thmm Package

July 1, 2017

Type Package

Title a tiny (but robust) discrete time univariate HMM library

Version 1.1

Date 2017-11-18

Author Alain Viari

Maintainer `<alain.viari@inria.fr>`

Description a small HMM library for discrete time univariate models
with careful handling of underflow.

License GPL

Depends methods

LazyData TRUE

RoxygenNote 5.0.1

R topics documented:

`mstep.dnorm` *M-step function for univariate normal distribution*

Description

used as the M-step function of thmm.baumwelch for univariate normal distribution.

Usage

```
mstep.dnorm(obs, cond, ctrl, mean, sd)
```

Arguments

`obs`	numerical vector of observations
`cond`	estimates of the conditional expectations as computed by thmm.estep
`ctrl`	list of control parameters created by thmm.bw.ctrl
`mean`	list of states means (may be missing)
`sd`	list of states sd (may be missing)

Value

list of updated parameters mean and sd

See Also

thmm.baumwelch

`print.DTHmm` *print method for DTHmm*

Description

print method for DTHmm

Usage

```
## S3 method for class 'DTHmm'
print(x, ...)
```

Arguments

`x`	a DTHmm object
`...`	further arguments passed to or from other methods

Value

invisible(x)

`thmm` *Overview of package thmm*

Description

A tiny R package for discrete time hidden Markov models (DTHmm) with careful handling of underflow.
Strongly inspired by the 'HiddenMarkov' package from David Harte (https://cran.r-project.org/web/packages/HiddenMarkov/index.html) but numerically more stable (for forward/backward computations) and with different (hopefully simpler) API. (the 'HiddenMarkov' package has more features (including Markov modulated GLMs))

Concepts: A discrete time hidden Markov model is defined by:

- m (discrete) states
- a probability density function `density` describing the (univariate or multivariate, continuous or discrete) distribution with different parameters for each state.
- a transition probability matrix (of size m x m) between states
- a vector (of length m) of initial probability for each state

important note: for the sake of simplicty the remaining of this documentation as well as functions documentation, will assume that `density` is univariate. However the library can be used with multivariate densities as well (of course this comes at a price of a bit of complication). Please consult the **multivariate densities** section below.

Given a sequence of observed (univariate) random variable `X` noted `obs=obs_1, ..., obs_n` this package solves the following problems:

1. Given the model parameters, compute the probability of `obs`. This problem is solved by the **forward and backward** algorithms.
2. Given the model parameters, find the most likely sequence of (hidden) states which could have generated `obs`. This problem is solved by the **Viterbi** algorithm.
3. Given `obs`, find the most likely set of `density` parameters and transition probabilities. This problem is solved by the **Baum-Welch** algorithm.

forward algorithm: is used to compute:

- `alpha_{i,j} = Pr{ X_1 = obs_1, ..., X_i = obs_i, state_i = j | hmm}` that is the probability of seeing the partial sequence `(obs_1, ..., obs_i)` and ending up in state j at time i for this hmm.

from which one can derive the **log-likelihood** of observation `obs` with this hmm:

- `log(Pr{ X_1 = obs_1, ..., X_n = obs_n | hmm})`

backward algorithm: is used to compute:

- `beta_{i,j} = Pr{ X_{i+1} = obs_{i+1}, ..., X_n = obs_n | state_i = j , hmm}` that is the probability of the ending partial sequence `(obs_i+1, ..., obs_n)` given that we started at state j at time i, for this hmm

forward-backward algorithm: putting everything together, one can further compute:

- `gamma_{i,j} = Pr{ state_i = j | obs , hmm}` that is the probability of being at time i in state j given this observation and this hmm

- `rho_{i} = Pr{ X_1 = obs_1, ..., X_i = obs_i | hmm}` that is the probability of seeing the partial sequence `(obs_1, ..., obs_i)` with this hmm

Viterbi's algorithm: The purpose of the Viterbi's algorithm is to determine the sequence of states `(k_1*, ..., k_n*)` which maximises the joint distribution of the hidden states given the entire observed process. i.e:

- `(k_1*, ..., k_n*) = argmax(Pr(state_1=k_1, ..., state_n=k_n, X_1=obs_1, ..., X_n=obs_n | hmm)`

this also allows to compute:

- `nu_{i+1,j} = Pr(state_1=k_1*, ..., state_i=k_i*, state_i+1 = j,  X_1=obs_1, ..., X_i=obs_i |` where k_i* is the optimal state at time i. (local decoding)
- `viterbi_score = Pr(state_1=k_1*, ..., state_n=k_n*, obs | hmm)` the joint probability of optimal sequence of states and observation.
- `probseq  = Pr(state_1=k_1*, ..., state_n=k_n* | obs, hmm)` the probability of the optimal sequence of state conditionally to the observation.

