Estimating Mixture Complexity with **mixComp**

Anja Weigel

28 April 2020

1 Introduction

The **mixComp** package provides R functions to estimate the *complexity* or *order* of a finite mixture distribution assumed to underlie some data. This vignette describes which estimation methods **mixComp** contains and how they can be applied to either given or randomly generated data.

> **Definition 1** (Mixture distribution)**.** A distribution F is called a *mixture distribution* (or short, *mixture*) if its density has the form
>
> $$f(x) = \int g(x;\theta)\ d\mu(\theta)$$
>
> where μ is a distribution on Θ, called *mixing distribution*, and $\{g(x;\theta) : \theta \in \Theta\}$ is a family of density functions (or probability mass functions[1]) parametrized by some $\theta \in \Theta \subset \mathbb{R}^d$.

As said, the applicability of **mixComp** and thus the scope of this documentation will be limited to *finite* mixture models, which arise if the mixing distribution μ is of a certain form.

> **Definition 2** (Finite mixture distribution)**.** A distribution F is called a *finite mixture* if its density is of the form given in Definition 1 and μ is a discrete distribution that puts masses $(w_1, \dots w_p : \sum w_i = 1)$ on a finite set of p points $(\theta_1, \dots \theta_p : \theta_i \in \Theta\ \forall i)$. Equivalently, $\mu = \sum_{i=1}^{p} w_i \delta_{\theta_i}$ and
>
> $$f(x) = \sum_{i=1}^{p} w_i\ g(x;\theta_i).$$
>
> In this case, the number of components p is called the *mixture complexity* or *order of the mixture*.

From here on it is assumed that the familiy $\{g(x;\theta) : \theta \in \Theta\}$ is known, but its parametrization $\theta \in \Theta^p$, the component weights $w \in \mathbb{R}^p$ and the mixture complexity p are unknown. Assume now that F is a finite mixture distribution with pdf/pmf $f(x) = \sum_{i=1}^{p} w_i g(x;\theta_i)$ and $\mathbf{X} = \{X_1, \dots, X_n\}$ is an i.i.d. sample of size n from F. The goal of this package is to estimate the order p from the sample $\mathbf{X}$. For most functions contained in **mixComp**, this goes hand in hand with estimating the weights w_i and the component distribution parameters θ_i, $i \in 1, \dots, p$.

[1] abbreviated pdf/pmf from here on

2 DatMix objects

2.1 Creation from data

R function	dat	dist	param.bound.list	MLE.function	Hankel.method	Hankel.function
datMix			NULL	NULL	NULL	NULL

Table 1: `datMix` function formals and defaults.

To estimate the mixture complexity with one of the functions contained in **mixComp**, a `datMix` object has to be created. This object contains the data $\mathbf{X}$ as well as other "static" information needed for the estimation procedure (in contrast to "tuning parameters", which can be changed with every function call). For an overview of which attributes need to be supplied for each function, see table 2. For an overview of which functions are restricted in the component distributions and which functions additionally estimate the component weights w_i and parameters θ_i, $i \in 1, \ldots, p$, see table 3. The arguments `dat` and `dist` always have to be specified.

`nonparamHankel`, `paramHankel`, `paramHankel.scaled`

In general, methods based on a Hankel matrix approach need `Hankel.method` and `Hankel.function` as an input, and can only be used on mixtures where estimates of the moments of mixing distribution can be given in closed form (`explicit`) or in the form found in Dacunha-Castelle and Gassiat (1997) equation (3) (`natural`), example (3.1) (`translation`) or example (3.2) (`scale`). If the procedure estimates the mixture parameters (all but `nonparamHankel`), `MLE.function` is used for optimization and initialization, if supplied, and `param.bound.list` has to be specified. For a thorough discussion of these methods, see section 3.1 and 3.2.

`L2.disc`, `L2.boot.disc`, `hellinger.disc`, `hellinger.boot.disc`

These functions can only be used if the component distribution is discrete, and need `param.bound.list` as input. If `MLE.function` is supplied, it is used for initialization of the parameters, which will be estimated in the process. For more details, see section 3.3 and 3.4.

`mix.lrt`

To estimate the mixture complexity (as well as the parameters) via likelihood ratio tests, `param.bound.list` needs to be given as input. The input `MLE.function` is again optional, but used for initialization and optimization if supplied. For further discussion, see section 3.5.

R function	param.bound.list	MLE.function	Hankel.method	Hankel.function
nonparamHankel			x	x
paramHankel(.scaled)	x	o + i (optional)	x	x
L2(.boot).disc	x	i (optional)		
hellinger(.boot).disc	x	i (optional)		
mix.lrt	x	o + i (optional)		

[o] used for optimization
[i] used for initialization

Table 2: Inputs needed for different functions.

R function	distribution restriction	estimation of w and θ
`nonparamHankel`	compatible with `explicit`, `natural`, `translation` or `scale`	
`paramHankel(.scaled)`	compatible with `explicit`, `natural`, `translation` or `scale`	x
`L2(.boot).disc`	discrete distributions	x
`hellinger(.boot).disc`	discrete distributions	x
`mix.lrt`		x

Table 3: Distribution restrictions and outputs for different functions.

As a simple example of observed data to which mixture models may be applied, consider the Old Faithful dataset. The variable `waiting` gives the time in minutes between eruptions of the Old Faithful geyser in the Yellowstone National Park. To estimate the number of components of an underlying mixture distribution via **mixComp**, the data has to be converted to a `datMix` object first. In this case, let it be assumed that the data comes from a normal mixture and all available functions are to be used (i.e. all arguments of `datMix` should be specified).

```
## observations from a (presumed) mixture model
obs <- faithful$waiting

## generate list of parameter bounds (assuming gaussian components)
norm.bound.list <- vector(mode = "list", length = 2)
names(norm.bound.list) <- c("mean", "sd")
norm.bound.list$mean <- c(-Inf, Inf)
norm.bound.list$sd <- c(0, Inf)

## generate MLE functions
# for "mean"
MLE.norm.mean <- function(dat) mean(dat)
# for "sd" (not using the sd function as it uses (n-1) as denominator)
MLE.norm.sd <- function(dat){
n <- length(dat)
var_hat <- (1/n)*sum((dat-mean(dat))^2)
sqrt(var_hat)
}
# combining the functions to a list
MLE.norm.list <- list("MLE.norm.mean" = MLE.norm.mean,
                      "MLE.norm.sd" = MLE.norm.sd)

## function giving the j^th raw moment of the standard normal distribution,
## needed for calculation of the Hankel matrix via the "translation" method
## (assuming gaussian components with variance 1)

mom.std.norm <- function(j){
  if(j %% 2 == 0){
    prod(seq(1, j-1, by = 2))
  }
  else 0
}

## generate 'datMix' object
faithful.dM <- datMix(obs, dist = "norm", param.bound.list = norm.bound.list,
                      MLE.function = MLE.norm.list, Hankel.method = "translation",
                      Hankel.function = mom.std.norm)
```

