Package ‘polyapost’

May 1, 2020

Version 1.6

Date 2020-04-30

Imports boot, stats

Depends R (>= 3.2.0), rcdd (>= 1.2)

Title Simulating from the Polya Posterior

Author Glen Meeden <glen@stat.umn.edu> and Radu Lazar

<lazar@stat.umn.edu> and Charles J. Geyer <charlie@stat.umn.edu>

Maintainer Glen Meeden <glen@stat.umn.edu>

ByteCompile TRUE

Description Simulate via Markov chain Monte Carlo (hit-and-run algorithm) a Dirichlet distribution conditioned to satisfy a finite set of linear equality and inequality constraints (hence to lie in a convex polytope that is a subset of the unit simplex).

License GPL (>= 2)

NeedsCompilation yes

Repository CRAN

Date/Publication 2020-05-01 06:40:03 UTC

R topics documented:

constrppmn *Estimating a Population Mean Using the Constrained Polya Posterior.*

Description

Let $p = (p_1, \ldots, p_n)$ be a probability distribution defined on y_{samp}, the set of observed values, in a sample of size n from some population. p is assumed to belong to a polytope which is a lower dimensional subset of the n-dimensional simplex. The polytope is defined by a collection of linear equality and inequality constraints. A dependent sequence of values for p are generated by a Markov chain using the Metropolis-Hastings algorithm whose stationary distribution is the uniform distribution over the polytope. For each generated value of p the corresponding mean, $\sum_i p_i y_i$ is found.

Usage

```
constrppmn(A1,A2,A3,b1,b2,b3,initsol,reps,ysamp,burnin)
```

Arguments

`A1`	The matrix for the equality constraints.This must always contain the constraint `sum(p) == 1`.
`A2`	The matrix for the <= inequality constraints. This must always contain the constraints `-p <= 0`.
`A3`	The matrix for the >= inequality constraints. If there are no such constraints `A3` must be set equal to NULL.
`b1`	The rhs vector for `A1`, each component must be nonnegative.
`b2`	The rhs vector for `A2`, each component must be nonnegative.
`b3`	The rhs vector for `A3`, each component must be nonnegative. If `A3` is `NULL` then `b3` must be NULL.
`initsol`	A vector which lies in the interior of the polytope.
`reps`	The total length of the chain that is generated.
`ysamp`	The observed sample from the population of interest.
`burnin`	The point in the chain at which the set of computed means begins.

Value

The returned value is a list whose first component is the chain of the means of length `reps -burnin -1`, whose second component is the mean of the first component (i.e. the Polya estimate of the population mean) and whose third component is the 2.5th and 97.5th quantiles of the first component (i.e. an approximate 95 percent confidence interval of the population mean).

Examples

```
A1<-rbind(rep(1,6),1:6)
A2<-rbind(c(2,5,7,1,10,8),diag(-1,6))
b1<-c(1,3.5)
b2<-c(6,rep(0,6))
initsol<-rep(1/6,6)
rep<-1006
burnin<-1000
ysamp<-c(1,2.5,3.5,7,4.5,6)
out<-constrppmn(A1,A2,NULL,b1,b2,NULL,initsol,rep,ysamp,burnin)
out[[1]] # the Markov chain of the means.
out[[2]] # the average of out[[1]]
out[[3]] # the 2.5th and 97.5th quantiles of out[[1]]
```

---

constrppprob — *Dependent Sampling from the Uniform Distribution on a Polytope.*

Description

Let $p = (p_1, \ldots, p_n)$ be a probability distribution which belongs to a lower dimensional polytope of the n-dimensional simplex. The polytope is defined by a collection of linear equality and inequality constraints. A dependent sequence of the p's are generated by a Markov chain using the Metropolis-Hastings algorithm whose stationary distribution is the uniform distribution over the polytope. This is done by generating k blocks of size `step` where the last member of each is returned.

Usage

```
constrppprob(A1,A2,A3,b1,b2,b3,initsol,step,k)
```

Arguments

`A1`	The matrix for the equality constraints.This must always contain the constraint `sum(p) == 1`.
`A2`	The matrix for the <= inequality constraints. This must always contain the constraints `-p <= 0`.
`A3`	The matrix for the >= inequality constraints. If there are no such constraints `A3` must be set equal to `NULL`.
`b1`	The rhs vector for `A1`, each component must be nonnegative.
`b2`	The rhs vector for `A2`, each component must be nonnegative.
`b3`	The rhs vector for `A3`, each component must be nonnegative. If `A3` is `NULL` then `b3` must be NULL.
`initsol`	A vector which lies in the interior of the polytope.
`step`	The number of p's generated in a block before the last member of a block is returned.
`k`	The total number of blocks generated and hence the number of p's returned.

