User guide for the onlineChange R package

Ning Dai, Yi Rong, Galin Jones, Jie Ding

May 7, 2019

This vignette gives a high level overview on how to use the `onlineChange` R package to detect a changepoint in the sequential setting and conduct simulation experiments to evaluate the performace of the proposed methods. It uses sequential Monte Carlo to perform Bayesian change point analysis, where the parameters before or after the change can be unknown. It also supports likelihood ratio based test to determine the stopping time (when change occurs).

We assume that the observations follow a distribution before the change, and a different distribution after the change. We have proposed a statistic-based stopping rule for continuous and binary data with known post-changed distributions. For continuous data with unknown post-changed distributions, we have developed a generalized Bayesian stopping rule implemented via the Sequential Monte Carlo algorithm.

Statistic-based Stopping Rule for Continuous Data

Consider a bivariate continuous distribution with a density defined up to a factor of normalizing constant:

$$f_\theta(\mathbf{x}) \propto \exp(\theta_1 x_1^2 x_2^2 + \theta_2 x_1^2 + \theta_3 x_2^2 + \theta_4 x_1 x_2 + \theta_5 x_1 + \theta_6 x_2),$$

where $\mathbf{x} = (x_1, x_2)^\top$ and $\theta = (\theta_1, \ldots, \theta_6)^\top$. Suppose the observations follow the bivariate distribution with $\theta_0 = (-1, -1, -1, 0, 0, 0)^\top$ before the change and $\theta_1 = (-1, -1, -1, 1, 0, 0)^\top$ after the change. The contour plots below illustrates the distribution with θ_0 on the left and θ_1 on the right.

Prior to the experiment, we sample 10,000 observations from each model using the robust adaptive Metropolis sampler with a burn-in of 10,000. During the experiment, we need only resample one observation at a time from the pre- or post-change pool, depending on whether the change has occurred.

```
#pre- and post-change parameter values
th.mat=matrix(nrow=2,ncol=6)
th.mat[1,]=c(-1, -1, -1, 0, 0, 0)
th.mat[2,]=c(-1, -1, -1, 1, 0, 0)

#log unnormalized probability function
ULP=function(x,th) sum(th*c(prod(x)^2,x[1]^2,x[2]^2,prod(x),x))

#prepare pre- and post-change pools of observations
obs=vector("list",2)
for(i in 1:2){
  set.seed(i+200)
  obs[[i]]=adaptMCMC::MCMC(ULP,n=2e4,init=c(0,1),scale=c(1,0.1),adapt=T,
                           acc.rate=0.3,th=th.mat[i,])$samples[1e4+1:1e4,]
}
```

We create a function `GEN(t)` that generates an observation at time t assuming a change occurs at $t = 15$. We also prepare the log unnormalized probability functions for computing the logarithmic score and the gradient and laplacian for computing the Hyvärinen score.

```
#data generater
GEN=function(t) {
  if(15>=t) { obs[[1]][sample.int(1e4,1),]
  } else obs[[2]][sample.int(1e4,1),]
}

#pre- and post-change log unnormalized probability functions
ULP0=function(x) ULP(x,th=th.mat[1,])
ULP1=function(x) ULP(x,th=th.mat[2,])

#gradient and laplacian of log unnormalized probability functions
GLP=function(x,th)
  c(2*th[1]*x[1]*x[2]^2+sum(c(th[2]*2,th[4])*x)+th[5],
                  2*th[1]*x[1]^2*x[2]+sum(c(th[4],th[3]*2)*x)+th[6])
LLP=function(x,th) 2*(th[1]*sum(x^2)+th[2]+th[3])

#pre- and post-change gradients and laplacians
GLP0=function(x) GLP(x,th=th.mat[1,])
GLP1=function(x) GLP(x,th=th.mat[2,])
LLP0=function(x) LLP(x,th=th.mat[1,])
LLP1=function(x) LLP(x,th=th.mat[2,])
```

We use the function `detect.stat` to detect the changepoint sequentially. The stopping time is returned when a change is detected. The parameter `alpha=0.5` controls the probability of false alarm at approximately 0.5. Suppose we know in advance that the change occurs between 10 and 25. To incorporate the prior information we set `nulower=10` and `nuupper=25`.

If `score` is omitted, the default Hyvärinen score is used, and the user needs to specify `GLP0,LLP0,GLP1,LLP1`, the gradients and laplacians of the pre- and post-change models. Alternatively, the user may specify the unnormalized probability functions `ULP0,ULP1`, based on which `detect.stat` will compute the gradients and laplacians. However, the alternative is less stable and hence not recommended.

```
detect.stat(GEN=GEN,alpha=0.5,nulower=10,nuupper=25,GLP0=GLP0,LLP0=LLP0,GLP1=GLP1,LLP1=LLP1)
#not recommended
detect.stat(GEN=GEN,alpha=0.5,nulower=10,nuupper=25,ULP0=ULP0,ULP1=ULP1)
```

Hyvärinen score has a positive tuning paramter with a default of 1. The user may change the tuning parameter values `par0,par1`.

```
#when using hyvarinen score, par0 and par1 are tuning parameters
detect.stat(GEN=GEN,alpha=0.5,nulower=10,nuupper=25,
            GLP0=GLP0,LLP0=LLP0,GLP1=GLP1,LLP1=LLP1,par0=0.2,par1=0.2)
```

To use the log score when the normalizing constants are unknown, `detect.stat` will compute the normalizing constants via numerical integration. The user needs to specify the dimension of an observation `lenx=2` and/or the lower and upper limits of integration `lower0,upper0,lower1,upper1` unless all the limits are infinite.

```
detect.stat(GEN=GEN,alpha=0.5,nulower=10,nuupper=25,score="log",ULP0=ULP0,ULP1=ULP1,lenx=2)
```