2.2 Creation from rMix objects

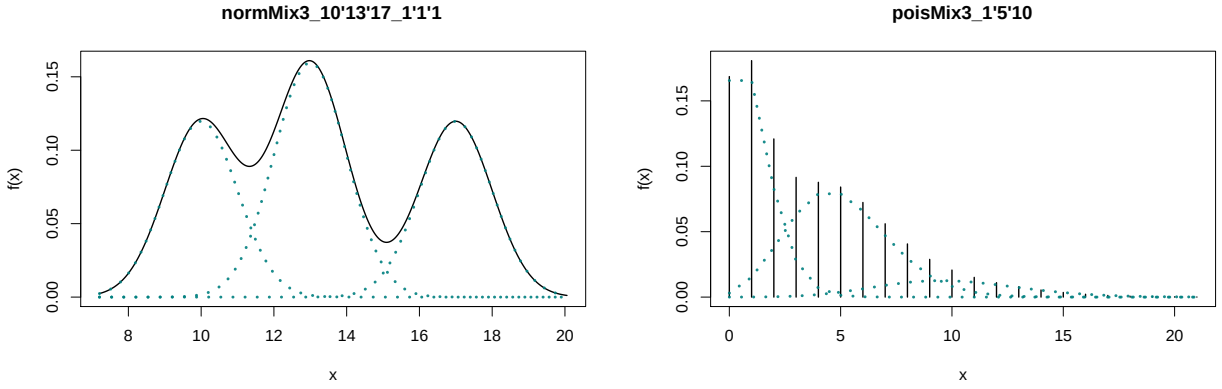
R function	dist	w	param.list	name	...
Mix		NULL	NULL	NULL	

R function	n	obj
rMix		

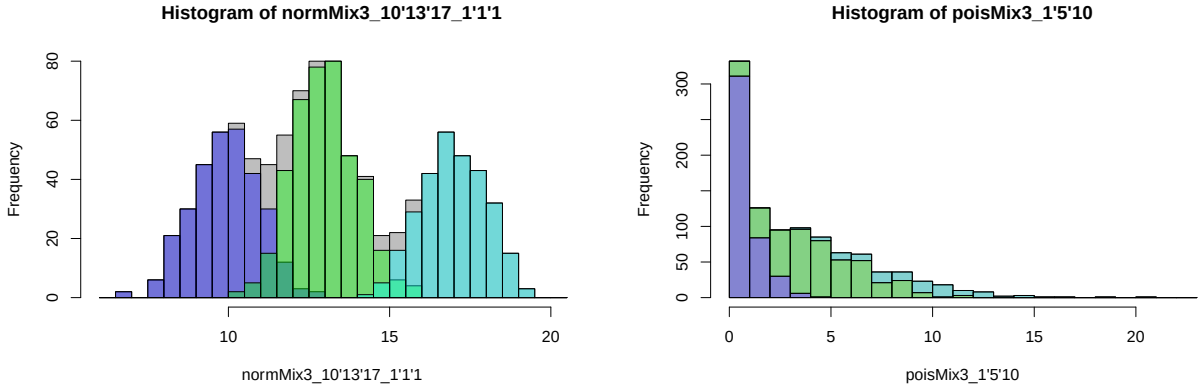
Table 4: `Mix` and `rMix` function formals and defaults.

If the user is interested in simulations rather than existing data, **mixComp** can also be used for generating random data based on some mixture, as a first step. Initially, a `Mix` object has to be created via the `Mix` function, which thereafter can be passed to further functions which plot the respective mixture density or generate a random sample based on the specified distribution. This object contains all information relating solely to the mixture distribution, namely the component distribution `dist`, the component weights `w`, the values of the distribution parameters for each component `param.list` and (optionally) the name of the mixture `name`. If `w` is not supplied, all weights will be taken as equally large. The component parameters can either be supplied as a named list (`param.list`) or via the `...` argument; in both cases it is essential that the names (either of the list elements or of the `...` arguments) match the names of the formals of the density and random number generation function (e.g. from the **stats** package, say for the normal distribution, this means the scale parameter has to be specified as `sd`, rather than `var`, `std.dev` or the likes). The argument `name` will be generated by the `Mix` function itself if left empty by the user.

```
normLocMix <- Mix("norm", w = c(0.3, 0.4, 0.3), mean = c(10, 13, 17), sd = c(1, 1, 1))
poisMix <- Mix("pois", w = c(0.45, 0.45, 0.1), lambda = c(1, 5, 10))

plot(normLocMix, parComp = rmdlist)
plot(poisMix, parComp = rmdlist)
```




```
set.seed(0)
normLocRMix <- rMix(1000, obj = normLocMix)
set.seed(0)
poisRMix <- rMix(1000, obj = poisMix)

plot(normLocRMix)
plot(poisRMix, stacked = TRUE)
```



Using the `rMix` function will generate an `rMix` object, which can be converted to a `datMix` object, if it is supposed to be passed to one of the functions estimating the mixture complexity. Again, relevant attributes have to be passed to the function (i.e. all `datMix` arguments except `dist`, as this is information is already contained in the `rMix` object).

```
normLoc.dM <- RtoDat(normLocRMix, param.bound.list = norm.bound.list,
                     MLE.function = MLE.norm.list, Hankel.method = "translation",
                     Hankel.function = mom.std.norm)
```

3 Complexity Estimation

Once the `datMix` object is created, it can be passed to one of the **mixComp** functions estimating the mixture complexity. All functions but `nonparamHankel` additionally return estimates of the weights w_i and the parameters of the component distributions θ_i. Throughout this discussion of the functions, randomly generated datasets will be used to make it possible to compare the estimation results to the true underlying mixture distribution.