Value

The returned value is a k by n matrix of probability vectors.

Examples

```
A1<-rbind(rep(1,6),1:6)
A2<-rbind(c(2,5,7,1,10,8),diag(-1,6))
A3<-matrix(c(1,1,1,0,0,0),1,6)
b1<-c(1,3.5)
b2<-c(6,rep(0,6))
b3<-0.45
initsol<-rep(1/6,6)
constrppprob(A1,A2,A3,b1,b2,b3,initsol,2000,5)
```

---

feasible — *Feasible Solution for a Probability Distribution which must Satisfy a System of Linear Equality and Inequality Constraints.*

Description

This function finds a feasible solution, $p = (p_1, \ldots, p_n)$, in the n-dimensional simplex of probability distributions which must satisfy $A_1 p = b_1$, $A_2 p = b_2$, and $A_3 p = b_3$, All the components of the b_i must be nonnegative In addition each probability in the solution must be at least as big as `eps`, a small positive number.

Usage

```
feasible(A1,A2,A3,b1,b2,b3,eps)
```

Arguments

`A1`	The matrix for the equality constraints.This must always contain the constraint `sum(p) == 1`.
`A2`	The matrix for the <= inequality constraints. This must always contain the constraints `-p <= 0`.
`A3`	The matrix for the >= inequality constraints. If there are no such constraints `A3` must be set equal to `NULL`.
`b1`	The rhs vector for `A1`, each component must be nonnegative.
`b2`	The rhs vector for `A2`, each component must be nonnegative.
`b3`	The rhs vector for `A3`, each component must be nonnegative. If `A3` is `NULL` then `b3` must be `NULL`.
`eps`	A small positive number. Each member of the solution must be at least as large as `eps`. Care must be taken not to choose a value of `eps` which is too large.

Value

The function returns a vector. If the components of the vector are positive then the feasible solution is the vector returned, otherwise there is no feasible solution.

Examples

```
A1<-rbind(rep(1,7),1:7)
b1<-c(1,4)
A2<-rbind(c(1,1,1,1,0,0,0),c(.2,.4,.6,.8,1,1.2,1.4))
b2<-c(1,2)
A3<-rbind(c(1,3,5,7,9,10,11),c(1,1,1,0,0,0,1))
b3<-c(5,.5)
eps<-1/100
feasible(A1,A2,A3,b1,b2,b3,eps)
```

---

hitrun — *Hit and Run Algorithm for Constrained Dirichlet Distribution*

Description

Markov chain Monte Carlo for equality and inequality constrained Dirichlet distribution using a hit and run algorithm.

Usage

```
hitrun(alpha, ...)

## Default S3 method:
hitrun(alpha, a1 = NULL, b1 = NULL, a2 = NULL, b2 = NULL,
     nbatch = 1, blen = 1, nspac = 1, outmat = NULL, debug = FALSE,
     stop.if.implied.equalities = FALSE, ...)

## S3 method for class 'hitrun'
hitrun(alpha, nbatch, blen, nspac, outmat, debug, ...)
```

Arguments

`alpha`	parameter vector for Dirichlet distribution. Alternatively, an object of class `"hitrun"` that is the result of a previous invocation of this function, in which case this run continues where the other left off.
`nbatch`	the number of batches.
`blen`	the length of batches.
`nspac`	the spacing of iterations that contribute to batches.
`a1`	a numeric or character matrix or `NULL`. See details.
`b1`	a numeric or character vector or `NULL`. See details.
`a2`	a numeric or character matrix or `NULL`. See details.

`b2`	a numeric or character vector or `NULL`. See details.
`outmat`	a numeric matrix, which controls the output. If `p` is the constrained Dirichlet random vector being simulated, then `outmat %*% p` is the functional of the state that is averaged. May be `NULL`, in which case the identity matrix is used.
`debug`	if `TRUE`, then additional output useful for debugging is produced.
`stop.if.implied.equalities`	If `TRUE` stop if there are any implied equalities.
`...`	ignored arguments. Allows the two methods to have different arguments. You cannot change the Dirichlet parameter or the constraints (hence cannot change the target distribution) when using the method for class `"hitrun"`.