Baum-Welch algorithm: The purpose of the Baum-Welch algorithm is to estimate the HMM parameters (i.e. distribution parameters and transition probabilities) that best fit a given observation. This is a version of the EM algorithm that iteratively alternates two steps: the **E-step** and the **M-step**. The E-step is generic but the M-step depends upon each distribution. A M-step for Normal distribution is provided (as `mstep.dnorm`) and some others are given in examples. See thmm.baumwelch for more details (in particular how to control which parameters are actually adjusted).

Implementation: current implementation uses either pure R code or compiled C code. the default is to use (quicker) C code, R code has been kept for debugging purposes.

Multivariate densities: Instead of a univariate density function you may define a bi- or multivariate density function as well. The main difference is that instead of a numerical vector of observations, you should provide a list of tuples (each element in the tuple corresponds to one of the random variable). For the bivariate case, however, a trick is to use complex numbers instead of a list of doublets. This makes the definition of density and observation much simpler. Please see examples below as well as samples in the `test` directory.

Examples

```
# ---------------------------------------
# Normal (gaussian) HMM with two states
# state 1 : mean=-1, sd=0.1
# state 2 : mean=+1, sd=0.1
# ergodic HMM with transition probability = 0.1
#
# full specification:
trans <- matrix(c(0.9, 0.1, 0.1, 0.9), nrow=2)
init <- c(0.5, 0.5)
hmm <- thmm.init(dnorm, trans, init, mean=c(-1,1), sd=c(0.1, 0.1))
#
# *same as previous* with simplified call:
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=0.1)

# simulate some sample
obs <- thmm.simulate(hmm, 100, .seed=0)

# run viterbi to retrieve states
vit <- thmm.viterbi(hmm, obs$values)
```

```
## Not run:
# compare actual and predicted states
plot(obs$values)
lines(thmm.parameters(hmm, "mean")[obs$states], col=3, lwd=5)
lines(thmm.parameters(hmm, "mean")[vit$states], col=2)
## End(Not run)

# --------------------------------------
# poisson distribution
#
hmm <- thmm.init(dpois, 0.1, lambda=c(5, 10))
obs <- thmm.simulate(hmm, 100, .seed=0)
vit <- thmm.viterbi(hmm, obs$values)
## Not run:
plot(obs$values)
lines(thmm.parameters(hmm, "lambda")[obs$states], col=3, lwd=5)
lines(thmm.parameters(hmm, "lambda")[vit$states], col=2)
## End(Not run)

# --------------------------------------
# mixture of gaussians
#
# we need to define the pdf
#
# note that this function is 'vectorized' on parameters (and x).
# the computation with log=TRUE will avoid underflow
#
dmixnorm <- function(x, mean=0, sd=1, prob=1, log=FALSE) {
  msp <- mapply(function(m, s, p) list(mean=m, sd=s, prob=p), mean, sd, prob, SIMPLIFY=FALSE)
  res <- if (log) {
    lapply(x, function(x) sapply(msp, function(p) {
      pp <- base::log(p$prob) + dnorm(x, mean=p$mean, sd=p$sd, log=TRUE)
      mp <- max(pp)
      mp + base::log(sum(exp(pp-mp)))
    }))
  } else {
    lapply(x, function(x) sapply(msp, function(p)
      sum(p$prob*dnorm(x, mean=p$mean, sd=p$sd, log=FALSE))))
  }
  unlist(res)
}
#
# and random generator
# again 'vectorized' on parameters (and x).
rmixnorm <- function(n, mean=0, sd=1, prob=1) {
  .cycle <- function(x, n) head(rep(x, n), n)
  msp <- mapply(function(m, s, p) list(mean=m, sd=s, prob=p), mean, sd, prob, SIMPLIFY=FALSE)
  unlist(lapply(seq_len(n), function(i) sapply(msp, function(p) {
         r <- mapply(function(m, s) rnorm(1, m, s), p$mean, p$sd)
         sample(r, size=1, prob=.cycle(p$prob, length(r)))
  })))
}

# hmm with 2 states
# state1: mixture of 2 gaussians with mean=-1,1 sd=0.1,0.1 and prob=0.1,0.9
# state2: single gaussian mean=0 sd=0.2
```

```
hmm <- thmm.init(dmixnorm, 0.2, mean=list(c(-1,1), 0), sd=list(0.1, 0.2),
                                prob=list(c(0.5,0.5), 1))
obs <- thmm.simulate(hmm, n=100, .seed=0)
vit <- thmm.viterbi(hmm, obs$values)
## Not run:
plot(obs$values)
lines((2-obs$states), col=3)
y <- thmm.parameters(hmm, "mean")[vit$states]