To use the log score when the normalizing constants are known, the user can pass the pre- and post-change negative log normalizing constants to `par0` and `par1`, respectively.

```
#calculate the negative log normalizing constants by numerical integration
par0=log(cubature::hcubature(function(x) exp(ULP0(x)),rep(-Inf,2),rep(Inf,2))$integral)
par1=log(cubature::hcubature(function(x) exp(ULP1(x)),rep(-Inf,2),rep(Inf,2))$integral)
#when using log score, par0 and par1 are negative log normalizing constants
detect.stat(GEN=GEN,alpha=0.5,nulower=10,nuupper=25,score="log",
            ULP0=ULP0,ULP1=ULP1,par0=par0,par1=par1)
```

Instead of a fixed changepoint, it would be useful to simulate various changepoints from a prior distribution and evaluate performance of the detection procedure. The function `sim.detect.stat` implements the simulation once and returns whether a false alarm has been raised, delay to detection and computation time. The user needs to create data generating functions `GEN0,GEN1` for the pre- and post-change models. The argument `detect.stat` takes a function that implements the statistic-based stopping rule: either `detect.stat` for continuous data or `detect.bin.stat` for binary data which we will introduce in the next section. If the lower and upper limits of the changepoint `nulower,nuupper` are specified, changepoint will be `nulower` plus a value simulated from the geometric distribution with `p=1/(1+(nulower+nuupper)/2)`. The default is `nulower=0` and `nuupper=18` which corresponds to the geometric distribution with `p=0.1`. Finally, the user should specify additional arguments to be passed to `detect.stat`.

```r
#pre- and post-change data generaters
GEN0=function() obs[[1]][sample.int(1e4,1),]
GEN1=function() obs[[2]][sample.int(1e4,1),]
#default hyvarinen score
sim.detect.stat(GEN0=GEN0,GEN1=GEN1,detect.stat=detect.stat,alpha=0.5,
                GLP0=GLP0,LLP0=LLP0,GLP1=GLP1,LLP1=LLP1)
#log score
sim.detect.stat(GEN0=GEN0,GEN1=GEN1,detect.stat=detect.stat,alpha=0.5,score="log",
                ULP0=ULP0,ULP1=ULP1,lenx=2)
```

To compare the performance of the statistic-based stopping rule using Hyvärinen score v.s. log score, we repeat the process 1000 times for various values of $\alpha \in (0, 1)$ and each score function. With the 1000 repetitions, we estimate the probability of false alarm, the average delay to detection and standard error, and record the average computation time.

```r
N=1000;K=20

#1. hyvarinen score
mat=matrix(nrow=K,ncol=4) #store summary statistics
alist=seq(0.05,0.9,length.out=K) #various values of alpha
set.seed(100)
for(k in 1:K)
{
  #default hyvarinen score
  tab=replicate(N,sim.detect.stat(GEN0=GEN0,GEN1=GEN1,detect.stat=detect.stat,
                nulower=10,nuupper=25,alpha=alist[k],
                GLP0=GLP0,LLP0=LLP0,GLP1=GLP1,LLP1=LLP1,par0=.2,par1=.2))
  #prob of false alarm, average delay to detection (se), average computation time
  mat[k,]=c(mean(tab[1,]),mean(tab[2,]),sd(tab[2,])/sqrt(N),mean(tab[3,]))
}
math=cbind(mat,alist)

#2. log score
mat=matrix(nrow=K,ncol=4) #store summary statistics
alist=seq(0.08,0.78,length.out=K) #various values of alpha
set.seed(100)
for(k in 1:K)
{
  #log score, normalizing constants are unknown
  tab=replicate(N,sim.detect.stat(GEN0=GEN0,GEN1=GEN1,detect.stat=detect.stat,
                nulower=10,nuupper=25,alpha=alist[k],score="log",
                ULP0=ULP0,ULP1=ULP1,lenx=2))
  #prob of false alarm, average delay to detection (se), average computation time
  mat[k,]=c(mean(tab[1,]),mean(tab[2,]),sd(tab[2,])/sqrt(N),mean(tab[3,]))
}
matl=cbind(mat,alist)
```

```r
#combine results using diff. scores into a data frame
dat=rbind(math,matl)
colnames(dat)=c("PFA","ADD","seADD","ACT","alpha")
dat=data.frame(dat)
dat$score=rep(c("hyvarinen","logarithmic"),each=K)
```

To visualize the results, we plot average delay to detection v.s. probability of false alarm, and log average computation time v.s. probability of false alarm. We observe that the stopping rule using Hyvärinen score achieves an average delay as low as using log score as the probability of false alarm ranges from 0.05 to 0.95, with a substantial improvement in computation.

```r
library(ggplot2)
#average delay to detection v.s. probability of false alarm
p=ggplot(dat,aes(x=PFA))+
  geom_line(aes(y=ADD,group=score,color=score))+
  geom_ribbon(aes(ymin=ADD-qnorm(0.975)*seADD,ymax=ADD+qnorm(0.975)*seADD,
                  group=score,fill=score),alpha=0.3)+
  scale_x_continuous(breaks=seq(0.1,0.9,0.2))+
  labs(x="prob of false alarm",y="average delay to detection")+
  theme(legend.position=c(0.82,0.86))
p
#log average computation time v.s. probability of false alarm
p=ggplot(dat,aes(x=PFA))+
  geom_line(aes(y=log(ACT),group=score,color=score))+
  scale_x_continuous(breaks=seq(0.1,0.9,0.2))+
  labs(x="prob of false alarm",y="log computation time (sec)")+
  theme(legend.position=c(0.82,0.5))
p
```

