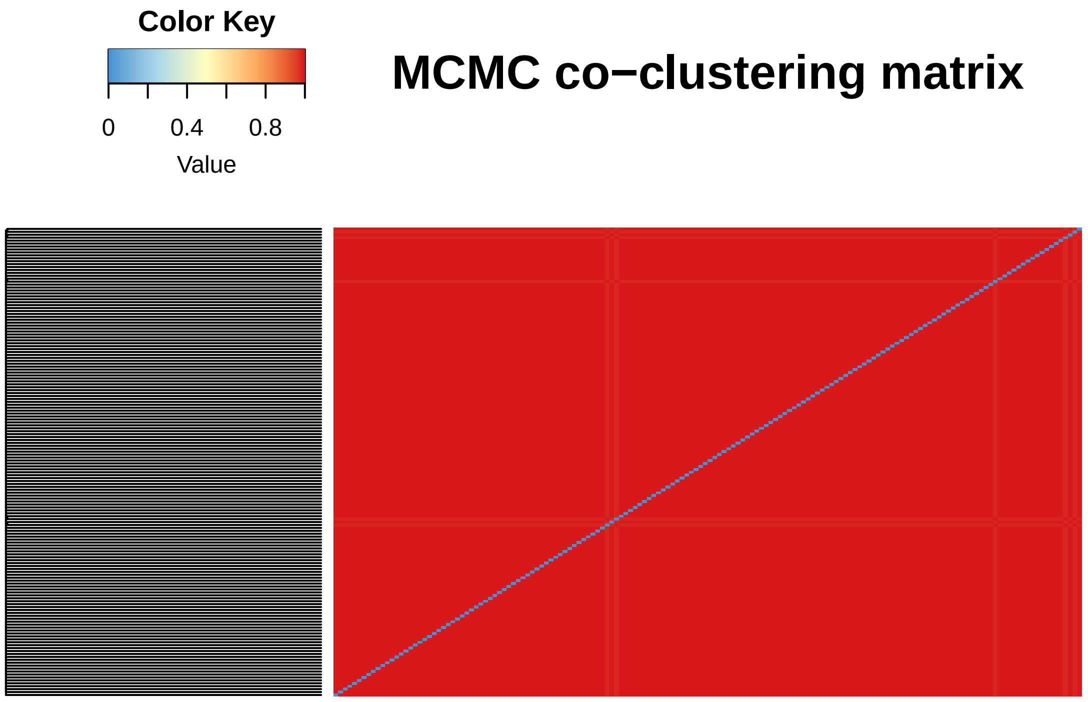
```r
# Plot the co-clustering matrix as a heatmap
require(gplots)
```

```
## Loading required package: gplots
```

```
##
## Attaching package: 'gplots'
```

```
## The following object is masked from 'package:stats':
##
##     lowess
```

```r
mypalette <- colorRampPalette(rev(c("#d7191c", "#fdae61", "#ffffbf", "#abd9e9", "#4395d2")),
  space = "Lab"
)(100)
h <- heatmap.2(
  coclust,
  col = mypalette, trace = "none",
  dendrogram = "row", labRow = "", labCol = "", key = TRUE,
  keysize = 1.5, density.info = c("none"),
  main = "MCMC co-clustering matrix",
  scale = "none"
)
```



We can then use the posterior co-clustering matrix to compute the resulting hard clustering using hierarchical clustering. Note that for this step, we need to specify the number of clusters that we want to obtain. This step should be guided by the posterior number of clusters estimated by the algorithm (see above).