3.1 nonparamHankel

R function	obj	j.max	pen.function	scaled	B	...
nonparamHankel		10	NULL	FALSE	1000	passed to boot

Table 5: `nonparamHankel` function formals and defaults.

Definition 3 (Hankel matrix)**.** For any vector c in $\mathbb{R}^{2k}$, the *Hankel matrix* of c is defined as the $(k+1) \times (k+1)$ matrix given by

$$H(c)_{i,j} = c_{i+j-2}, \qquad 1 \leq i, j \leq k+1,$$

with $c_0 = 1$ by definition.

Let $c^j \in \mathbb{R}^{2j}$ be a vector containing the first $2j$ (raw) moments of the mixing distribution μ, i.e.

$$c_m^j = \int_\Theta \theta^m d\mu(\theta) = \sum_{i=1}^{p} w_i \theta_i^m, \qquad \text{for } m \in \{1, \ldots, 2j\}.$$

`nonparamHankel` estimates mixture complexity p by iteratively increasing the assumed order j and calculating the determinant of the $(j+1) \times (j+1)$ Hankel matrix made up of the first $2j$ moments of the mixing

distribution. As shown by Dacunha-Castelle and Gassiat (1997), once the correct order is reached (i.e. for all $j \geq p$), the determinant of this matrix is zero.

Theorem 1.

$$p = \min\{j \geq 1 : \det H(c^j) = 0\}$$

□

This suggests an estimation procedure for p based on initially finding a consistent estimator of the moments of the mixing distribution and then choosing $\hat{p}$ as the value of j which yields a sufficiently small value of the determinant. Since the determinant is close to 0 for all $j \geq p$, this could lead to choosing $\hat{p}$ rather larger than the true value. The function therefore returns all estimated determinant values corresponding to complexities up to `j.max`, so that the user can decide from which point on the determinant is small enough. It is also possible to include a penalty term as a function of the sample size `n` and the current assumed complexity `j` which will be added to the determinant value (by supplying `pen.function`), and/or to scale the determinants (by setting `scaled = TRUE`). For scaling, a nonparametric bootstrap is used to calculate the covariance of the estimated determinants (with `B` being the size of the bootstrap samples). The inverse of the squareroot of this covariance matrix is then multiplied with the estimated determinant vector to get the scaled determinant vector.

Four methods of estimating the moments of the mixing distribution are implemented in **mixComp**. The method to be used is specified when creating the `datMix` object via the argument `Hankel.method`. To each method corresponds a function (or multiple), which is specified via `Hankel.function`, again when creating the `datMix` object.

1. `Hankel.method = "natural"`

   Dacunha-Castelle and Gassiat (1997), page 283 equation (3) states that if

   $$c^j = f_j(\mathbb{E}[\psi_j(X_i)]),$$

   one may take as an estimator

   $$\hat{c}^j = f_j(\frac{1}{n}\sum_{i=1}^{n}\psi_j(X_i)),$$

   which will be called the `"natural"` estimator of c^j. To make use of this method, the functions ψ_j and f_j have to be entered as a list to `Hankel.function`, with ψ_j as first element and f_j as second element. The function ψ_j has the data vector as first argument and j as second, and should give back the vector $\psi_j(X_i), 1 \leq i \leq n$. f_j contains the average of $\psi_j(X_i)$ as first argument and j as second (even if it is unused in the function body). If only one function is supplied this will be taken as ψ_j, and f_j will be taken to be the identity function.

   Take, as an example, a mixture of poisson distributions, where

   $$\lambda^j = \mathbb{E}[X(X-1)\dots(X-j+1)].$$

   Then f_j simply is the identity function and $\psi_j(X) = X(X-1)\dots(X-j+1)$.

```
psi.pois <- function(dat, j){
  res <- 1
  for (i in 0:(j-1)){
    res <- res*(dat-i)
  }
  res
}
```

2. `Hankel.method = "explicit"`

   For this method, `Hankel.function` contains a single function which explicitly estimates the moments of the mixing distribution. Note that `"natural"` is equivalent to using `"explicit"` with `Hankel.function` $f_j(\frac{1}{n}\sum_{i=1}^{n}(\psi_j(X_i)))$ (i.e. `f(mean(psi(dat, j)), j)`). As an example, take a mixture of geometric distributions, where it can be shown that

$$c^j = 1 - \sum_{l=0}^{j-1} f(l)$$

   and one may take

$$\hat{c}^j = 1 - \hat{F}_n(j-1)$$

   as an estimator, with $\hat{F}_n$ being the empirical estimator.

```
explicit.geom <- function(dat, j){
  n <- length(dat)
  res <- numeric(j)
  for(l in 0:(j-1)){
    res[l+1] <- sum(ifelse(dat == l, 1, 0))
  }
  return(1 - sum(res/n))
}
```

3. `Hankel.method = "translation"`

   Dacunha-Castelle and Gassiat (1997), page 284 example 3.1. describes how to estimate c^j if the family of component distributions (G_θ) is given by $\mathrm{d}G_\theta(x) = \mathrm{d}G(x-\theta)$, where the moments of G are known. In this case, a triagular linear system can be solved for $\hat{c}^j$, using the empirical moments of the mixture distribution and the known moments of G. So, to apply this method, `Hankel.function` contains a function of j, giving back the moments of G.

