distance prior

When using dirichlet process to model the prior of partitions. It is convenient to use an equivalent representation

$$P(\pi) \propto \prod_{i=1}^{K} \theta\Gamma(q_i).........(1)$$

.

Assuming there are K clusters and q_i is number of elements in cluster i. Now we only interested in the case that K is known(fixed). We want to approximate the posterior inference of clustering based on dirichlet process prior through a random distance matrix manner.

An intuitive idea is that we sample partitions of data using the probability in (1). We then generate a distance matrix $\hat{D} = \{\hat{d_{ij}}\}$ based on the sampled partition. $\hat{d_{ij}}$ is 1 if point i and j come from different cluster, 0 otherwise. Then we convert the prior of partitions to prior of distance matrices. And using those prior on distance matrices should have approximately same posterior inference as using dirichlet process. Unfortunately, this is computational intractable. For example (below)

```
##function to calculate number of possible ways to partition n subjects to k groups
count_partitions = function(n,k){
  #! param n, number of elements
  #! param k, number of groups
  # return number of partitions of n elements into k groups
  dp = matrix(0, nrow = n + 1, ncol = k + 1)

  for (i in 2:(n + 1)){
    for (j in 2:(k + 1)){
      if (j == 2 || i == j){
        dp[i,j] = 1
      }
      else{
        dp[i,j] = (j - 1) * dp[i-1,j] + dp[i-1,j-1]
      }
    }
  }
  return(dp[n + 1 , k + 1])
}

count_partitions(100,5)
```

```
## [1] 6.573841e+67
```

We found that the possible number of ways to classify 100 subjects to 5 sets is 6.5e+67. Also there is not an explicit way to generate partitions, partitions are generated sequentially. Given that amount of possible paritions, both time complexity and storage problem make it infeasible.

Another simple idea is that no matter what prior we put on partitions, the marginal probability that subject i and j under no information about their similarity belong to same cluster is $1/K$(K: number of clusters)

$$\sum_{partition} P(\text{subject } i \text{ and subject } j \text{ belong to same cluster}|\text{partition}) Prior(\text{partiton}) = 1/K$$

For any proper prior on the possible paritions that classify n subjects into K groups. As given the parition label of subject i, the chances that subject j has the same label is $1/K$

Then we could sample distance matrix $\hat{D} = \{\hat{d_{ij}}\}$, that $\hat{d_{ij}}$ is bernoulli distributed with $p = 1 - 1/K$.