Statistic-based Stopping Rule for Binary Data

An RBM consists of m visible units $\mathbf{v}$ and n hidden units $\mathbf{h}$. The random variables take binary values $(\mathbf{v},\mathbf{h}) \in \{0,1\}^{m+n}$. The joint probability distribution is given by

$$f_{\mathbf{w},\mathbf{b},\mathbf{c}}(\mathbf{v},\mathbf{h}) \propto \exp\left(\sum_{i=1}^{n}\sum_{j=1}^{m} w_{ij}h_i v_j + \sum_{j=1}^{m} b_i v_j + \sum_{i=1}^{n} c_i h_i\right).$$

Suppose the pre- and post-change models are RBMs with different parameter values. Assume $m = 6$, $n = 2$, $\mathbf{w}_0 = (-1, -0.5, 0.5, 1, 0.5, -0.5; 0.5, 0, 0.5, -0.5, -0.5, -0.5)$, $\mathbf{b}_0 = (0, -0.5, 0.5, 0, 0, -0.5)^\top$, and $\mathbf{c}_0 = \mathbf{0}$. Then we flip the coefficients for the first two visible units to obtain $\mathbf{w}_1 = (-0.5, -1, 0.5, 1, 0.5, -0.5; 0, 0.5, 0.5, -0.5, -0.5, -0.5)$, $\mathbf{b}_1 = (-0.5, 0, 0.5, 0, 0, -0.5)^\top$, and $\mathbf{c}_1 = \mathbf{0}$.

We use Edwin Chen's example to illustrate the model. Suppose we have a set of six movies (Harry Potter, Avatar, LOTR 3, Gladiator, Titanic, and Glitter) and two hidden units underlying movie preferences – for example, two natural groups in our set of six movies appear to be Oscar winners and SF/fantasy, so we might hope that our hidden units will correspond to these categories – then our RBM would look like the table below with parameter value $(\mathbf{w}_0, \mathbf{b}_0, \mathbf{c}_0)$. Suppose a change occurs that swaps Harry Potter and Avatar. Then the post-change parameter value is given by $(\mathbf{w}_1, \mathbf{b}_1, \mathbf{c}_1)$.

	Harry Potter	**Avatar**	**LOTR 3**	**Gladiator**	**Titanic**	**Glitter**
Hidden 1	-1	-0.5	0.5	1	0.5	-0.5
Hidden 2	0.5	0	0.5	-0.5	-0.5	-0.5
Bias Unit	0	-0.5	0.5	0	0	-0.5

Prior to the experiments, we use Gibbs sampler to sample 20,000 observations from each model with a burn-in of 1,000,000. During the experiments, we need only resample one observation at a time from the pre- or post-change pool, depending on whether the change has occurred.

```
sigmoid=function(x) 1/(1+exp(-x))
#Gibbs sampler for the RBM moodel
gibbs=function(th,m=6,n=2,burn=1e6,N=2e4)
{
  w=matrix(th[1:(m*n)],nrow=n,byrow=T)
  b=th[m*n+(1:m)]
  c=th[m*n+m+(1:n)]
  mat=matrix(0,ncol=m+n,nrow=N)
  v=rep(0,m)
  for (i in 1:burn)
  {
    h=rbinom(n,1,sigmoid( w%*%v+c ))
    v=rbinom(m,1,sigmoid( t(w)%*%h+b ))
  }
  for (i in 1:N)
  {
    h=rbinom(n,1,sigmoid( w%*%v+c ))
    v=rbinom(m,1,sigmoid( t(w)%*%h+b ))
    mat[i,]=c(v,h)
  }
  mat
}

#pre- and post-change parameter values
th.mat=matrix(nrow=2,ncol=6*2+6+2)
```

```r
th.mat[1,]=c(-1,-0.5,0.5,1,0.5,-0.5, 0.5,0,0.5,-0.5,-0.5,-0.5, 0,-0.5,0.5,0,0,-0.5, 0,0)
th.mat[2,]=c(-0.5,-1,0.5,1,0.5,-0.5, 0,0.5,0.5,-0.5,-0.5,-0.5, -0.5,0,0.5,0,0,-0.5, 0,0)

#prepare pre- and post-change pools of observations
obs=vector("list",2)
for(i in 1:2){
  set.seed(i+200)
  obs[[i]]=gibbs(th.mat[i,])
}
```

Suppose the change occurs at $t=10$. We create a function `GEN(t)` that generates an observation at time t with a change at $t=10$. For binary data, the log unnormalized probability functions are required both when using the Hyvärinen score and the logarithmic score.

```r
#data generater
GEN=function(t) {
  if(10>=t) { obs[[1]][sample.int(2e4,1),]
  } else obs[[2]][sample.int(2e4,1),]
}
#log unnormalized probability function for the RBM model
ULP=function(x,th,m=6,n=2)
{
  w=matrix(th[1:(m*n)],nrow=n,byrow=T)
  b=th[m*n+(1:m)]
  c=th[m*n+m+(1:n)]
  v=x[1:m];h=x[m+ 1:n]
  return(sum(c(w%*%v)*h)+sum(b*v)+sum(c*h))
}
#pre- and post-change log unnormalized probability functions
ULP0=function(x) ULP(x,th=th.mat[1,])
ULP1=function(x) ULP(x,th=th.mat[2,])
```

We use the function `detect.bin.stat` to detect the changepoint sequentially. The stopping time is returned when a change is detected. The parameter `alpha=0.2` controls the probability of false alarm at approximately 0.2. Suppose we know in advance that the change cannot occur before $t=5$. Hence we set `nulower=5`.