   As an example, consider a mixture of normal distributions with unknown mean and variance equal to 1. Then G is the standard normal distribution, and its j^{th} moment M^j is defined as

$$M^j = \begin{cases} 0 & \text{if } j \text{ uneven} \\ (j-1)!! & \text{if } j \text{ even} \end{cases}$$

```
mom.std.norm <- function(j){
  if(j %% 2 == 0){
    prod(seq(1, j-1, by = 2))
  }
  else 0
}
```

4. `Hankel.method = "scale"`

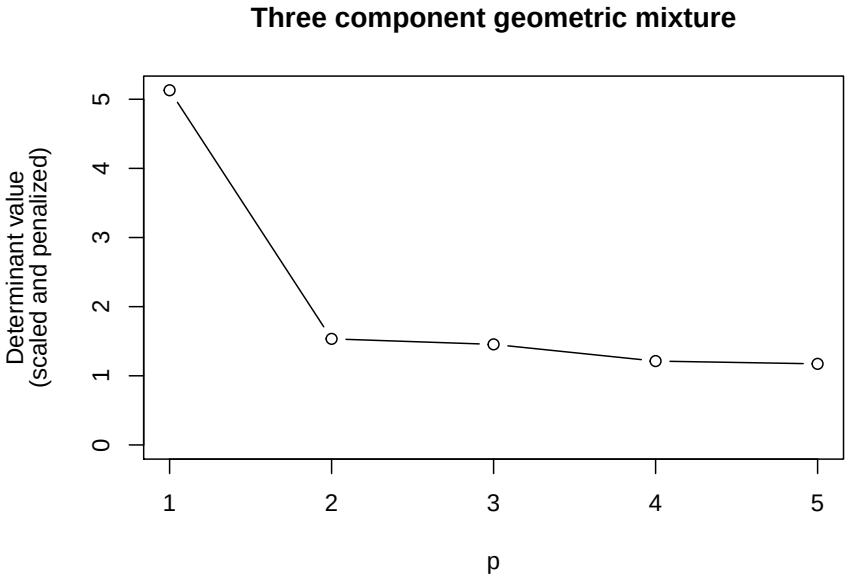
   Similar to the `"translation"` method, Dacunha-Castelle and Gassiat (1997), page 284 example 3.2. describes how to estimate c^j if the family of component distributions (G_θ) is given by $\mathrm{d}G_\theta(x) = \mathrm{d}G(\frac{x}{\theta})$, where again the moments of G are known. As above, a triagular linear system can be solved for $\hat{c}^j$, using the empirical moments of the mixture distribution and the known moments of G, and `Hankel.function` contains a function of j giving back the moments of G. The only difference is that squares have to be taken everywhere if for some integer k, $M^k = 0$.

As an example, consider a mixture of normal distributions with zero mean and unknown variance. Again, G is the standard normal distribution, and its j^{th} moment is defined as above.

```
## create 'Mix' object
geomMix <- Mix("geom", w = c(0.1, 0.6, 0.3), prob = c(0.8, 0.2, 0.4))

## create random data based on 'Mix' object (gives back 'rMix' object)
set.seed(1)
geomRMix <- rMix(1000, obj = geomMix)

## create 'datMix' object for estimation (using the function defined above)
geom.dM <- RtoDat(geomRMix, Hankel.method = "explicit", Hankel.function = explicit.geom)

## function for penalization
pen <- function(j, n){
  (j*log(n))/(sqrt(n))
}

## estimate determinants
set.seed(1)
geomdets_sca_pen <- nonparamHankel(geom.dM, scaled = TRUE, pen.function = pen, j.max = 5)
print(geomdets_sca_pen)
```

```
##
## Estimation of the scaled and penalized determinants for a 'geom' mixture model:
##
##  Number of components Determinant
##                     1    5.129757
##                     2    1.534127
##                     3    1.454279
##                     4    1.212697
##                     5    1.172762
```

```
plot(geomdets_sca_pen, main = "Three component geometric mixture")
```



3.2 paramHankel, paramHankel.scaled

R function	obj	j.max	B	ql	qu	control	...
paramHankel		10	1000	0.025	0.975	c(trace = 0)	passed to boot

R function	obj	j.max	B	ql	qu	control	...
paramHankel.scaled		10	100	0.025	0.975	c(trace = 0)	passed to boot

Table 6: paramHankel and paramHankel.scaled function formals and defaults.

paramHankel estimates the mixture complexity p by iteratively increasing the assumed order j and calculating the determinant of the $(j+1) \times (j+1)$ Hankel matrix made up of the first $2j$ raw moments of the mixing distribution (for details see section 3.1). Then, for a given j, the MLE for a j component mixture $(\hat{w}_j, \hat{\theta}_j)$ is calculated and B parametric bootstrap samples of size n are generated from the distribution corresponding to $(\hat{w}_j, \hat{\theta}_j)$. For each of the B samples the determinant of the resulting $(j+1) \times (j+1)$ Hankel matrix is calculated, and the original determinant value is compared to the bootstrap quantiles ql and qu. If the original determinant lies within this range, j is returned as the order estimate $\hat{p}$, otherwise j is increased by 1 and the procedure is started over.

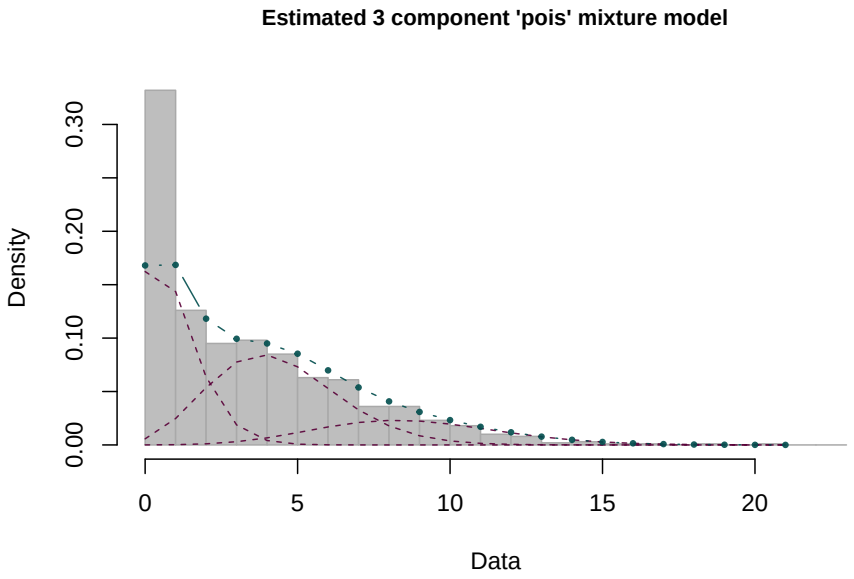
paramHankel.scaled does the same as paramHankel with the exception that the determinants are devided by their standard deviation. For the bootstrapped determinants, this denominator is simply calculated as the empirical standard deviation of the bootstrap sample. For the original determinant, B nonparametric bootstrap samples of size n are generated from the data, the corresponding determinants are calculated and their empirical standard deviation is used.