Below we have functions to generate $p(c_i = c_j|x)$ based on random distance method. c_i, c_j are the parition label of subject i and j, x is the data

```
mimic_dp = function(n, K){
  #! param n, number of elements
  #! param K, number of clusters
  alp = K - 1
  if(alp < 1){
    print("error, K should be greater than 1")
    return(0)
  }
  p_ = rep(0,n)
  #number of elements added, start with 1 element
  num = 1
  #number of elements per group
  num_per_cl = rep(0,n)
  num_per_cl[1] = 1
  #initial prob for assigning class under dirichlet process
  p_[1] = num_per_cl[1] / (num + alp)
  p_[2] = alp / (num + alp)

  #class label
  cl_label = rep(0, n)
  cl_label[0] = 1
  pool = 1:n
  for(i in 2:n){
    idx_ = sample(pool,1,F,p_)
    num_per_cl[idx_] = num_per_cl[idx_] + 1
    num = num + 1
    p_ = num_per_cl / (num + alp)
    # position for staring new class
    pos = which(p_ == 0)[1]
    p_[pos] = alp / (num + alp)
    cl_label[i] = idx_
  }

  noise = 1 * outer(cl_label,cl_label,"==")
  result = list()
  result[[1]] = noise
  result[[2]] = max(cl_label)
  return(result)

}



boot = function(n, K, x, B, wt){
  p_ = 1 / K
  D_ = as.matrix(dist(x))
  rand_boot = rep(0,B)
  Aboot <- matrix(0,n,n)

  for(b_ in 1:B){
    noise = matrix(rbinom(n^2, prob=1-p_,size=1) + 1,nrow=n) * 0.5
    ## to make noise be a symmetric matrix
    up_ = 1 * upper.tri(noise, diag = FALSE)
    noise = noise * up_
    noise = noise + t(noise)
    # noise = mimic_dp(n,K)
    weight = matrix(rexp(n ^ 2, wt), ncol = n)
    #weight = matrix(rep(1 / wt,n^2), ncol = n)
    noise = noise * weight
    bar = noise + D_
    dst.star <- as.dist( bar )
    hc = hclust(dst.star,"complete")
    clus = cutree(hc, k = K)
    # clus = pam(bar,K,diss = T)$clustering
    rand_boot[b_] = adjustedRandIndex(clus, true.clust)
    tmp = outer(clus,clus, "==")
    Aboot <- Aboot + tmp/B

  }
  res = list()
  res[[1]] = Aboot
  res[[2]] = rand_boot
  return(res)
}
```

```
boot2 = function(n, K, x, B,wt){
  p_ = 1 / K
  D_ = as.matrix(dist(x))
  rand_boot = rep(0,B)
  Aboot <- matrix(0,n,n)
  count = 0
  for(b_ in 1:B){
    result = mimic_dp(n,K)
    noise = result[[1]]
    numc = result[[2]]
    # if(numc != K){
    #   next
    # }
    count = count + 1
    # weight = matrix(runif(n ^ 2, 0, 20), ncol = n)
    weight = matrix(rexp(n ^ 2, wt), ncol = n)
    noise = noise * weight
    bar = noise + D_
    dst.star <- as.dist( bar )
    hc = hclust(dst.star,"complete")
    clus = cutree(hc, k = K)
    # clus = pam(bar,K,diss = T)$clustering
    rand_boot[b_] = adjustedRandIndex(clus, true.clust)
    tmp = outer(clus,clus, "==")
    Aboot <- Aboot + tmp

  }
  Aboot = Aboot/ count
  rand_boot = rand_boot[which(rand_boot > 0)]
  res = list()
  res[[1]] = Aboot
  res[[2]] = rand_boot
  return(res)
}

pij = function(xi,xj,MU,sig,prob){
  dsi = dnorm(xi,mean = MU, sd = sig)
  dsj = dnorm(xj,mean = MU, sd = sig)
  probi = dsi * prob
  probj = dsj * prob
  return(sum(probi * probj) / (sum(probi) * sum(probj)))
}


post = function(D, clus, K){
  n = ncol(D)
  class_label = list()
  prop = rep(0,n)
  for(i in 1:K){
    class_label[[i]] = which(clus == i)
    prop[i] = length(class_label[[i]])
  }

  S = matrix(0,n,K)
  for(i in 1:n){
    tmp = D[i,]
    cur_min = 0
    for(j in 1:K){
      tmpj = tmp[class_label[[j]]]
      S[i,j] = sum( tmpj ) / prop[j]
      cur_min = min(cur_min, S[i,j])
    }
    cur_sum = 0
    for(j in 1:K){
      tmpp = cur_min - S[i,j]
      S[i,j] = exp(cur_min - S[i,j])
      cur_sum = cur_sum + S[i,j]
    }
    for(j in 1:K){
      S[i,j] = S[i,j] / cur_sum
    }
  }