If `score` is omitted, the default Hyvärinen score is used. Hyvärinen score has a positive tuning paramter with a default of 1. The user may change the tuning parameter values `par0,par1`.

```r
detect.bin.stat(GEN=GEN,alpha=0.2,nulower=5,ULP0=ULP0,ULP1=ULP1)
#when using hyvarinen score, par0 and par1 are tuning parameters
detect.bin.stat(GEN=GEN,alpha=0.2,nulower=5,ULP0=ULP0,ULP1=ULP1,par0=0.5,par1=0.5)
```

To use the log score when the normalizing constants are unknown, `detect.bin.stat` will compute the normalizing constants by summing over the sample space. The dimension of one observation `lenx=8` is required. The function supports two binary types. The default binary type is $\{1,0\}$. For the alternative binary type $\{1,-1\}$, the user should set `tbin=2`.

```r
detect.bin.stat(GEN=GEN,alpha=0.2,nulower=5,score="log",ULP0=ULP0,ULP1=ULP1,lenx=8)
```

To use the log score when the normalizing constants are known, the user can pass the pre- and post-change negative log normalizing constants to `par0` and `par1`, respectively.

```r
#compute normalizing constant by summing over the sample space
dom=as.matrix(expand.grid(rep(list(0:1),8)))
par0=log(sum(exp(apply(dom,1,ULP0))) )
par1=log(sum(exp(apply(dom,1,ULP1))) )
#when using log score, par0 and par1 are negative log normalizing constants
detect.bin.stat(GEN=GEN,alpha=0.2,nulower=5,score="log",ULP0=ULP0,ULP1=ULP1,par0=par0,par1=par1)
```

As is the case for continuous data, we can use the function `sim.detect.stat` to conduct change detection with simulated changepoint. For binary data, set `detect.stat=detect.bin.stat`.

```r
#pre- and post-change data generaters
GEN0=function() obs[[1]][sample.int(2e4,1),]
GEN1=function() obs[[2]][sample.int(2e4,1),]
#default hyvarinen score
sim.detect.stat(GEN0=GEN0,GEN1=GEN1,detect.stat=detect.bin.stat,alpha=0.2,nulower=5,
                ULP0=ULP0,ULP1=ULP1)
#log score
sim.detect.stat(GEN0=GEN0,GEN1=GEN1,detect.stat=detect.bin.stat,alpha=0.2,nulower=5,
                score="log",ULP0=ULP0,ULP1=ULP1,lenx=8)
```

To compare the performance of the statistic-based stopping rule using Hyvärinen score v.s. log score, we repeat the process 1000 times for various values of $\alpha \in (0,1)$ and each score function. With the 1000 repetitions, we estimate the probability of false alarm, the average delay to detection and standard error, and record the average computation time.

```r
N=1000;K=20
#1. hyvarinen score
mat=matrix(nrow=K,ncol=4) #store summary statistics
alist=seq(0.05,0.99,length.out=K) #various values of alpha
set.seed(100)
for(k in 1:K)
{
  #default hyvarinen score
  tab=replicate(N,sim.detect.stat(GEN0=GEN0,GEN1=GEN1,detect.stat=detect.bin.stat,
                alpha=alist[k],nulower=5,ULP0=ULP0,ULP1=ULP1))
  #prob of false alarm, average delay to detection (se), average computation time
  mat[k,]=c(mean(tab[1,]),mean(tab[2,]),sd(tab[2,])/sqrt(N),mean(tab[3,]))
}
math=cbind(mat,alist)
```

```
#2. log score
mat=matrix(nrow=K,ncol=4) #store summary statistics
alist=seq(0.05,0.99,length.out=K) #various values of alpha
set.seed(100)
for(k in 1:K)
{
  #log score, normalizing constants are unknown
  tab=replicate(N,sim.detect.stat(GEN0=GEN0,GEN1=GEN1,detect.stat=detect.bin.stat,
                alpha=alist[k],nulower=5,score="log",ULP0=ULP0,ULP1=ULP1,lenx=8))
  #prob of false alarm, average delay to detection (se), average computation time
  mat[k,]=c(mean(tab[1,]),mean(tab[2,]),sd(tab[2,])/sqrt(N),mean(tab[3,]))
}
matl=cbind(mat,alist)
```

We plot the results using the same codes as in the preceding section. The stopping rule using Hyvärinen score performs at par with the stopping rule using the log score, with a substantial improvement in

computation.

Bayesian Stopping Rule for Continuous Data

Suppose that a system changes from the standard normal distribution to the normal distribution with mean 2 and standard deviation 2. We do not know the mean or standard deviation of the post-change normal distribution but have prior information that both lie between 1 and 3. We also know that change is unlikely to occur after $t = 20$. For such change detection problems where data is continuous and the post-changed distribution is unknown, we can use the Bayesian stopping rule.

To prepare for the change detection experiment, we construct a function `GEN(t)` that generates an observation at time t assuming a change occurs at $t = 10$. We also prepare the log unnormalized probability functions for computing the logarithmic score and the gradient and laplacian for computing the Hyvärinen score.

```
#data generater
GEN=function(t){ if(10>=t) rnorm(1) else 2*rnorm(1)+2 }

#post- and pre-change log unnormalized probability functions, gradients and laplacians
ULP1=function(x,th) -(x-th[1])^2/2/th[2]^2
GLP1=function(x,th) -(x-th[1])/th[2]^2
LLP1=function(x,th) -1/th[2]^2
ULP0=function(x) ULP1(x,c(0,1))
GLP0=function(x) GLP1(x,c(0,1))
LLP0=function(x) LLP1(x,c(0,1))
```