The MLEs $(\hat{w}_j, \hat{\theta}_j)$ are calculated via the MLE.function attribute for $j = 1$ (and thus $\hat{w}_j = 1$), if it is supplied. For all other j (and also for $j = 1$ in case MLE.function = NULL) the solver solnp is used to calculate the minimum of the negative log likelihood. As initial values (for solnp), the data is clustered into j groups via the function clara and the data corresponding to each group is given to MLE.function (if supplied, otherwise numerical optimization is used here as well). The size of the groups is taken as initial component weights $\hat{w}_j^{\text{init}}$ and the MLEs are taken as initial parameter estimates $\hat{\theta}_j^{\text{init}}$.

```r
## create 'datMix' object for estimation

# generate list of parameter bounds
poisList <- vector(mode = "list", length = 1)
names(poisList) <- "lambda"
poisList$lambda <- c(0, Inf)

# generate MLE function
MLE.pois <- function(dat){
  mean(dat)
}

# generating 'datMix' object
pois.dM <- RtoDat(poisRMix, param.bound.list = poisList, MLE.function = MLE.pois,
                  Hankel.method = "natural", Hankel.function = psi.pois)


## complexity and parameter estimation
set.seed(1)
res <- paramHankel(pois.dM)
```

```
##
## Parameter estimation for a 1 component 'pois' mixture model:
## Function value: 2898.6059
##
##              w lambda
## Component 1: 1   3.73
## Optimization via user entered MLE-function.
## ------------------------------------------------------------------------
## Parameter estimation for a 2 component 'pois' mixture model:
## Function value: 2434.4711
##
##                     w lambda
## Component 1: 0.52043 1.2690
## Component 2: 0.47957 6.4006
## Converged in 3 iterations.
## ------------------------------------------------------------------------
## Parameter estimation for a 3 component 'pois' mixture model:
## Function value: 2410.5443
##
##                     w lambda
## Component 1: 0.39298 0.8844
## Component 2: 0.43707 4.3396
## Component 3: 0.16995 8.7420
## Converged in 3 iterations.
## ------------------------------------------------------------------------
## The estimated order is 3.
```

```r
plot(res)
```



3.3 L2.disc, L2.boot.disc

R function	obj	j.max	n.inf	treshold	control
L2.disc		10	1000	"LIC"	c(trace = 0)

R function	obj	j.max	n.inf	B	ql	qu	control	...
L2.boot.disc		10	1000	100	0.025	0.975	c(trace = 0)	passed to boot

Table 7: `L2.disc` and `L2.boot.disc` function formals and defaults.

`L2.disc` estimates the mixture complexity p by iteratively increasing the assumed order j and finding the "best" estimate for both, the density of a mixture with j and $j+1$ components, by calculating the parameters that minimize the squared L2 distances to the empirical mass function $\hat{f}_n$ (these two procedures only work for *discrete* data). Given probability mass functions g and f, the square of the L2 distance is defined as

$$L_2^2(g, f) = \sum_{x=0}^{\infty} (g(x) - f(x))^2.$$

So, the procedure finds $\hat{w}_j \in \mathbb{R}^j, \hat{\theta}_j \in \Theta^j$ and $\hat{w}_{j+1} \in \mathbb{R}^{j+1}, \hat{\theta}_{j+1} \in \Theta^{j+1}$ defined as

$$(\hat{w}_j, \hat{\theta}_j) = \underset{(\mathrm{w}_j, \vartheta_j)}{\arg\min}\, L_2^2(f_{(\mathrm{w}_j, \vartheta_j)}, \hat{f}_n) = \underset{(\mathrm{w}_j, \vartheta_j)}{\arg\min} \sum_{x=0}^{\infty} f_{(\mathrm{w}_j, \vartheta_j)}^2(x) - \frac{2}{n} \sum_{i=1}^{n} f_{(\mathrm{w}_j, \vartheta_j)}(X_i)$$

$$(\hat{w}_{j+1}, \hat{\theta}_{j+1}) = \underset{(\mathrm{w}_{j+1}, \vartheta_{j+1})}{\arg\min}\, L_2^2(f_{(\mathrm{w}_{j+1}, \vartheta_{j+1})}, \hat{f}_n) = \underset{(\mathrm{w}_{j+1}, \vartheta_{j+1})}{\arg\min} \sum_{x=0}^{\infty} f_{(\mathrm{w}_{j+1}, \vartheta_{j+1})}^2(x) - \frac{2}{n} \sum_{i=1}^{n} f_{(\mathrm{w}_{j+1}, \vartheta_{j+1})}(X_i)$$

Once these parameters are obtained, the difference in squared distances is compared to a predefined `threshold`. This threshold can either be set to `"LIC"` or `"SBC"`, defined as

$$\text{LIC} = \frac{0.6}{n} \ln\left(\frac{j+1}{j}\right) \qquad \text{and} \qquad \text{SBC} = \frac{0.6\ln(n)}{n} \ln\left(\frac{j+1}{j}\right).$$

If the difference is smaller than the selected threshold, the algorithm terminates and the true order is estimated as j, otherwise j is increased by 1 and the procedure is started over (for details, see Umashanger and Sriram (2009)).

`L2.boot.disc` does the same as `L2.disc` with the exception that the difference is not compared to a predefined threshold but to a value generated by bootstrap. In every iteration (of j), a parametric bootstrap is used to generate `B` samples of size n from a j component mixture (given the previously calculated "best" parameter values $(\hat{w}_j, \hat{\theta}_j)$). For each of the bootstrap samples, again the "best" estimates corresponding to densities with j and $j+1$ components are calculated, as well as the difference in squared L2 distances from the empirical mass function. The original difference in squared distances is then compared to the `ql` and `qu` quantiles of the bootstrapped differences; if it lies within this range, j is returned as the order estimate $\hat{p}$, otherwise j is increased by 1 and the procedure is started over.

To calculate the minimum of the squared L2 distance (and the corresponding parameter values), the solver `solnp` is used. As initial values, the data is clustered into j groups via `clara` and the data corresponding to each group is given to `MLE.function` (if this attribute is supplied via the `datMix` object, otherwise numerical optimization is used here as well). The size of the groups is taken as initial component weights $\hat{w}_j^{\text{init}}$ and the MLEs are taken as initial parameter estimates $\hat{\theta}_j^{\text{init}}$. The same initialization procedure is done for optimizing over mixtures with $j+1$ components. As the distance measure contains an infinite sum, the user has the option to specify up to which cutoff this sum should be calculated (`n.inf`, with default `1000`).