  return(S %*% t(S))

}


boot1 = function(n,K, x, B, wt){
  D_ = as.matrix(dist(x))
  est = pam(D_,K,diss = T)
  mds = as.numeric(est$medoids)
  mlabel = rep(0,n)
  for(II in 1:K){
    mlabel[which(est$clustering == II)] = mds[II]
  }
  #tmp_res = pam(D_,K,diss = T)
  #clus = tmp_res$clustering
  #P = post(D_, clus, K)
  #for(i in 1:n){
    #P[i,i] = 1
  #}
  rand_boot = rep(0,B)
  Aboot <- matrix(0,n,n)
  D <- matrix(0,n,n)
  count = 0
  for(b_ in 1:B){
    #noise = rexp(n,wt)
    #NS = 2 * outer(noise, noise , "+")
    smds = mds[sample(1:K,n,replace = T)]
    noise = rexp(n * 2,wt)
    noise = matrix(noise,nrow = n)
    NS = matrix(0,n,n)
    for(i in 1:n){
      others = max(noise[i,1], noise[i,2]) + 10
      NS[i,] = others
      NS[i,smds[i]] = noise[i,1]
      NS[i,mlabel[i]] = noise[i,2]
    }
    NS[upper.tri(NS)] = 0
    diag(NS) = 0
    NS = NS + t(NS)
    NS = NS * 2
    for(i in 1:n){
      for(j in 1:n){
        #D[i,j] = (1 - rbinom(1,1, P[i,j])) * wt
        D[i,j] = ((D_[i,j])^2 + NS[i,j])
      }
    }
    count = count + 1
    #dst.star <- as.dist( D )
    #hc = hclust(dst.star,"average")
    #clus = cutree(hc, k = K)
    clus = pam(D,K,diss = T)$clustering
    rand_boot[b_] = adjustedRandIndex(clus, true.clust)
    tmp = outer(clus,clus, "==")
    Aboot <- Aboot + tmp
  }
  Aboot = Aboot/ count
  rand_boot = rand_boot[which(rand_boot > 0)]
  res = list()
  res[[1]] = Aboot
  res[[2]] = rand_boot
  return(res)
}



boot3 = function(n, K, x, B, wt, sig, prob){

    D_ = as.matrix(dist(x))
    tmp_res = pam(D_,K,diss = T)
    clus = tmp_res$clustering
    centers = x[as.numeric(tmp_res$medoids)]

    tmp_res = kmeans(x, centers)
    MU = tmp_res$centers

    p_ = matrix(0,n,n)
    for(i in 1:(n -1) ){
      for(j in (i + 1):n){
        p_[i,j] = pij(x[i],x[j], MU, sig, prob)
      }
    }
    p_ = diag(n) + p_ + t(p_)

    rand_boot = rep(0,B)
    Aboot <- matrix(0,n,n)
    D <- matrix(0,n,n)
    count = 0
    for(b_ in 1:B){
      for(i in 1:n){
        for(j in 1:n){
          D[i,j] = (1 - rbinom(1,1, p_[i,j])) * wt
        }
      }
    count = count + 1
    dst.star <- as.dist( D )
    hc = hclust(dst.star,"average")
    clus = cutree(hc, k = K)
    rand_boot[b_] = adjustedRandIndex(clus, true.clust)
    tmp = outer(clus,clus, "==")
    Aboot <- Aboot + tmp

  }
  Aboot = Aboot/ count
  rand_boot = rand_boot[which(rand_boot > 0)]
  res = list()
  res[[1]] = Aboot
  res[[2]] = rand_boot
  return(res)
}

boot4 = function(x, K, B, gm){
  #mu = c(1,1)
  #sig = diag(2) * 0.1
  n = length(x)
  rand_boot = rep(0,B)
  Aboot <- matrix(0,n,n)
  D_ = as.matrix(dist(x))
  for(b_ in 1:B){
    noise = matrix(0, n, n)
    noise[1,2:n] = runif(n - 1,0,1)
    noise[1,1] = 0
    for(j in 2:n){
        if(j < 3){
          next
        }
        for(i in 2:(j -1)){
          a = 0
          b = 2000
          for(t_ in 1:(i - 1)){
            if(abs(noise[t_,i] - noise[t_,j]) > a){
              a = abs(noise[t_,i] - noise[t_,j])
            }
            if(noise[t_,i] + noise[t_,j] < b){
              b = noise[t_,i] + noise[t_,j]
            }
          }
          tmp_e = runif(1,a,b)
          noise[i,j] = tmp_e
        }
      }


    noise = noise + t(noise)
    noise = noise * gm #sum(D_) / (n * (n - 1))
    bar = noise + D_
    dst.star <- as.dist( bar )
    hc = hclust(dst.star)
    clus = cutree(hc, k = K)
    # clus = pam(bar,K,diss = T)$clustering
    rand_boot[b_] = adjustedRandIndex(clus, true.clust)
    tmp = outer(clus,clus, "==")
    Aboot <- Aboot + tmp/B
  }
  res = list()
  res[[1]] = Aboot
  res[[2]] = rand_boot
  return(res)
}
```