We use the function `detect.bayes` to detect the changepoint sequentially. When a change is detected, `detect.bayes` returns the stopping time, the low ESS rate, and the average acceptance rate of the Metropolis-Hastings sampling in the move steps. If ESS never drops below the threshold, no move step is performed and the return value for the average acceptance rate will be `NaN`. We set the parameter `alpha=0.1` to control the probability of false alarm at approximately 0.1 and set `nuupper=20` to incorporate the prior information that change is unlikely to occur after $t = 20$. The dimension of the post-change parameter `lenth=2` is required. We specify the lower limit of the parameter `thlower=c(-Inf,0)` and omit the upper limit to use the default of infinity. To specify the prior distribution of the post-change parameter, we create a random generation function `GENTH` and a log unnormalized probability function `ULPTH` of the prior distribution. For simplicity we use independent uniform distributions from 1 to 3.

```
#let prior distribution of post-change mean and sd be independent uniform on (1,3)
#random generation function. n=sample size
GENTH=function(n) matrix(runif(n*2)*2+1,nrow=n)
#log unnormalized probability function
ULPTH=function(th) prod(th>1)*prod(th<3)
```

If `score` is omitted, the default Hyvärinen score is used, and the user needs to specify `GLP0,LLP0,GLP1,LLP1`, the gradients and laplacians of the pre- and post-change models. The prior distribution of the post-change parameter is specified through `GENTH,ULPTH`. If omitted, the default prior will be used which is standard normal on the unconstrained space. Hyvärinen score has a positive tuning paramter with a default of 1. The user may change the tuning parameter values `par0,par1`.

```
#user-specified prior distribution of post-change parameter
detect.bayes(GEN=GEN,alpha=0.1,nuupper=20,lenth=2,thlower=c(-Inf,0),GENTH=GENTH,ULPTH=ULPTH,
             GLP0=GLP0,LLP0=LLP0,GLP1=GLP1,LLP1=LLP1)
#default prior is used if GENTH,ULPTH are omitted
detect.bayes(GEN=GEN,alpha=0.1,nuupper=20,lenth=2,thlower=c(-Inf,0),
             GLP0=GLP0,LLP0=LLP0,GLP1=GLP1,LLP1=LLP1)
#when using hyvarinen score, par0 and par1 are tuning parameters
detect.bayes(GEN=GEN,alpha=0.1,nuupper=20,lenth=2,thlower=c(-Inf,0),GENTH=GENTH,ULPTH=ULPTH,
             GLP0=GLP0,LLP0=LLP0,GLP1=GLP1,LLP1=LLP1,par0=.6,par1=.6)
```

To use the log score when the normalizing constants are unknown, `detect.bayes` will compute the normalizing constants via numerical integration. The user needs to specify the dimension of one observation `lenx` and/or the lower and upper limits of integration `lower0,upper0,lower1,upper1` if not all infinite.

```
detect.bayes(GEN=GEN,alpha=0.1,nuupper=20,lenth=2,thlower=c(-Inf,0),score="log",
             ULP0=ULP0,ULP1=ULP1,lenx=1)
```

To use the log score when the normalizing constants are known, the user should pass the pre- and post-change negative log normalizing constants to `par0` and `par1`, respectively. Notice that `par1` is a function of the post-change parameter.

```
#post-change negative log normalizing constant
par1=function(th) log(2*pi)/2+log(th[2])
#pre-change negative log normalizing constant
par0=par1(c(0,1))
#when using log score, par0 and par1 are negative log normalizing constants
detect.bayes(GEN=GEN,alpha=0.1,nuupper=20,lenth=2,thlower=c(-Inf,0),score="log",
             GENTH=GENTH,ULPTH=ULPTH,ULP0=ULP0,ULP1=ULP1,par0=par0,par1=par1)
```

The function `sim.detect.bayes` conducts a simulation experiment of changepoint detection with known post-change distributions using the statistic-based stopping rule. It simulates a changepoint from the prior distribution, perform a change detection experiment, and returns whether a false alarm has been raised, delay to detection, computation time, low ESS rate, and average acceptance rate of the move step. The user needs to create data generating functions `GEN0,GEN1` for the pre- and post-change models. The true post-change parameter must be specified through `th0`. The argument `detect.bayes` takes a function that implements the statistic-based stopping rule. Currently only the above-mentioned function `detect.bayes` for continuous data is supported. If the lower and upper limits of the changepoint `nulower,nuupper` are specified, changepoint will be `nulower` plus a value simulated from the geometric distribution with `p=1/(1+(nulower+nuupper)/2)`. The default is `nulower=0` and `nuupper=18` which corresponds to the geometric distribution with `p=0.1`. Finally, the user should specify additional arguments to be passed to `detect.bayes`.

```
#post- and pre-change data generaters
GEN1=function(th) th[2]*rnorm(1)+th[1]
GEN0=function() GEN1(c(0,1))
#default hyvarinen score
sim.detect.bayes(GEN0=GEN0,GEN1=GEN1,th0=c(2,2),detect.bayes=detect.bayes,
                 nuupper=20,alpha=0.1,thlower=c(-Inf,0),GENTH=GENTH,ULPTH=ULPTH,
                 GLP0=GLP0,LLP0=LLP0,GLP1=GLP1,LLP1=LLP1,par0=.6,par1=.6)
#log score
sim.detect.bayes(GEN0=GEN0,GEN1=GEN1,th0=c(2,2),detect.bayes=detect.bayes,
                 nuupper=20,alpha=0.1,thlower=c(-Inf,0),GENTH=GENTH,ULPTH=ULPTH,
                 score="log",ULP0=ULP0,ULP1=ULP1,lenx=1)
```