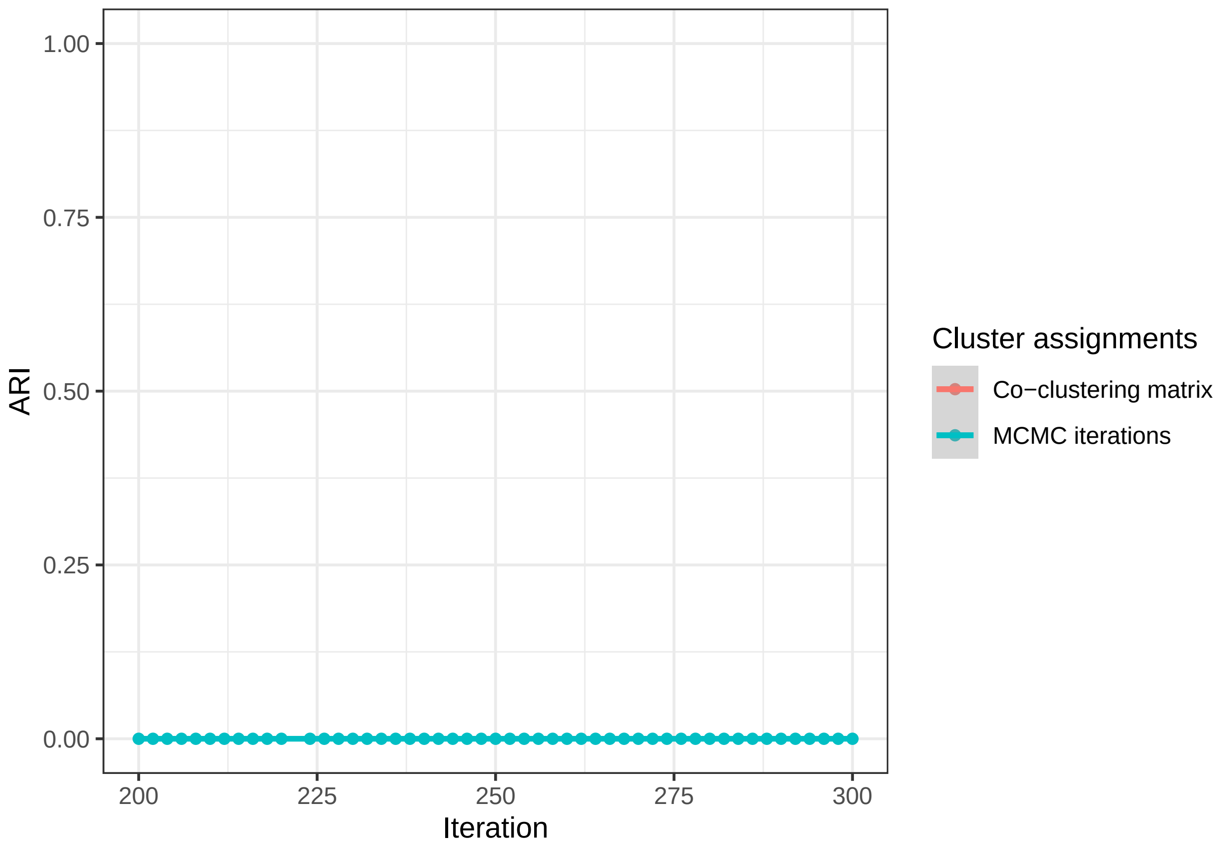
```r
diag(coclust) <- 1
fit <- hclust(as.dist(1 - coclust))
hardAssignments <- cutree(fit, k = 4)
```

Adjusted Rand index

Now we can check if the estimated global cluster assignments correspond to the true assignments that were used to generate the simulated dataset. We use the adjusted Rand index (ARI), which measures how well do two sets of assignments correspond to each other. The following plot shows the ARI across the MCMC iterations, and also the ARI of the result obtained from the co-clustering matrix. ARI equal to 1 represents complete agreement between the estimated assignments and the true clustering, ARI equal to 0 corresponds to random assignments.

```r
# Calculate the adjusted rand index
aris <- laply(
  1:nrow(clusters),
  function(i) {
    mclust::adjustedRandIndex(clusters[i, ], testData$groups)
  }
) %>%
  as.data.frame()

colnames(aris) <- "ARI"
aris$Iteration <- seq(from = burnin, to = maxIter, by = lag)
coclustAri <- mclust::adjustedRandIndex(hardAssignments, testData$groups)
aris$Coclust <- coclustAri

ggplot(aris, aes(x = Iteration, y = ARI, colour = "MCMC iterations")) +
  geom_point() +
  ylim(0, 1) +
  geom_smooth(size = 1) +
  theme_bw() +
  geom_line(aes(x = Iteration, y = Coclust, colour = "Co-clustering matrix"), size = 1) +
  scale_colour_discrete(name = "Cluster assignments")
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'

## Warning: Removed 1 rows containing non-finite values (stat_smooth).

## Warning: Removed 1 rows containing missing values (geom_point).

## Warning: Removed 51 rows containing missing values (geom_path).
```



References

Gabasova, Evelina, John Reid, and Lorenz Wernisch. 2017. "Clusternomics: Integrative Context-Dependent Clustering for Heterogeneous Datasets." *PLoS Computational Biology* 13 (10). Public Library of Science: e1005781.

Green, Peter J, and Sylvia Richardson. 2001. "Modelling Heterogeneity with and Without the Dirichlet Process." *Scandinavian Journal of Statistics* 28 (2). Wiley Online Library: 355–75.

Savage, Richard S, Zoubin Ghahramani, Jim E Griffin, Paul Kirk, and David L Wild. 2013. "Identifying Cancer Subtypes in Glioblastoma by Combining Genomic, Transcriptomic and Epigenomic Data." *arXiv Preprint arXiv:1304.3577*.