Also we have functions to generate $p(c_i = c_j|x)$ based on dirichlet process

```
DP_bayes = function(x, mcmc, prior, K, f){
  n = length(x)
  state = NULL
  fit = DPlmm( fixed=x~1, random=~1|f, mcmc=mcmc, state=state, status=TRUE, prior=prior )
  u <- fit$save.state$randsave[,-(n+1)]
  ok <- fit$save.state$thetasave[,5] == K
  u <- fit$save.state$randsave[ok,-(n+1)]
  B <- sum(ok)
  prob <- matrix(0, B, K)
  ABayes <- matrix(0, n,n )
  rand_bayes = rep(0,B)
  for( b in 1:B )
  {
    tmp <- outer( u[b,], u[b,], "==" )
    ABayes <- ABayes + tmp/B
    tmp = u[b,]
    tt <- match( tmp, unique(tmp) )
    rand_bayes[b] = adjustedRandIndex(tt,true.clust)
    prob[b,] = table(tmp) / n
  }
  res = list()
  res[[1]] = ABayes
  res[[2]] = rand_bayes
  res[[3]] = prob
  return(res)
}
```

Then we compare the results of two methods

```
library(DPpackage)  ## for MCMC
```

```
## Warning: package 'DPpackage' was built under R version 3.4.3

##

## DPpackage 1.1-7.4

##

## Copyright (C) 2006 - 2012, Alejandro Jara

## Department of Statistics

## P.U. Catolica de Chile

##

## Support provided by Fondecyt

## 11100144 grant.

##
```

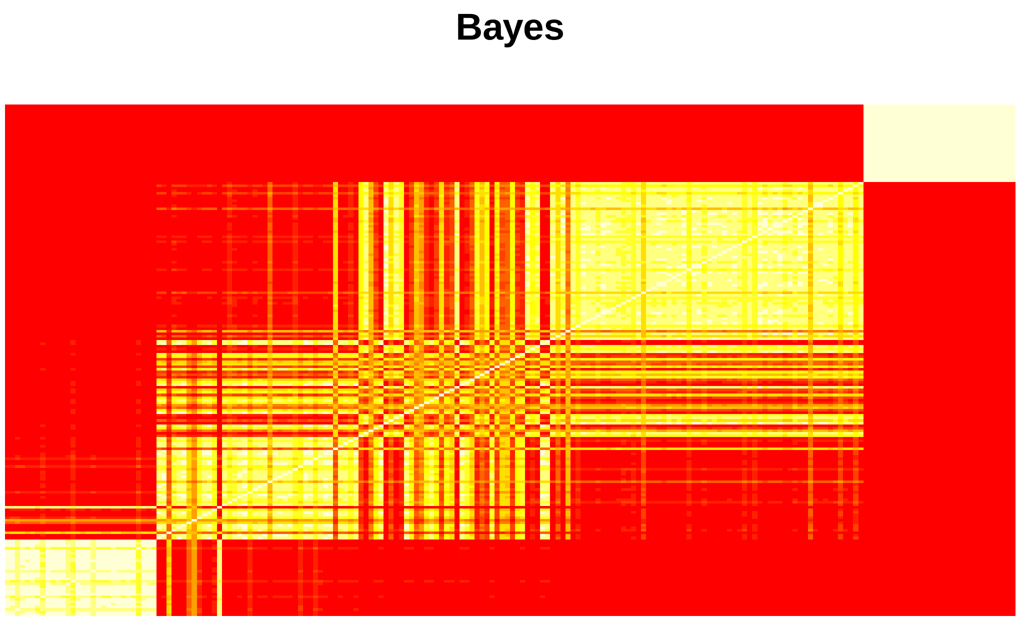
```
library(cluster)
library(mclust)
```

```
## Package 'mclust' version 5.3

## Type 'citation("mclust")' for citing this R package in publications.
```