lines(sapply(y, function(x) x[2%%length(x)+1]), col=2)
## End(Not run)

# ---------------------------------------
# discrete distribution : the dishonest casino
#
# the pdf (with p6 parameter = probability of drawing a six)
dice <- function(x, p6=1/6, log=FALSE) {
  r <- unlist(lapply(x, function(x) {
    unlist(lapply(p6, function(p6) {
      if (is.na(x)) NA
      else if (x == 6) p6
      else if (x %in% 1:5) (1-p6)/5
      else 0
    })) }), recursive=FALSE)
  if (log) r <- base::log(r)
  r
}
# and the random generator (d->r)ice
rice <- function(n, p6=1/6) {
  sample(1:6, n, prob=c(rep((1-p6)/5, 5), p6), replace=TRUE)
}

hmm <- thmm.init(dice, 0.1, p6=c(1/6, 3/6))  # the second dice is loaded
obs <- thmm.simulate(hmm, 100, .seed=0)
vit <- thmm.viterbi(hmm, obs$values)
## Not run:
plot(obs$values)
lines((obs$states-1)*5+1, col=3) # truth
lines((vit$states-1)*5+1, col=2) #predicted
## End(Not run)

# ---------------------------------------
# univariate normal : Baum-Welch
#
# reference hmm
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=0.1)
obs <- thmm.simulate(hmm, n=1000, .seed=0)

# this one converge to the correct solution
hmm0 <- thmm.init(dnorm, 0.1, mean=c(0,5))
bw   <- thmm.baumwelch(hmm0, obs$values)

# but not that one
hmm0 <- thmm.init(dnorm, 0.1, mean=c(0,10))
bw   <- thmm.baumwelch(hmm0, obs$values)

# so adding constraints: no update of trans nor init
hmm0 <- thmm.init(dnorm, 0.1, mean=c(0,10))
```

```
ctrl <- thmm.bw.ctrl(hmm0, do.trans=FALSE, do.init=FALSE)
bw   <- thmm.baumwelch(hmm0, obs$values, ctrl)


# ----------------------------------------
# bivariate normal
# implemented using complex numbers
# see test.multivar.norm.r for a more general multivariate version
#

# helper
.c <- function(x,y=x) complex(real=x, imaginary=y)

#
# pdf bivariate normal (with 0 covariance)
#
dxynorm <- function(x, mean=.c(0), sd=.c(1), log=FALSE) {
  re <- dnorm(Re(x), mean=Re(mean), sd=Re(sd), log=log)
  im <- dnorm(Im(x), mean=Im(mean), sd=Im(sd), log=log)
  if (log) re+im else re*im
}

#
# random generation function
#
rxynorm <- function(n, mean=.c(0), sd=.c(1)) {
  re <- rnorm(n, mean=Re(mean), sd=Re(sd))
  im <- rnorm(n, mean=Im(mean), sd=Im(sd))
  .c(re, im)
}

hmm <- thmm.init(dxynorm, 0.1,
                 mean=c(.c(10, 20), .c(50, 100)),
                 sd=c(.c(5,10),.c(25, 40)))

obs <- thmm.simulate(hmm, 100, with.values=TRUE)
vit <- thmm.viterbi(hmm, obs$values)
lvl <- thmm.parameters(hmm, "mean")

## Not run:
par(mfrow=c(2,1))
plot(Re(obs$values))
lines(Re(lvl[obs$states]), col=3, lwd=2)
lines(Re(lvl[vit$states]), col=2, lty=2, lwd=2)
plot(Im(obs$values))
lines(Im(lvl[obs$states]), col=3, lwd=2)
lines(Im(lvl[vit$states]), col=2, lty=2, lwd=2)
par(mfrow=c(1,1))
## End(Not run)

# ----------------------------------------
# bivariate normal + poisson
#

# helper : take ith element of tuple
.i <- function(x, i) sapply(x, function(x) x[i])
```

```
# pdf : bivariate normal + poisson
#

dnorpois <- function(x, mean=0, sd=1, lambda=10, log=FALSE) {
  .prod <- if (log) sum else prod
  unlist(mapply(function(m, s, l) {
    xn <- dnorm(.i(x,1), m, s, log=log)
    xp <- dpois(.i(x,2), l, log=log)
    mapply(.prod, xn, xp)
  },
  mean, sd, lambda, SIMPLIFY=FALSE), use.names=FALSE)
}

# random number generator
#

rnorpois <- function(n, mean=0, sd=1, lambda=10) {
  mapply(c, rnorm(n, mean, sd), rpois(n, lambda), SIMPLIFY=FALSE)
}

hmm <- thmm.init(dnorpois, 0.1,
                 mean=c(10, 20, 30), sd=c(1, 2, 3),
                 lambda=c(5, 10, 15))

obs <- thmm.simulate(hmm, 100, with.values=TRUE)

vit <- thmm.viterbi(hmm, obs$values)

lvl.mean <- thmm.parameters(hmm, "mean")
lvl.lamb <- thmm.parameters(hmm, "lambda")

## Not run:
par(mfrow=c(2,1))
plot(.i(obs$values,1), main="normal")
lines(lvl.mean[obs$states], col=3, lwd=2)
lines(lvl.mean[vit$states], col=2, lty=2, lwd=2)
plot(.i(obs$values,2), main="poisson")
lines(lvl.lamb[obs$states], col=3, lwd=2)
lines(lvl.lamb[vit$states], col=2, lty=2, lwd=2)
par(mfrow=c(1,1))
## End(Not run)
```

thmm.aic *compute AIC*

Description

compute Akaike's Information Criterion (AIC) of hmm and observation.