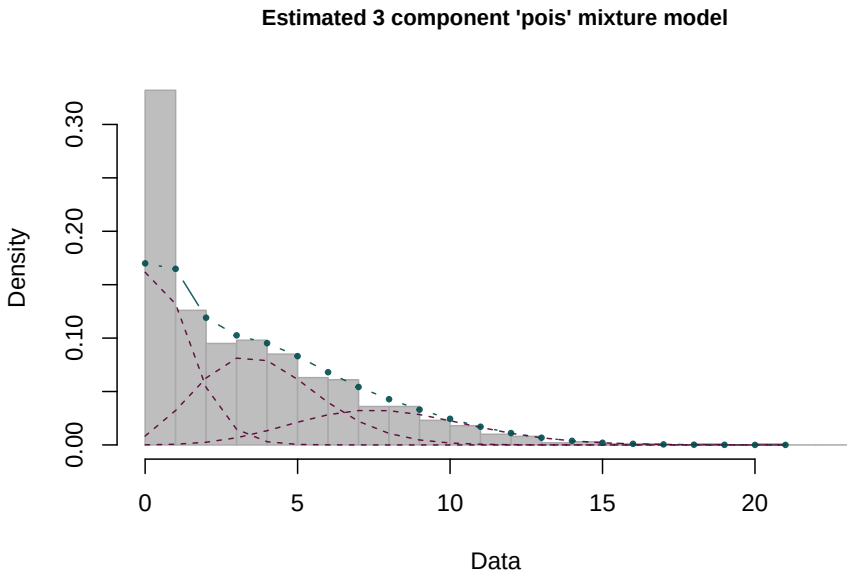
```
## complexity and parameter estimation with previously defined 'datMix' object
## without bootstrap
set.seed(1)
res <- L2.disc(pois.dM)
```

```
##
## Parameter estimation for a 1 component 'pois' mixture model:
## Function value: -0.05849
##
##              w lambda
## Component 1: 1 2.5874
## Converged in 3 iterations.
...
```

```
print(res)
```

```
##
## Parameter estimation for a 3 component 'pois' mixture model:
## Function value: -0.1079
##
##                    w lambda
## Component 1: 0.36560 0.8164
## Component 2: 0.40464 3.8865
## Component 3: 0.22976 7.9718
## Converged in 3 iterations.
##
## The estimated order is 3.
```

```
plot(res)
```




```
## complexity and parameter estimation with previously defined 'datMix' object
## with bootstrap
set.seed(1)
res <- L2.boot.disc(pois.dM, B = 30)
```

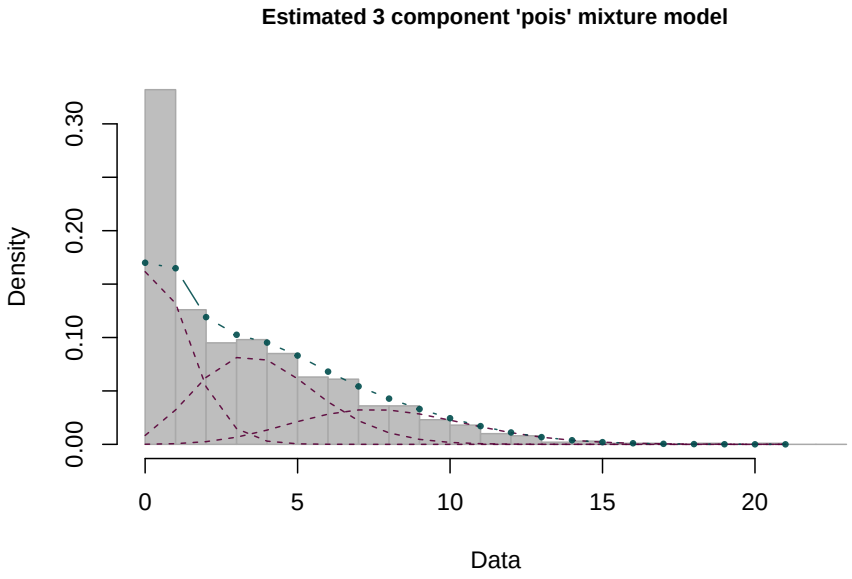
```
##
## Parameter estimation for a 1 component 'pois' mixture model:
## Function value: -0.05849
##
##              w lambda
## Component 1: 1 2.5874
## Converged in 3 iterations.
## ------------------------------------------------------------------------
## Parameter estimation for a 2 component 'pois' mixture model:
## Function value: -0.1067
##
##                    w lambda
## Component 1: 0.45384 1.0093
## Component 2: 0.54616 5.3764
## Converged in 3 iterations.
## ------------------------------------------------------------------------
## Running bootstrap iteration 1 testing for 1 components.
## Running bootstrap iteration 2 testing for 1 components.
## Running bootstrap iteration 3 testing for 1 components.
## Running bootstrap iteration 4 testing for 1 components.
## Running bootstrap iteration 5 testing for 1 components.
## Running bootstrap iteration 6 testing for 1 components.
...
```

```
print(res)
```

```
##
## Parameter estimation for a 3 component 'pois' mixture model:
## Function value: -0.1079
```

```
##
##                      w lambda
## Component 1: 0.36560 0.8164
## Component 2: 0.40461 3.8864
## Component 3: 0.22979 7.9715
## Converged in 3 iterations.
##
## The estimated order is 3.
```

```
plot(res)
```



3.4 hellinger.disc, hellinger.boot.disc

R function	obj	j.max	treshold	control
`hellinger.disc`		10	`"AIC"`	`c(trace = 0)`

R function	obj	j.max	B	ql	qu	control	...
`hellinger.disc`		10	100	0.025	0.975	`c(trace = 0)`	passed to `boot`

Table 8: `hellinger.disc` and `hellinger.boot.disc` function formals and defaults.