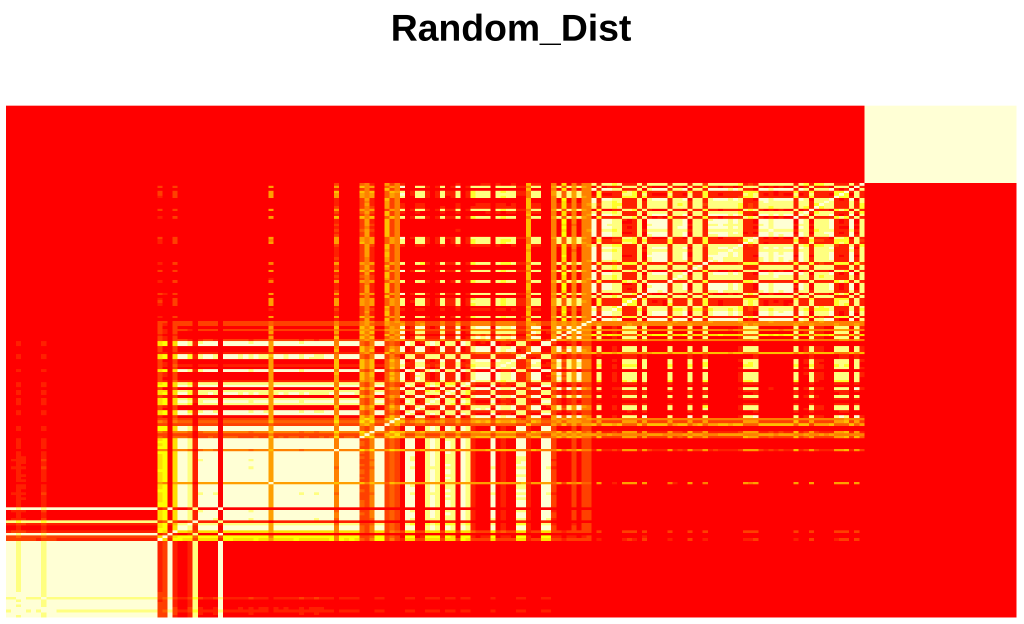
```r
set.seed(75751)

# a toy data set
size = 2
mu <- 2*c( rep(-3,15 * size), rep(-1,20 * size), rep(0,20 * size), rep(1,30 * size), rep(5,15 * size) )
sig <-1
true.clust <- match(mu, unique(mu))
prob = c(0.15,0.2,0.2,0.3,0.15)
MU = 2 * c(-3,-1,0,1,5)
n <- length(mu)
y <- rnorm(n,mean=mu, sd=sig)
f <- as.factor(1:n)


# try DPlmm

#prior <- list( alpha=1, nu0=1, tau1=1, tau2=1, tinv=diag(1), mub=0, Sb=diag(1) )
prior <- list( alpha=1, nu0=1, tau1=0.1, tau2=0.1, tinv=diag(1), mub=0, Sb=diag(1) )

nburn <-50
nsave <- 2e3
nskip <-100
ndisplay <-100
mcmc <- list(nburn=nburn,nsave=nsave,nskip=nskip,ndisplay=ndisplay)


######data set up complete

###get results from random_distance and dirichlet process
K = 5
D_ = as.matrix(dist(y))
tmp_res = pam(D_,K,diss = T)
clus = tmp_res$clustering
P = post(D_, clus, K)
for(i in 1:n){
  P[i,i] = 1
}
#Aboot = P
res_dist = boot1(length(y),5,y,500,0.5)
# res_dist = boot4(y,5,100,5)
Aboot = res_dist[[1]]
rand_boot = res_dist[[2]]
res_bayes = DP_bayes(y, mcmc, prior, 5, f)
```

```
##
## MCMC scan 100 of 2000 (CPU time: 0.685 s)
## MCMC scan 200 of 2000 (CPU time: 1.375 s)
## MCMC scan 300 of 2000 (CPU time: 2.058 s)
## MCMC scan 400 of 2000 (CPU time: 2.749 s)
## MCMC scan 500 of 2000 (CPU time: 3.437 s)
## MCMC scan 600 of 2000 (CPU time: 4.121 s)
## MCMC scan 700 of 2000 (CPU time: 4.813 s)
## MCMC scan 800 of 2000 (CPU time: 5.491 s)
## MCMC scan 900 of 2000 (CPU time: 6.179 s)
## MCMC scan 1000 of 2000 (CPU time: 6.865 s)
## MCMC scan 1100 of 2000 (CPU time: 7.560 s)
## MCMC scan 1200 of 2000 (CPU time: 8.241 s)
## MCMC scan 1300 of 2000 (CPU time: 8.928 s)
## MCMC scan 1400 of 2000 (CPU time: 9.633 s)
## MCMC scan 1500 of 2000 (CPU time: 10.336 s)
## MCMC scan 1600 of 2000 (CPU time: 11.034 s)
## MCMC scan 1700 of 2000 (CPU time: 11.717 s)
## MCMC scan 1800 of 2000 (CPU time: 12.401 s)
## MCMC scan 1900 of 2000 (CPU time: 13.102 s)
## MCMC scan 2000 of 2000 (CPU time: 13.803 s)
```