Usage

```
thmm.aic(hmm, obs, np, nt = nrow(hmm$trans) * (nrow(hmm$trans) - 1),
  ni = nrow(hmm$trans) - 1, k = 2, llk = NA, .useC = TRUE)
```

Arguments

`hmm`	DTHmm model (from thmm.init)
`obs`	numerical vector of observations (may be NULL if `llk` is provided
`np`	number of free parameters of hmm density function (e.g. 2 for normal, 1 for poisson).
`nt`	number of free parameters for transition matrix.
`ni`	number of free parameters for initial probabilities.
`k`	the penalty parameter. the default k=2 is the classical AIC.
`llk`	numeric, log-likelihood(obs | hmm). if NA then llk will be computed using thmm.forward.
`.useC`	logical if TRUE (default) use C code else use R code for thmm.forward

Details

AIC = `-2 * llk + (nt + ni + M * np) * k`
with `M=number of states`

Value

numeric AIC

Examples

```
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=0.1)
obs <- thmm.simulate(hmm, 100, .seed=0)$values
thmm.aic(hmm, obs, np=2, nt=1, ni=0)
```

`thmm.backward` *(log) backward probabilities*

Description

compute backward probabilities of observations according to a discrete time hidden markov model (DTHmm).

Usage

```
thmm.backward(hmm, obs, .useC = TRUE)
```

Arguments

`hmm`	DTHmm model (from thmm.init)
`obs`	numerical vector of observations
`.useC`	logical if TRUE (default) use C code else use R code

Value

list with one components: `logbeta` (for symmetry with thmm.forward)

- `logbeta` : the (log of) backward probabilities (densities) matrix:
  `beta_{i,j} = Pr{ X_{i+1} = obs_{i+1}, ..., X_n = obs_n | state_i = j , hmm}`
  that is the probability (density) of the ending partial sequence `(obs_i+1, ..., obs_n)` given that we started at state j at time i, for this hmm.

Note

the code takes care of rescaling values during calculation to avoid underflow problems and is quite robust in practice.

See Also

thmm.forward, thmm.forward.backward

Examples

```
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=0.1)
obs <- c(rnorm(100, -1, 0.1), rnorm(100, 1, 0.1))
bwd <- thmm.backward(hmm, obs)
#
# test numerical stability
#
dif1 <- bwd$logbeta[-c(200,99),1] - bwd$logbeta[-c(200,99),2]
obs[100] <- 1000000  # this may creates overflow
bwd <- thmm.backward(hmm, obs)
dif2 <- bwd$logbeta[-c(200,99),1] - bwd$logbeta[-c(200,99),2]
max(abs((dif1-dif2)/dif1))
## Not run:
plot(dif1)
points(dif2, col=2, cex=0.1)
## End(Not run)
```

`thmm.baumwelch` *Baum-Welch algorithm*

Description

HMM parameters estimation using the Baum-Welch algorithm

Usage

```
thmm.baumwelch(hmm0, obs, ctrl = thmm.bw.ctrl(hmm0), .mstep, .useC = TRUE)
```

Arguments

`hmm0`	a starting DTHmm (see thmm.init)
`obs`	numerical vector of observations
`ctrl`	a list of control settings obtained by thmm.bw.ctrl
`.mstep`	function name of the mstep function (see details)
`.useC`	logical if TRUE (default) use C code else use R code

Details

the Baum-Welch EM algorithm iteratively alternates two steps: the **E-step** and the **M-step**. The **E-step** (expectation) is distribution independent and is performed by the thmm.estep function. The **M-step** (maximisation) is distribution dependent and you should provide a distribution specific function to perform it. The name of this function is `mstep.<density>` (e.g. `mstep.dnorm` for a gaussian distribution) unless you provide another name as the `.mstep` parameter. Default function is provided for `dnorm` (may be extended in the future). Have a look at mstep.dnorm for an example of function format.
you may control which hmm parameters are actually optimized with `ctrl`. see thmm.bw.ctrl.

Value

a named list of three elements:

- hmm : optimized hmm
- info : information about the iterative process
- loglike : final log-likelihood

Examples

```
# reference hmm
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=0.1)
obs <- thmm.simulate(hmm, n=1000, .seed=0)

# this one converge to the correct solution
hmm0 <- thmm.init(dnorm, 0.1, mean=c(0,5))
bw   <- thmm.baumwelch(hmm0, obs$values)

# but not that one
hmm0 <- thmm.init(dnorm, 0.1, mean=c(0,10))
bw   <- thmm.baumwelch(hmm0, obs$values)

# so adding constraints: no update of trans nor init
hmm0 <- thmm.init(dnorm, 0.1, mean=c(0,10))
ctrl <- thmm.bw.ctrl(hmm0, do.trans=FALSE, do.init=FALSE, verbose=TRUE)
bw   <- thmm.baumwelch(hmm0, obs$values, ctrl)

# that one do not converge either to proper solution
hmm0 <- thmm.init(dnorm, 0.1, mean=c(0,0), sd=0.5)
bw   <- thmm.baumwelch(hmm0, obs$values)

# adding constraints: mean bounds
hmm0 <- thmm.init(dnorm, 0.1, mean=c(0,0), sd=0.5)
ctrl <- thmm.bw.ctrl(hmm0, lower.mean=c(-2,0), upper.mean=c(0,2), verbose=TRUE)
bw   <- thmm.baumwelch(hmm0, obs$values, ctrl)

# adding constraint: as as previous but more general form
hmm0 <- thmm.init(dnorm, 0.1, mean=c(0,0), sd=0.5)
myconst <- function(params) {
  params$mean <- pmax(pmin(params$mean, c(0,2)), c(-2,0))
  params
}
ctrl <- thmm.bw.ctrl(hmm0, constraint=myconst, verbose=TRUE)
bw   <- thmm.baumwelch(hmm0, obs$values, ctrl)
```