Details

Runs a hit and run algorithm (for which see the references) producing a Markov chain with equilibrium distribution having a Dirichlet distribution with parameter vector `alpha` constrained to lie in the subset of the unit simplex consisting of `x` satisfying

```
a1 %*% x <= b1
a2 %*% x == b2
```

Hence if `a1` is `NULL` then so must be `b1`, and vice versa, and similarly for `a2` and `b2`.

If any of `a1`, `b1`, `a2`, `b2` are of type `"character"`, then they must be valid GMP (GNU multiple precision) rational, that is, if run through q2q, they do not give an error. This allows constraints to be represented exactly (using infinite precision rational arithmetic) if so desired. See also the section on this subject below.

Value

an object of class `"hitrun"`, which is a list containing at least the following components:

`batch`	`nbatch` by `p` matrix, the batch means, where `p` is the row dimension of `outmat`.
`initial`	initial state of Markov chain.
`final`	final state of Markov chain.
`initial.seed`	value of `.Random.seed` before the run.
`final.seed`	value of `.Random.seed` after the run.
`time`	running time from `system.time()`.
`alpha`	the Dirichlet parameter vector.
`nbatch`	the argument `nbatch` or `obj$nbatch`.
`blen`	the argument `blen` or `obj$blen`.
`nspac`	the argument `nspac` or `obj$nspac`.
`outmat`	the argument `outmat` or `obj$outmat`.

GMP Rational Arithmetic

The arguments `a1`, `b1`, `a2`, and `b2` can and should be given as GMP (GNU multiple precision) rational values. This allows the computational geometry calculations for the constraint set to be done exactly, without error. For example, if `a1` has elements that have been rounded to two decimal places one should do

```
a1 <- z2q(round(100 * a1), rep(100, length(a1)))
```

and similarly for `b1`, `a2`, and `b2` to make them exact. For all the conversion functions between ordinary computer numbers and GMP rational numbers see ConvertGMP. For all the functions that do arithmetic on GMP rational numbers, see ArithmeticGMP.

Warning About Implied Equality Constraints

If any constraints supplied as inequality constraints (specified by rows of `a1` and the corresponding components of `b1`) actually hold with equality for all points in the constraint set, this is called an implied equality constraint. The program must establish that none of these exist (which is a fast operation) or, otherwise, find out which constraints supplied as inequality constraints are actually implied equality constraints, and this operation is very slow when the state is high dimensional. One example with 1000 variables took 3 days of computing time when there were implied equality constraints in the specification. The same example takes 9 minutes when the same constraint set is specified in a different way so that there are no implied equality constraints.

This issue is not a big deal if there are only in the low hundreds of variables, because the algorithm to find implied equality constraints is not that slow. The same example that takes 3 days of computing time with 1000 variables takes only 15 seconds with 100 variables, 3 and 1/2 minutes with 200 variables, and 23 minutes with 300 variables. As one can see, this issue does become a big deal as the number of variables increases. Thus users should avoid implied inequality constraints, if possible, when there are many variables. Admittedly, there is no sure way users can identify and eliminate implied equality constraints. (The sure way to do that is precisely the time consuming step we are trying to avoid.) The argument `stop.if.implied.equalities` can be used to quickly test for the presence of implied equalities.

Philosophy of MCMC

This function follows the philosophy of MCMC used in the CRAN package `mcmc` and the introductory chapter of the *Handbook of Markov Chain Monte Carlo* (Geyer, 2011).

The `hitrun` function automatically does batch means in order to reduce the size of output and to enable easy calculation of Monte Carlo standard errors (MCSE), which measure error due to the Monte Carlo sampling (not error due to statistical sampling — MCSE gets smaller when you run the computer longer, but statistical sampling variability only gets smaller when you get a larger data set). All of this is explained in the package vignette for the `mcmc` package (`vignette("demo","mcmc")`) and in Section 1.10 of Geyer (2011).