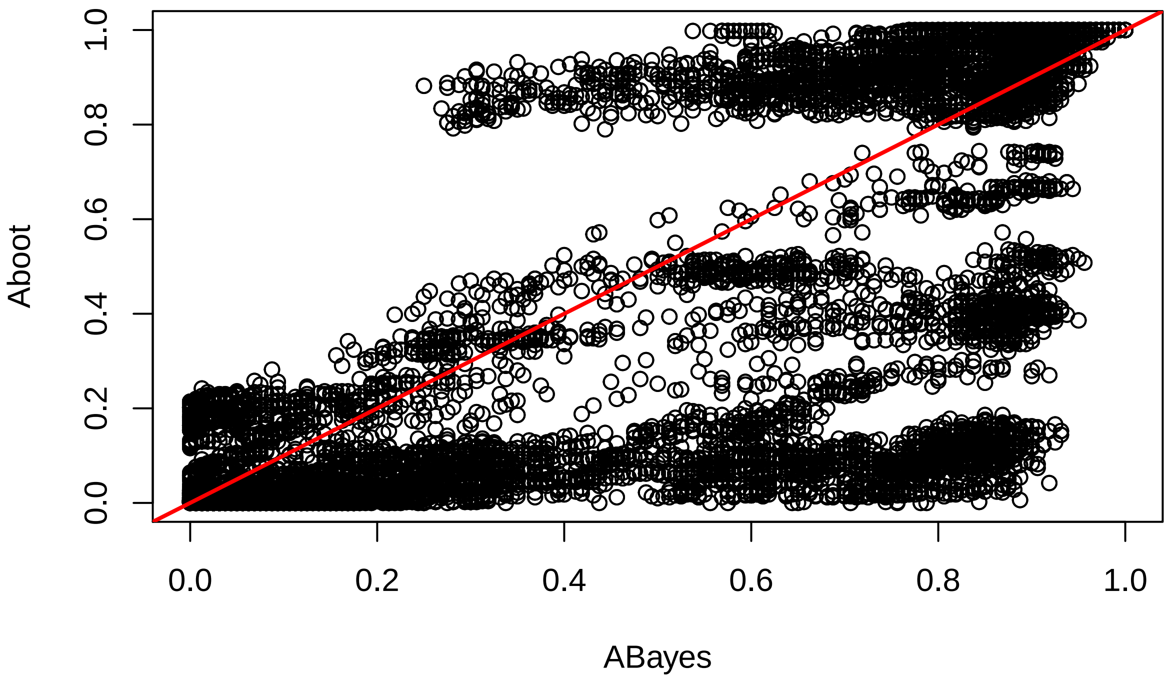
```r
ABayes = res_bayes[[1]]
rand_bayes = res_bayes[[2]]
#p = res_bayes[[3]]
#p_ = apply(p, 2, mean)

#par(mfrow = c(2,2))
image(ABayes, axes=FALSE, main="Bayes")
```