To compare the performance of the statistic-based stopping rule using Hyvärinen score v.s. log score, we repeat the process 1000 times for various values of $\alpha \in (0, 1)$ and each score function. With the 1000 repetitions, we estimate the probability of false alarm, the average delay to detection and standard error, average computation time, low ESS rate, and average acceptance rate.

```r
N=1000;K=20
#1. hyvarinen score
mat=matrix(nrow=K,ncol=8) #store summary statistics
alist=seq(0.15,0.95,length.out=K) #various values of alpha
set.seed(100)
for(k in 1:K)
{
  #default hyvarinen score
  tab=replicate(N,sim.detect.bayes(GEN0=GEN0,GEN1=GEN1,th0=c(2,2),detect.bayes=detect.bayes,
               nuupper=20,alpha=alist[k],thlower=c(-Inf,0),GENTH=GENTH,ULPTH=ULPTH,
               GLP0=GLP0,LLP0=LLP0,GLP1=GLP1,LLP1=LLP1,par0=.6,par1=.6))
  #prob of false alarm, average delay to detection (se), average computation time,
  mat[k,]=c(mean(tab[1,]),mean(tab[2,]),sd(tab[2,])/sqrt(N),mean(tab[3,]),
           #low ESS rate (sd), average acceptance rate (sd)
           mean(tab[4,]),sd(tab[4,]),mean(tab[5,],na.rm=T),sd(tab[5,],na.rm=T))
}
math=cbind(mat,alist)

#2. log score
mat=matrix(nrow=K,ncol=8) #store summary statistics
alist=c(seq(0.25,.95,length.out=16),c(.96,.97,.98,.99))
set.seed(100)
for(k in 1:K)
{
  #log score, normalizing constant unknown
  tab=replicate(N,sim.detect.bayes(GEN0=GEN0,GEN1=GEN1,th0=c(2,2),detect.bayes=detect.bayes,
               nuupper=20,alpha=alist[k],thlower=c(-Inf,0),GENTH=GENTH,ULPTH=ULPTH,
               score="log",ULP0=ULP0,ULP1=ULP1,lenx=1))
  #prob of false alarm, average delay to detection (se), average computation time,
  mat[k,]=c(mean(tab[1,]),mean(tab[2,]),sd(tab[2,])/sqrt(N),mean(tab[3,]),
           #low ESS rate (sd), average acceptance rate (sd)
           mean(tab[4,]),sd(tab[4,]),mean(tab[5,],na.rm=T),sd(tab[5,],na.rm=T))
}
matl=cbind(mat,alist)

#combine results using diff. scores into a data frame
dat=rbind(math,matl)
colnames(dat)=c("PFA","ADD","seADD","ACT","LER","sdLER","AAR","sdAAR","alpha")
dat=data.frame(dat)
dat$score=rep(c("hyvarinen","logarithmic"),each=K)
```

We plot average delay to detection v.s. probability of false alarm, and log average computation time v.s. probability of false alarm using the codes demonstrated in the first section. The plots show that the stopping rule using Hyvärinen score achieves an average delay almost as low as using log score as the probability of false alarm ranges from 0.05 to 0.9, with a substantial improvement in computation.

It is important to check the quality of the sequential Monte Carlo sampling through low ESS rate and average acceptance rate of the move step. The following diagnostic plots show small low ESS rates and reasonable acceptance rates, which suggests no problem in sequential Monte Carlo.

```r
#low ESS rate v.s. probability of false alarm
p=ggplot(dat,aes(x=PFA))+
  geom_line(aes(y=LER,group=score,color=score))+
  scale_x_continuous(breaks=seq(0.1,0.9,0.2),limits=c(0.04,0.9))+
  labs(x="prob of false alarm",y="low ESS rate")+
  theme(legend.position=c(0.82,0.86))
p

#average acceptance rate v.s. probability of false alarm
p=ggplot(dat,aes(x=PFA))+
  geom_line(aes(y=AAR,group=score,color=score))+
  scale_x_continuous(breaks=seq(0.1,0.8,0.2),limits=c(0.04,0.8))+
  labs(x="prob of false alarm",y="average acceptance rate")+
  theme(legend.position=c(0.82,0.5))
p
```