The `hitrun` function does not apparently do “burn-in” because this concept does not actually help with MCMC (Geyer, 2011, Section 1.11.4) but the re-entrant property of the hitrun function does allow one to do “burn-in” if one wants. Assuming `alpha`, `a1`, `b1`, `a2`, and `b2` have been already defined

```
out <- hitrun(alpha, a1, b1, a2, b2, nbatch = 1, blen = 1e5)
out <- hitrun(out, nbatch = 100, blen = 1000)
```

throws away a run of 100 thousand iterations before doing another run of 100 thousand iterations that is actually useful for analysis, for example,

```
apply(out$batch, 2, mean)
apply(out$batch, 2, sd)
```

gives estimates of posterior means and their MCSE assuming the batch length (here 1000) was long enough to contain almost all of the signifcant autocorrelation (see Geyer, 2011, Section 1.10, for more on MCSE). The re-entrant property of the `hitrun` function (the second run starts where the first one stops) assures that this is really "burn-in".

The re-entrant property allows one to do very long runs without having to do them in one invocation of the `hitrun` function.

```
out2 <- hitrun(out)
out3 <- hitrun(out2)
batch <- rbind(out$batch, out2$batch, out3$batch)
```

produces a result as if the first run had been three times as long.

References

Belisle, C. J. P., Romeijn, H. E. and Smith, R. L. (1993) Hit-and-run algorithms for generating multivariate distributions. *Mathematics of Operations Research*, **18**, 255–266. doi: 10.1287/moor.18.2.255.

Chen, M. H. and Schmeiser, B. (1993) Performance of the Gibbs, hit-and-run, and Metropolis samplers. *Journal of Computational and Graphical Statistics*, **2**, 251–272.

Geyer, C. J. (2011) Introduction to MCMC. In *Handbook of Markov Chain Monte Carlo*. Edited by S. P. Brooks, A. E. Gelman, G. L. Jones, and X. L. Meng. Chapman & Hall/CRC, Boca Raton, FL, pp. 3–48.

See Also

`ConvertGMP` and `ArithmeticGMP`

Examples

```
# Bayesian inference for discrete probability distribution on {1, ..., d}
# state is probability vector p of length d
d <- 10
x <- 1:d
# equality constraints
#     mean equal to (d + 1) / 2, that is, sum(x * p) = (d + 1) / 2
# inequality constraints
#     median less than or equal to (d + 1) / 2, that is,
#         sum(p[x <= (d + 1) / 2]) <= 1 / 2
a2 <- rbind(x)
```

```
b2 <- (d + 1) / 2
a1 <- rbind(as.numeric(x <= (d + 1) / 2))
b1 <- 1 / 2
# simulate prior, which Dirichlet(alpha)
# posterior would be another Dirichlet with n + alpha - 1,
#    where n is count of IID data for each value
alpha <- rep(2.3, d)
out <- hitrun(alpha, nbatch = 30, blen = 250,
    a1 = a1, b1 = b1, a2 = a2, b2 = b2)
# prior means
round(colMeans(out$batch), 3)
# Monte Carlo standard errors
round(apply(out$batch, 2, sd) / sqrt(out$nbatch), 3)
```

---

polyap *Polya Sampling from an Urn*

Description

Consider an urn containing a finite set of values. An item is selected at random from the urn. Then it is returned to the urn along with another item with the same value. Next a value is selected at random from the reconstituted urn and it and a copy our returned to the urn. This process is repeated until k additional items have been added to the original urn. The original composition of the urn along with the selected values, in order, are returned.

Usage

```
polyap(ysamp, k)
```

Arguments

ysamp	A vector of real numbers which make up the urn.
k	A positive integer which specifies the number of items added to the original composition of the urn.

Value

The returned value is a vector of length equal to the length of `ysamp` plus `k`.

Examples

```
polyap(c(0,1),20)
```

wtpolyap *Polya Sampling from an Urn with Possibly Unequal Weights*

Description

Consider an urn containing a finite set of values along with their respective positive weights. An item is selected at random from the urn with probability proportional to its weight. Then it is returned to the urn and its weight is increased by one. The process is repeated on the adjusted urn. We continue until the total weight in the urn has been increased by k. The original composition of the urn along with the k selected values, in order, are returned.

Usage

```
wtpolyap(ysamp, wts, k)
```

Arguments

ysamp	A vector of real numbers which make up the urn.
wts	A vector of positive weights which defines the initial probability of selection.
k	A positive integer which specifies the number of Polya samples taken from the urn where after each draw the weight of the selected item is increased by one.

Value

The returned value is a vector of length equal to the length of the sample plus k.

Examples

```
wtpolyap(c(0,1,2),c(0.5,1,1.5),22)
```

Index