`thmm.bic` *compute BIC*

Description

compute Bayesian Information Criterion (BIC) of hmm and observation.

Usage

```
thmm.bic(hmm, obs, np, nt = nrow(hmm$trans) * (nrow(hmm$trans) - 1),
  ni = nrow(hmm$trans) - 1, llk = NA, .useC = TRUE)
```

Arguments

`hmm`	DTHmm model (from thmm.init)
`obs`	numerical vector of observations
`np`	number of free parameters of hmm density function (e.g. 2 for normal, 1 for poisson).
`nt`	number of free parameters for transition matrix.
`ni`	number of free parameters for initial probabilities.
`llk`	numeric, log-likelihood(obs | hmm). if NA then llk will be computed using thmm.forward.
`.useC`	logical if TRUE (default) use C code else use R code for thmm.forward

Details

BIC = `-2 * llk + (nt + ni + M * np) * log(N)`
with `M=number of states; N=length(obs)`

Value

numeric BIC

Examples

```
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=0.1)
obs <- thmm.simulate(hmm, 100, .seed=0)$values
thmm.bic(hmm, obs, np=2, nt=1, ni=0)
```

`thmm.bw.ctrl` *control parameters for Baum-Welch algorithm*

Description

make a list of parameters to control thmm.baumwelch execution.

Usage

```
thmm.bw.ctrl(hmm, maxiter = NA, tol = 1e-05, verbose = FALSE,
  converge = expression(difLL <= ctrl$tol), do.trans = TRUE,
  do.init = TRUE, constraint = NULL, ...)
```

Arguments

`hmm`	DTHmm model (from thmm.init)
`maxiter`	integer max number of iteration (NA will set to hmm dependent value)
`tol`	numeric convergence criterion (delta_log-likelihood)
`verbose`	logical be verbose
`converge`	expression of the convergence criterion
`do.trans`	logical do optimize transition matrix
`do.init`	logical do optimize initial probabilities
`constraint`	if not NULL, should contain a function receiving a list of optimized parameters (at each M-step) and should return a list (with the same format) of constrained parameters values (see examples).
`...`	any parameter passed to `<density>.mstep` function (see thmm.baumwelch). in particular: • parameters of the form `do.<parameter>` are interpreted as logical controling whether or not we should optimize the <parameter>. note that to prevent update you should explicitely set `do.<parameter>` to `FALSE`. if `do.<parameter>` is absent (or set to TRUE) then `<parameter>` **will be** updated. • parameters of the form `upper|lower.<parameter>` are interpreted as numerical upper/lower bounds. if absent then no bounds (i.e. [-Inf, +Inf]) are assumed.

Value

named list of control parameters for thmm.baumwelch

Note

`constraint` provides a more general form of control than `do.<parameter>, upper|lower.<parameter>` but may be less efficient since it is called right after the mstep function.

(for developpers) `do.<parameter>, upper|lower.<parameter>` should be implemented for every mstep function. see mstep.dnorm as an exemple. `constraint` is called by thmm.baumwelch directly.

See Also

thmm.baumwelch

`thmm.estep` *E-Step of EM Baum-Welch algorithm*

Description

performs the expectation step of the EM algorithm for a discrete time HMM process. this function is called by the thmm.baumwelch.

Usage

```
thmm.estep(hmm, obs, .useC = TRUE)
```

Arguments

`hmm`	DTHmm model (from thmm.init)
`obs`	numerical vector of observations
`.useC`	logical if TRUE (default) use C code else use R code

Details

u is defined as:
`u{i,j}=Pr(state_i=j | obs, hmm)`
that is the same as `gamma` in thmm.forward.backward

v is defined as:
`v({i,j,k}=Pr(state_i-1=j, state_i=k | obs, hmm)`
also known as `chsi`

Value

list with three components:

- u length(obs)*number_states matrix containing estimates of the conditional expectations (see details)
- v length(obs)*number_states*number_states matrix containing estimates of the conditional expectations (see details)
- loglike log-likelihood

See Also

thmm.baumwelch

`thmm.forward` *(log) forward probabilities*

Description

compute forward probabilities of observations according to a discrete time hidden markov model (DTHmm).