Statistic-based Stopping Rule for RBMs

Suppose the pre- and post-change models are RBMs with same numbers of visible layers and hidden layers. The differences between them are parameter values. We have a training set of the pre-change model X_0 and a training set of the pre-change model X_1, which we could use to train the RBMs and get the parameter values. We use the package RBM to do this. We use the MNIST, a hand-written numbers dataset as an example.

```r
# Load devtools
```

```r
library(devtools)
# Install RBM
# install_github("TimoMatzen/RBM")
# load RBM
library(RBM)
# Load the MNIST data
data(MNIST)
# re-arrange the MNIST$trainX and MNIST$trainY
train.Y0.index <- which(MNIST$trainY==0); length(train.Y0.index)
train.Y1.index <- which(MNIST$trainY==1); length(train.Y1.index)

train.X0 <- MNIST$trainX[train.Y0.index,]
train.X1 <- MNIST$trainX[train.Y1.index,]
# re-arrange the MNIST$testX and MNIST$testY
test.Y0.index <- which(MNIST$testY==0); length(test.Y0.index)
test.Y1.index <- which(MNIST$testY==1); length(test.Y1.index)

test.X0 <- MNIST$testX[test.Y0.index,]
test.X1 <- MNIST$testX[test.Y1.index,]

# randomly sample 30 out of 784 features
samp <- sample(784,30)
trainX <- MNIST$trainX[,samp]

# the length of an observation is 30 and we use 100 observations to train each RBM
modelRBM.X0 <- RBM(x = trainX[train.Y0.index,][1:100,], n.iter = 1000, n.hidden = 50,
size.minibatch = 10)
modelRBM.X1 <- RBM(x = trainX[train.Y1.index,][1:100,], n.iter = 1000, n.hidden = 50,
size.minibatch = 10)

# compute the normalizng parameter z for log-score
VisToHid <- function(vis, weights, y, y.weights) {
  V0 <- vis
  if ( is.null(dim(V0))) {
    V0 <- matrix(V0, nrow= length(V0))
  }
  if(missing(y) & missing(y.weights)) {
    H <- 1/(1 + exp(-( V0 %*% weights)))
  } else {
    Y0 <- y
    H <- 1/(1 + exp(- ( V0 %*% weights + Y0 %*% y.weights)))
  }
  return(H)
}
# function VisToHid is in InternalFun.R file from package RBM
E.X0 <- 1:nrow(trainX[train.Y0.index,][1:100,])
for (k in 1 : nrow(trainX[train.Y0.index,][1:100,])){
  t <- matrix(trainX[train.Y0.index,][1:100,][k,], nrow = 1)
  vis <- cbind(1, t[1,, drop = FALSE])
  h0 <- VisToHid(vis, modelRBM.X0$trained.weights)
  E.X0[k] <- -sum(vis %*% modelRBM.X0$trained.weights %*% t(h0))
}

Z.X0 = sum(exp(E.X0))

E.X1 <- 1:nrow(trainX[train.Y1.index,][1:100,])
for (k in 1 : nrow(trainX[train.Y1.index,][1:100,])){
  t <- matrix(trainX[train.Y1.index,][1:100,][k,], nrow = 1)
  vis <- cbind(1, t[1,, drop = FALSE])
  h0 <- VisToHid(vis, modelRBM.X1$trained.weights)
  E.X1[k] <- -(vis %*% modelRBM.X1$trained.weights %*% t(h0))
}

Z.X1 = sum(exp(E.X1))

# build ULP0, ULP1
ULP <- function(x,w){
  vis <- cbind(1, x[1,, drop = FALSE])
  hid <- VisToHid(vis, w)
  out <- -(vis %*% w %*% t(hid))
  return(out)
}

ULP0 <- function(x) ULP(x,modelRBM.X0$trained.weights)
ULP1 <- function(x) ULP(x,modelRBM.X1$trained.weights)

# we choose 50 observations from the test set, at t=17 the class changes from 0 to 1
testX.reordered <- MNIST$testX[order(MNIST$testY),]
testY.reordered <- MNIST$testY[order(MNIST$testY)]
length(which(testY.reordered==0))
GEN <- function(t){
  data <- testX.reordered[180:230,samp]
  datum <- data[t,]
  x <- matrix(datum,nrow=1)
  return(x)
}
```

Note that this method is very sensitive to the outlier.

We use `detect.stat` to detect the changepoint sequentially.

```
detect.stat(GEN=GEN,score="log",alpha=0.1,nuupper=100,
            ULP0=ULP0,ULP1=ULP1,par0 = log(Z.X0),par1 = log(Z.X1))

detect.stat(GEN=GEN,alpha=0.01,nuupper=100,
            ULP0=ULP0,ULP1=ULP1,par0 = 0.01,par1 = 0.01)
```

Baysian Stopping Rule with Unknown Pre- and Post-change Parameters

We can extend the Baysian stopping rule to the circumstances when both the pre- and post-change parameters are unknown. Like before, we construct a function GEN(t) that generates an observation at time t assuming a change occurs at t=10. We also prepare the log unnormalized probability functions for computing the logarithmic score and the gradient and laplacian for computing the Hyvarinen score.

```
#data generater
GEN=function(t){ if(10>=t) rnorm(1) else 2*rnorm(1)+2 }

#post- and pre-change log unnormalized probability functions, gradients and laplacians
ULP1=function(x,th) -(x-th[1])^2/2/th[2]^2
GLP1=function(x,th) -(x-th[1])/th[2]^2
LLP1=function(x,th) -1/th[2]^2

ULP0=function(x,th) -(x-th[1])^2/2/th[2]^2
GLP0=function(x,th) -(x-th[1])/th[2]^2
LLP0=function(x,th) -1/th[2]^2
```

We use the function `detect.bayes.unknown.pre` to detect the changepoint sequentially, which returns similar outputs as function `detect.bayes`.

```
detect.bayes.unknown.pre(GEN=GEN,alpha=0.5, nuupper=100,lenth1=2,
                lenth0=2, thlower1=c(-Inf,0),thlower0=c(-Inf,0),
                score="logarithmic",ULP0=ULP0,ULP1=ULP1,lenx=1)

detect.bayes.unknown.pre(GEN=GEN,alpha=0.5, nuupper=100,lenth1=2,
                lenth0=2, thlower1=c(-Inf,0),thlower0=c(-Inf,0),
                score="logarithmic",ULP0=ULP0,ULP1=ULP1,
                par1=par1, par0=par0, lenx=1)

detect.bayes.unknown.pre(GEN=GEN,alpha=0.5,,nuupper=100,lenth1=2,
                lenth0=2, thlower1=c(-Inf,0),thlower0=c(-Inf,0),
                score="hyvarinen",ULP0=ULP0,ULP1=ULP1,
                par1=0.1, par0=0.1)
```

We plot the average delay to detection v.s. probability of false alarm, and log average computation time v.s. probability of false alarm,low ESS rates v.s. probability of false alarm, average acceptance rate v.s. probability of false alarm using the same codes as before:

Baysian Stopping Rule with RBMs

Suppose both the pre- and post-change models are RBMs with different parameters. There would be too much calcultion if we use these RBMs directly. Instead, we use the energy of each RBM as our observation.

```
modelRBM <- RBM(x = MNIST$trainX, n.iter = 1000, n.hidden = 1000, size.minibatch = 10)

# compute the energy for each observation
E <- 1:nrow(test)
for (k in 1 : nrow(test)){

  t <- matrix(test[k,], nrow = 1)
  vis <- cbind(1, t[1,, drop = FALSE])
  h0 <- VisToHid(vis, modelRBM$trained.weights)
  E[k] <- -sum(vis %*% modelRBM$trained.weights %*% t(h0))
}
```