These two functions work the same as `L2.disc` and `L2.boot.disc` (see section 3.3), however, a different measure of distance and different thresholds are implemented. As the name suggests, these procedures are based on the square of the hellinger distance between two probability mass functions f and g, defined as

$$H^2(f,g) = \sum_{x=0}^{\infty} \left(\sqrt{f(x)} - \sqrt{g(x)}\right)^2 = 2 - 2\sum_{x=0}^{\infty} \left(\sqrt{f(x)}\sqrt{g(x)}\right).$$

The two thresholds that are implemented are given by

$$\text{AIC} = \frac{2}{n} \qquad \text{and} \qquad \text{SBC} = \frac{\ln(n)}{n}.$$

For more details, see Woo and Sriram (2007).

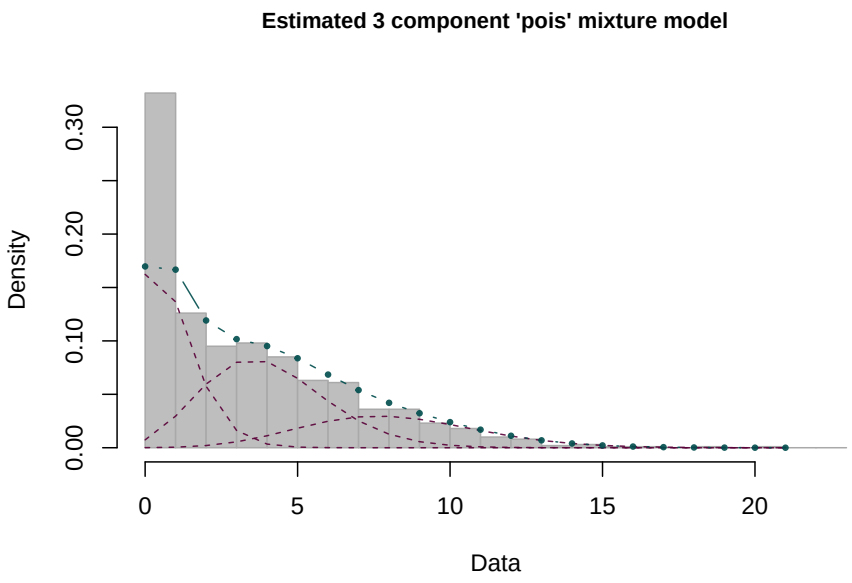
```
## complexity and parameter estimation with previously defined 'datMix' object
## without bootstrap
set.seed(1)
res <- hellinger.disc(pois.dM)
```

```
##
## Parameter estimation for a 1 component 'pois' mixture model:
## Function value: 0.1764
##
##              w lambda
## Component 1: 1 3.4196
## Converged in 3 iterations.
...
```

```
print(res)
```

```
##
## Parameter estimation for a 3 component 'pois' mixture model:
## Function value: 0.003054
##
##                    w lambda
## Component 1: 0.37658 0.8418
## Component 2: 0.41250 4.0282
## Component 3: 0.21092 8.1478
## Converged in 3 iterations.
##
## The estimated order is 3.
```

```
plot(res)
```




```
## complexity and parameter estimation with previously defined 'datMix' object
## with bootstrap
```

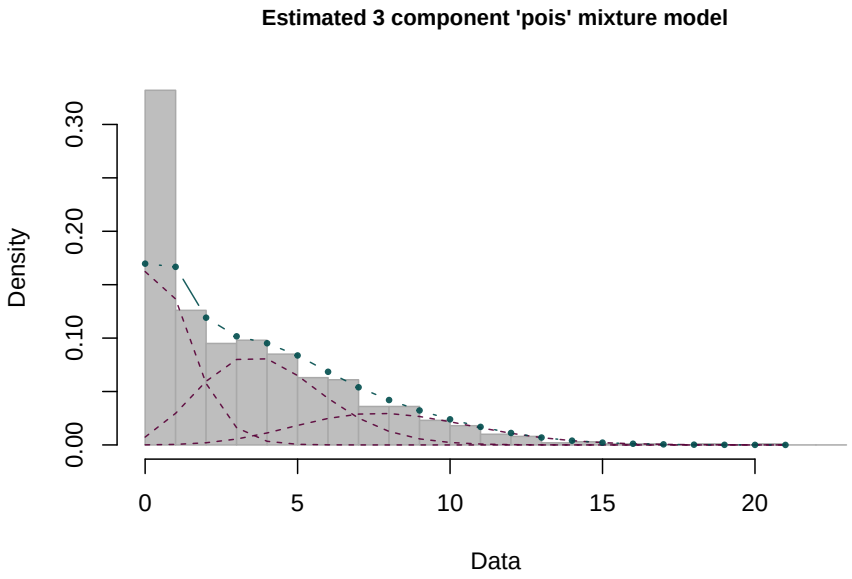
```r
set.seed(1)
res <- hellinger.boot.disc(pois.dM, B = 30)
```

```
##
## Parameter estimation for a 1 component 'pois' mixture model:
## Function value: 0.1764
##
##              w lambda
## Component 1: 1 3.4196
## Converged in 3 iterations.
## ------------------------------------------------------------------------
## Parameter estimation for a 2 component 'pois' mixture model:
## Function value: 0.01211
##
##                     w lambda
## Component 1: 0.51159 1.2246
## Component 2: 0.48841 6.2068
## Converged in 3 iterations.
## ------------------------------------------------------------------------
## Running bootstrap iteration 1 testing for 1 components.
## Running bootstrap iteration 2 testing for 1 components.
## Running bootstrap iteration 3 testing for 1 components.
## Running bootstrap iteration 4 testing for 1 components.
## Running bootstrap iteration 5 testing for 1 components.
## Running bootstrap iteration 6 testing for 1 components.
...
```

```r
print(res)
```

```
##
## Parameter estimation for a 3 component 'pois' mixture model:
## Function value: 0.003054
##
##                     w lambda
## Component 1: 0.37658 0.8418
## Component 2: 0.41249 4.0282
## Component 3: 0.21092 8.1478
## Converged in 3 iterations.
##
## The estimated order is 3.
```

```r
plot(res)
```



3.5 mix.lrt

R function	obj	j.max	B	quantile	control	...
mix.lrt		10	100	0.95	c(trace = 0)	passed to boot

Table 9: `mix.lrt` function formals and defaults.