Usage

```
thmm.forward(hmm, obs, .useC = TRUE)
```

Arguments

`hmm`	DTHmm model (from thmm.init)
`obs`	numerical vector of observations
`.useC`	logical if TRUE (default) use C code else use R code

Value

list with two components: `logalpha` and `loglike`

- `logalpha` : the (log of) forward probabilities (densities) matrix:
  `alpha_{i,j} = Pr{ X_1 = obs_1, ..., X_i = obs_i, state_i = j | hmm}` that is the probability (density) of seeing the partial sequence `(obs_1, ..., obs_i)` and ending up in state j at time i for this hmm.
- `loglike` : the log-likelihood of this observation with this hmm:
  `log(Pr{ X_1 = obs_1, ..., X_n = obs_n | hmm})=max_j(logalpha_{n,j}`

Note

the code takes care of rescaling values during calculation to avoid underflow problems and is quite robust in practice.

since the computation uses density probabilities it is perfectly valid to get a positive log-likelihood.

for multivariate HMM, obs should be a list of (numerical) tuples. (see thmm examples).

See Also

thmm.backward, thmm.forward.backward

Examples

```
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=0.1)
obs <- c(rnorm(100, -1, 0.1), rnorm(100, 1, 0.1))
fwd <- thmm.forward(hmm, obs)
#
# test numerical stability
#
dif1 <- fwd$logalpha[-100,1] - fwd$logalpha[-100,2]
obs[100] <- 1000000  # this may creates overflow
fwd <- thmm.forward(hmm, obs)
```

```
dif2 <- fwd$logalpha[-100,1] - fwd$logalpha[-100,2]
max(abs((dif1-dif2)/dif1))
## Not run:
plot(dif1)
points(dif2, col=2, cex=0.1)
## End(Not run)
```

thmm.forward.backward *(log) forward/backward probabilities*

Description

compute forward/backward probabilities of observations according to a discrete time hidden markov model (DTHmm).

Usage

```
thmm.forward.backward(hmm, obs, .useC = TRUE)
```

Arguments

`hmm`	DTHmm model (from thmm.init)
`obs`	numerical vector of observations
`.useC`	logical if TRUE (default) use C code else use R code

Value

list with five components: `logalpha`, `logbeta`, `gamma`, `rho`, `loglike`

- `logalpha` : the (log of) forward probabilities (densities) matrix:
  `alpha_{i,j} = Pr{ X_1 = obs_1, ..., X_i = obs_i, state_i = j | hmm}` that is the probability (density) of seeing the partial sequence `(obs_1, ..., obs_i)` and ending up in state j at time i for this hmm.
- `logbeta` : the (log of) backward probabilities (densities) matrix:
  `beta_{i,j} = Pr{ X_{i+1} = obs_{i+1}, ..., X_n = obs_n | state_i = j , hmm}` that is the probability (density) of the ending partial sequence `(obs_i+1, ..., obs_n)` given that we started at state j at time i, for this hmm.
- `gamma` : the probabilities matrix:
  `gamma_{i,j} = Pr{ state_i = j | obs , hmm}` that is the probability of being at time i in state j given this observation and this hmm.
- `logrho` : the (log of) probabilities (densities) vector:
  `rho_{i} = Pr{ X_1 = obs_1, ..., X_i = obs_i | hmm}` that is the probability (density) of seeing the partial sequence `(obs_1, ..., obs_i)` for this hmm.
- `loglike` : the log-likelihood of this observation with this hmm:
  `log(Pr{ X_1 = obs_1, ..., X_n = obs_n | hmm})` (see thmm.forward)

Note

the code takes care of rescaling values during calculation to avoid underflow problems and is quite robust in practice.

by definition `loglike = logrho(length(obs))` (although it is not computed this way).

See Also

thmm.forward, thmm.backward

Examples

```
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=0.1)
obs <- c(rnorm(100, -1, 0.1), rnorm(100, 1, 0.1))
fb <- thmm.forward.backward(hmm, obs)
#
# test numerical stability
#
obs[101] <- 1000000  # this may creates overflow
fb <- thmm.forward.backward(hmm, obs)
## Not run:
plot(fb$gamma[,2])
## End(Not run)
```

`thmm.init` *initialize HMM*

Description

initialize a discrete time hidden markov model (DTHmm)

Usage

```
thmm.init(density, trans, init = NULL, ...)
```

Arguments

`density`	a probability density function (pdf) such as dnorm (see notes)
`trans`	the transition matrix given either as a matrix, a vector or a constant (see details)
`init`	vector of probabilities of the initial states (see details)
`...`	parameters of `density` (see examples)

Details

the number of states (m) is determined from the ... arguments (see notes) and should be consistent with the value passed to `trans` and `init` if provided.

if `trans` is a constant then the transition matrix is computed as a mxm matrix with **off-diagonal** elements `trans`.

if `trans` is a vector (it should then be of length m) then the transition matrix is computed as a mxm matrix with **diagonal** elements `trans`.

if `init` is NULL then `init` is computed as a vector of uniform probabilities (1/m).

if `init` is not of length m then it is recycled/shortened to proper length m.