We plot the energies:

```
gg.data_rbm01 <- as.data.frame(cbind(1:length(data_rbm01),data_rbm01))
colnames(gg.data_rbm01) <- c('t','Energy')

library(ggplot2)
gg1 = ggplot(gg.data_rbm01, aes(y=Energy, x=t)) +
  geom_point(size=2,shape=1) +
  #ggtitle('Observed Energy at Time t') +
  geom_vline(aes(xintercept=197),col="red")+
  scale_y_continuous(limits = c(-2,4))
```

```
gg1
```

Obviously, we can see the distribution of the energy changes at time t=197.

Taking the energy of each point as our observation, here we make different prior assumptions of the pre- and post- change model and apply the function `detect.bayes.unknown.pre`.

Supposing the pre- and post-change models follow normal distribution

```
# alter the test observations for easier calculation
data_rbm01 <- E[c(test.Y0.index, test.Y1.index)]/100+2
data_rbm01_cut <-data_rbm01[150:250]
# generate the test observations
GEN=function(t) { data_rbm01_cut[t] }
# post- and pre-change log unnormalized probability functions, gradients and laplacians for
normal distribution
ULP1=function(x,th) -(x-th[1])^2/2/th[2]^2
GLP1=function(x,th) -(x-th[1])/th[2]^2
LLP1=function(x,th) -1/th[2]^2
ULP0=function(x,th) -(x-th[1])^2/2/th[2]^2
GLP0=function(x,th) -(x-th[1])/th[2]^2
LLP0=function(x,th) -1/th[2]^2
par1=function(th) log(2*pi)/2+log(th[2])
par0=function(th) log(2*pi)/2+log(th[2])

GENTH0=function(n) matrix(runif(n*2),nrow=n)
ULPTH0=function(th) prod(th>0)*prod(th<1)
GENTH1=function(n) matrix(runif(n*2)*2,nrow=n)
ULPTH1=function(th) prod(th>0)*prod(th<2)

# logarithmic score with unknown normalizing parameter:
detect.bayes.unknown.pre(GEN=GEN,alpha=0.5 ,nulower=20, nuupper=100,lenth1=2,
                lenth0=2, thlower1=c(-Inf,0),thlower0=c(-Inf,0),
                score="logarithmic",ULP0=ULP0,ULP1=ULP1,lenx=1)

# logarithmic score with known normalizing parameter:
detect.bayes.unknown.pre(GEN=GEN,alpha=0.5,nulower=20,nuupper=100,lenth1=2,
                lenth0=2, thlower1=c(-Inf,0),thlower0=c(-Inf,0),
                score="logarithmic",ULP0=ULP0,ULP1=ULP1,
                par1=par1,par0=par0,lenx=1)
```

```
# logarithmic score withknown normalizing parameter and GENTH0, ULPTH0, GENTH1, ULPTH1:
detect.bayes.unknown.pre(GEN=GEN,alpha=0.1,nulower=20,nuupper=100,lenth1=2,
                lenth0=2, thlower1=c(-Inf,0),thlower0=c(-Inf,0),
                score="logarithmic",ULP0=ULP0,ULP1=ULP1,
                par1=par1,par0=par0,lenx=1,
                GENTH0 = GENTH0, GENTH1 = GENTH1,
                ULPTH0 = ULPTH0, ULPTH1 = ULPTH1)

# hyvarinen score with known par0, par1, GLP0, GLP1, LLP0, LLP1
detect.bayes.unknown.pre(GEN=GEN,alpha=0.4,nulower=20,nuupper=100,lenth1=2,
                lenth0=2, thlower1=c(-Inf,0),thlower0=c(-Inf,0),
                score="hyvarinen",GLP0=GLP0,LLP0=LLP0,GLP1=GLP1,LLP1=LLP1,
                par1=0.1, par0=0.1)

# hyvarinen score with known par0, par1, ULP0, GENTH0, ULPTH0, GENTH1, ULPTH1
detect.bayes.unknown.pre(GEN=GEN,alpha=0.4,nulower=20,nuupper=100,lenth1=2,
                lenth0=2, thlower1=c(-Inf,0),thlower0=c(-Inf,0),
                score="hyvarinen",ULP0=ULP0,ULP1=ULP1,
                par1=0.1, par0=0.1,
                GENTH0 = GENTH0, GENTH1 = GENTH1,
                ULPTH0 = ULPTH0, ULPTH1 = ULPTH1)
```

We can also suppose the energy follows cauchy distribution:

```
# post- and pre-change log unnormalized probability functions for cauchy distribution
ULP1=function(x,th) -log(1+((x-th[1])/th[2])^2)
ULP0=function(x,th) -log(1+((x-th[1])/th[2])^2)
par1=function(th) pi*th[2]
par0=function(th) pi*th[2]

detect.bayes.unknown.pre(GEN=GEN,alpha=0.5 ,nulower=20, nuupper=100,lenth1=2,
                lenth0=2, thlower1=c(-Inf,0),thlower0=c(-Inf,0),
                score="logarithmic",ULP0=ULP0,ULP1=ULP1,lenx=1)

detect.bayes.unknown.pre(GEN=GEN,alpha=0.7,nulower=20,nuupper=100,lenth1=2,
                lenth0=2, thlower1=c(-Inf,0),thlower0=c(-Inf,0),
                score="hyvarinen",ULP0=ULP0,ULP1=ULP1,
                par1=0.1, par0=0.1)
```