`mix.lrt` estimates the mixture complexity p by iteratively increasing the assumed order j, finding the maximum likelihood estimator (MLE) for both, the density of a mixture with j and $j+1$ components (giving $(\hat{w}_j, \hat{\theta}_j)$) and $(\hat{w}_{j+1}, \hat{\theta}_{j+1})$)), and calculating the corresponding likelihood ratio test statistic (LRTS).

$$\text{LRTS} = -2 \ln \left(\frac{L_0}{L_1} \right) \quad \text{, with}$$

$$L_0 = L_{\mathbf{X}}(\hat{w}_j, \hat{\theta}_j) \qquad \text{and} \qquad L_1 = L_{\mathbf{X}}(\hat{w}_{j+1}, \hat{\theta}_{j+1}),$$

$L_{\mathbf{X}}$ being the likelihood function given data $\mathbf{X}$.

Then, a parametric bootstrap is used to generate `B` samples of size n from a j component mixture (given the previously calculated MLE $(\hat{w}_j, \hat{\theta}_j)$). For each of the bootstrap samples, again the MLEs corresponding to densities of mixtures with j and $j+1$ components are calculated, as well as the LRTS. The original LRTS is then compared to the `quantile` quantile of the bootstrapped counterparts; if it lies within this range, j is returned as the order estimate $\hat{p}$, otherwise j is increased by 1 and the procedure is started over (for details, see Xekalaki and Karlis (1999)).

The MLEs are calculated via the `MLE.function` attribute for $j = 1$, if it is supplied. For all other j (and also for $j = 1$ in case `MLE.function = NULL`) the solver `solnp` is used to calculate the minimum of the negative log likelihood. As initial values (for `solnp`), the data is clustered into j groups via `clara` and the data corresponding to each group is given to `MLE.function` (if this attribute is supplied via the `datMix` object, otherwise numerical optimization is used here as well). The size of the groups is taken as initial component weights $\hat{w}_j^{\text{init}}$ and the MLEs are taken as initial parameter estimates $\hat{\theta}_j^{\text{init}}$. The same initialization procedure is done for optimizing over mixtures with $j+1$ components.

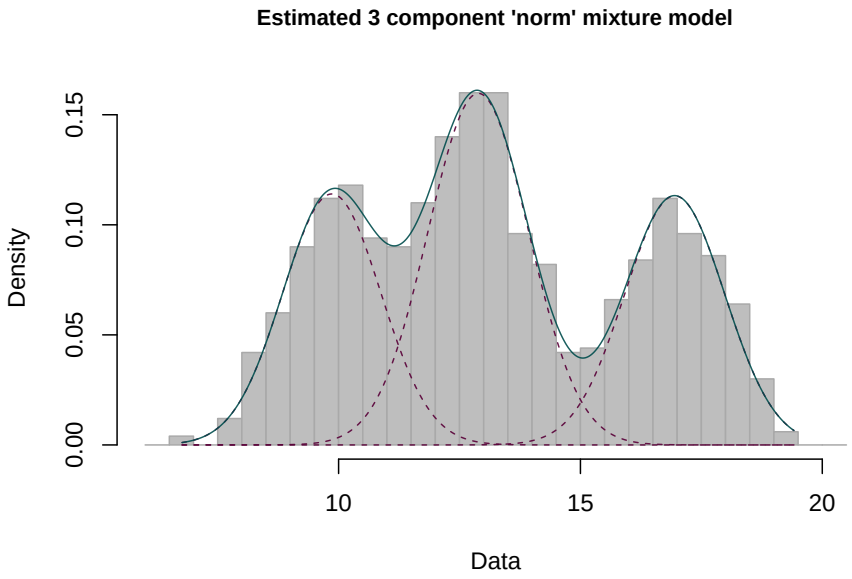
```
## using already generated 'datMix' object to estimate the mixture

set.seed(0)
res <- mix.lrt(normLoc.dM, B = 30)
```

```
##
## Parameter estimation for a 1 component 'norm' mixture model:
## Function value: 2486.3396
##
##                 w mean     sd
## Component 1:  1.0 13.2 2.9078
## Optimization via user entered MLE-function.
## ------------------------------------------------------------------------
## Parameter estimation for a 2 component 'norm' mixture model:
## Function value: 2392.6800
##
##                     w     mean     sd
## Component 1:  0.74923 11.88077 2.0151
## Component 2:  0.25077 17.14082 0.9251
## Converged in 3 iterations.
## ------------------------------------------------------------------------
## Running bootstrap iteration 1 testing for 1 components.
## Running bootstrap iteration 2 testing for 1 components.
## Running bootstrap iteration 3 testing for 1 components.
## Running bootstrap iteration 4 testing for 1 components.
## Running bootstrap iteration 5 testing for 1 components.
## Running bootstrap iteration 6 testing for 1 components.
...
```

```
print(res)
```

```
##
## Parameter estimation for a 3 component 'norm' mixture model:
## Function value: 2364.6794
##
##                     w     mean     sd
## Component 1:  0.28987  9.85917 1.0143
## Component 2:  0.41724 12.88906 1.0422
## Component 3:  0.29289 16.94882 1.0324
## Converged in 3 iterations.
##
## The estimated order is 3.
```

```
plot(res)
```



4 References

Dacunha-Castelle, Didier, and Elisabeth Gassiat. 1997. "The Estimation of the Order of a Mixture Model." *Bernoulli* 3 (3). Bernoulli Society for Mathematical Statistics; Probability: 279–99. https://projecteuclid.org:443/euclid.bj/1177334456.

Umashanger, T., and T.N. Sriram. 2009. "L2E Estimation of Mixture Complexity for Count Data." *Computational Statistics & Data Analysis* 53 (12): 4243–54. doi:https://doi.org/10.1016/j.csda.2009.05.013.

Woo, Mi-Ja, and T.N. Sriram. 2007. "Robust Estimation of Mixture Complexity for Count Data." *Computational Statistics & Data Analysis* 51 (9): 4379–92. doi:https://doi.org/10.1016/j.csda.2006.06.006.

Xekalaki, Evdokia, and Dimitris Karlis. 1999. "On Testing for the Number of Components in a Mixed Poisson Model." *Annals of the Institute of Statistical Mathematics* 51 (February): 149–62. doi:10.1023/A:1003839420071.