Value

a DTHmm object describing the DTHmm

Note

the pdf function `density` has the form:
`density(x, ..., log=TRUE|FALSE)`
see dnorm, dpois, dbeta, etc. as examples but you may write your own as well, with the following specifications:

- `density` parameters must have default values if you don't specify them in `...`
- `density` must accepts vectors (including of NA's) as parameters (possibly recycling them) (each parameter corresponds to each state) and return a vector of length m (possibly of NA's) when called with a scalar `x` (including NA). this is used to determine the actual number of states m.
- (not mandatory) It may accept a vector (including of NA's) as `x` argument (with individual states parameters) and return a vector of size `length(x)`. this is used by the C code only, so if this condition is not satisfied the C code will workaround but will run a bit slower (albeit still quicker than R code).

You may pass the function symbol or function name (character) as `density` argument (but not a direct lambda expression nor anything fancy).

For multivariate HMM, the density function should accept a list of (numeric) tuples as `x` argument (or anything equivalent like a list of complex numbers for bivariate). The parameters can be anything relevant to your mutivariate density distribution but may be received as lists (one element for each state) instead of vectors. See examples in thmm for details.

Examples

```
# ----------------------------------------
# Normal (gaussian) HMM with two states
# full specification
trans <- matrix(c(0.9, 0.1, 0.1, 0.9), nrow=2)
init <- c(0.5, 0.5)
hmm <- thmm.init(dnorm, trans, init, mean=c(-1,1), sd=c(0.1, 0.1))

# ----------------------------------------
# *same as previous* with simplified call
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=0.1)

# ----------------------------------------
# three states HMM with Poisson distribution
hmm <- thmm.init(dpois, 0.1, lambda=1:3)

# ----------------------------------------
# two states HMM with Gamma distribution
hmm <- thmm.init(dgamma, 0.1, shape=c(3, 10), rate=c(1, 2))

# ----------------------------------------
# mixture of gaussians
# state1: mixture of 2 gaussians with mean=-1,1 sd=0.1,0.1 and prob=0.1,0.9
# state2: gaussian mean=0 sd=0.2
#
# we define first \code{dmixnorm} as the pdf of a mixture of gaussians
# note that this function is 'vectorized' on parameters (and x either).
# the computation with log=TRUE will avoid underflow
#
```

```
dmixnorm <- function(x, mean=0, sd=1, prob=1, log=FALSE) {
 msp <- mapply(function(m, s, p) list(mean=m, sd=s, prob=p), mean, sd, prob, SIMPLIFY=FALSE)
  res <- if (log) {
    lapply(x, function(x) sapply(msp, function(p) {
      pp <- base::log(p$prob) + dnorm(x, mean=p$mean, sd=p$sd, log=TRUE)
      mp <- max(pp)
      mp + base::log(sum(exp(pp-mp)))
    }))
  } else {
    lapply(x, function(x) sapply(msp, function(p)
      sum(p$prob*dnorm(x, mean=p$mean, sd=p$sd, log=FALSE))))
  }
  unlist(res)
}
#
# then the hmm
hmm <- thmm.init(dmixnorm, 0.1, mean=list(c(-1,1), 0), sd=list(0.1, 0.2),
                                prob=list(c(0.5,0.5), 1))

# ----------------------------------------
# discrete distribution : binomial
#
# dbinom does not have default parameters
# so we need to specify all of them
hmm <- thmm.init(dbinom, 0.1, size=c(50, 100), prob=c(0.5, 0.1))

# ----------------------------------------
# discrete distribution : the dishonest casino
#
# we first define the pdf
#
dice <- function(x, p6=1/6, log=FALSE) {
  r <- unlist(lapply(x, function(x) {
    unlist(lapply(p6, function(p6) {
      if (is.na(x)) NA
      else if (x == 6) p6
      else if (x %in% 1:5) (1-p6)/5
      else 0
    })) }), recursive=FALSE)
  if (log) r <- base::log(r)
  r
}
# and the random generator (d->r)ice (see \link{thmm.simulate})
rice <- function(n, p6=1/6) {
  sample(1:6, n, prob=c(rep((1-p6)/5, 5), p6), replace=TRUE)
}

hmm <- thmm.init(dice, 0.3, p6=c(1/6, 3/6))
obs <- thmm.simulate(hmm, 100)
## Not run:
plot(obs$values)

## End(Not run)
```

`thmm.parameters` *get hmm parameters*

Description

get hmm global and individual states parameters

Usage

```
thmm.parameters(hmm, what = NULL)
```

Arguments

`hmm`	DTHmm model (from thmm.init)
`what`	character parameter name to retrieve (all parameters if NULL (default))

Value

if `what==NULL` return a list of 5 components: `nstates`, `density`, `args`, `init`, `trans`

- nstates : integer number of states
- density : character string name of probability density function
- args : list (of length nstates) of named lists, each element gives the density function parameters for state i. (see note).
- init : numerical vector of initial probabilities
- trans : numerical transition matrix

if `what!=NULL` then what should be either one of the above names or the name of one of the density function parameters. the function then return only this information.

Examples

```
# univariate gaussian hmm
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=1)
ns <- thmm.parameters(hmm, "nstates")
means <- thmm.parameters(hmm, "mean")
# viterbi decode
obs <- rnorm(100)
vit <- thmm.viterbi(hmm, obs)
# as values
means[vit$states]
```

`thmm.simulate` *simulate hidden markov process*

Description

generate states and observations vectors following a discrete time HMM

Usage

```
thmm.simulate(hmm, npts, with.values = TRUE, log = FALSE, .random = NULL,
  .seed = NA)
```

Arguments

`hmm`	DTHmm model (from thmm.init)
`npts`	number of points to simulate
`with.values`	logical also compute observed values (see details)
`log`	(if `with.values=TRUE`) compute log of values
`.random`	name of random generation function (see details).
`.seed`	if not NA set random seed before generation

Details

if `with.values=TRUE` then the random generation function used to produce values from states is `'(d->r)ensity'` i.e. the name of density probability function with first letter (usually a 'd') replaced by a 'r'. (e.g. if `density=dnorm` then the random function is `rnorm`) this can be overriden by using the `.random` parameter.
This function has the following form:
`random(n, ...)` where `...` are the same parameters as `density`.
In contrast to `density`, the generation function does not need to be vectorized on parameters nor `n` (actually it is always called with `n=1` and individual state parameters).

Value

if `with.values=FALSE` integer vector of simulated states. if `with.values=TRUE` a list of two vectors `states` and `values`.

Note

neither `states` not `values` are deterministic, you should call set.seed to produce reproducible vectors.

for multivariate HMM, the result is a list of (numeric) tuples (one tuple element per each random variable). See examples in thmm for details.

Examples

```
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=0.1)
set.seed(0)
thmm.simulate(hmm, 10, log=TRUE)
```

`thmm.viterbi` *Viterbi algorithm*

Description

This function calculates the optimal hidden states sequence using Viterbi's algorithm.

Usage

```
thmm.viterbi(hmm, obs, with.probseq = FALSE, .useC = TRUE)
```

Arguments

`hmm`	DTHmm model (from thmm.init)
`obs`	numerical vector of observations
`with.probseq`	should we compute `logprobseq and loglike` (see value)
`.useC`	logical if TRUE (default) use C code else use R code

Details

the purpose of the Viterbi's algorithm is to determine the sequence of states `(k_1*, ..., k_n*)` which maximises the joint distribution of the hidden states given the entire observed process. i.e:
`(k_1*, ..., k_n*) = argmax(Pr(state_1=k_1, ..., state_n=k_n, X_1=obs_1, ..., X_n=obs_n | hmm))`

Value

list with five components: `states`, `lognu`, `logviterbi` `loglike` and `logprobseq`

- `states` : vector of optimal states `(k_1*, ..., k_n*)`
- `lognu` : the (log of) joint probabilities (densities) matrix:
  `nu_{i+1,j} = Pr(state_1=k_1*, ..., state_i=k_i*, state_i+1 = j,  X_1=obs_1, ..., X_i=obs_i | h`
  where k_i* is the optimal state at time i.
- `logviterbi` : log of Viterbi's score, i.e. the log of joint probability (density) of the optimal sequence of states and observations with this hmm
  `log(Pr(state_1=k_1*, ..., state_n=k_n*, obs | hmm))`
  `=max_j(lognu{n,j})`
- `loglike` : the log-likelihood of this observation with this hmm:
  `log(Pr{ X_1 = obs_1, ..., X_n = obs_n | hmm})`=log(Pr(obs | hmm)) (see thmm.forward)
  this value is NA if `with.probseq == FALSE`
- `logprobseq` : log of probability (density) of the optimal sequence of states conditionally to the observations. i.e
  `log(Pr(state_1=k_1*, ..., state_n=k_n* | obs, hmm))`
  `=log(Pr(state_1=k_1*, ..., state_n=k_n*, obs | hmm) / Pr(obs | hmm))`
  `=logviterbi - loglike`

this value is NA if `with.probseq == FALSE`

Note

the code takes care of rescaling values during calculation to avoid underflow problems and is quite robust in practice.

for multivariate HMM, obs should be a list of (numerical) tuples. (see thmm examples).

See Also

thmm.forward

Examples

```
hmm <- thmm.init(dnorm, 0.1, mean=c(-1,1), sd=0.1)
obs <- c(rnorm(100, -1, 0.1), rnorm(100, 1, 0.1))
vit <- thmm.viterbi(hmm, obs, with.probseq=TRUE)
table(vit$states)
#
```

```
# test numerical stability
#
obs[101] <- 1000000  # this may creates overflow
vit <- thmm.viterbi(hmm, obs, with.probseq=TRUE)
table(vit$states)
## Not run:
plot(vit$states)
## End(Not run)
```

Index