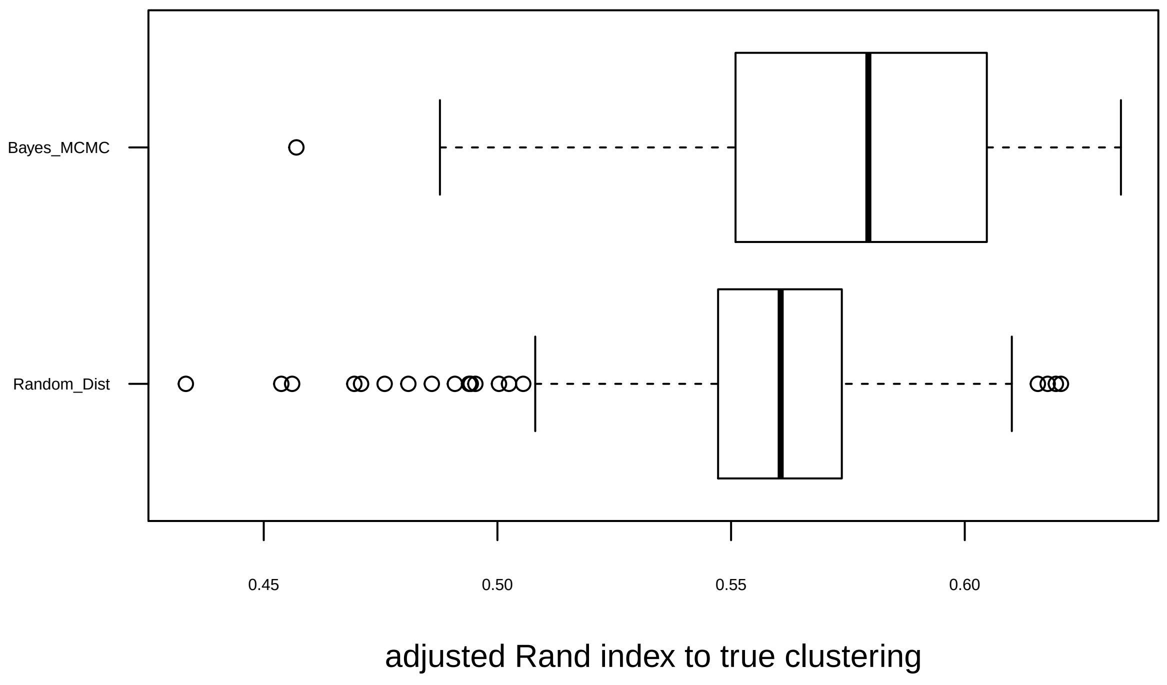
```r
image(Aboot,axes = F, main = "Random_Dist")
```



```r
plot( ABayes, Aboot )
abline(0,1,col="red", lwd=2 )
```



```r
boxplot( rev(list( Bayes_MCMC=rand_bayes, Random_Dist=rand_boot
)), horizontal=TRUE, las=1, cex.axis=.5, cex.label=.8,
xlab="adjusted Rand index to true clustering" )
```



Let us try random sampled y

loose the constraint that posterior(two elements belong to same cluster | given obs) is not exactly the same as optimized result from traditional clustering (hierarchical, k-medoids) that purely based on distance. And it should since likelihood contains more information of data than the euclidean distance.

let us consider the P(i,j being same class | marginal mixture distribution)

```
# boot_mt = matrix(0,n,n)
# boot2_mt = matrix(0,n,n)
# Bayes_mt = matrix(0,n,n)
# for(i in 1:100){
#   #random sample y
#   y <- rnorm(n,mean=mu, sd=sig)
#   res_dist = boot(length(y),5,y,100)
#   res_dist2 = boot4(y,5,100,5)
#   Aboot = res_dist[[1]]
#   Aboot2 = res_dist2[[1]]
#   #rand_boot = res_dist[[2]]
#   res_bayes = DP_bayes(y, mcmc, prior, 5)
#   ABayes = res_bayes[[1]]
#   #rand_bayes = res_bayes[[2]]
#   boot_mt = boot_mt + Aboot
#   Bayes_mt = Bayes_mt + ABayes
#   boot2_mt = boot2_mt + Aboot2
# }
# # par(mfrow = c(3,1))
# # hist(boot_mt / 100)
# # hist(Bayes_mt / 100)
# # hist(boot2_mt/100)
# hist(boot_mt / 100)
# hist(boot2_mt / 100)
# hist(Bayes_mt / 100)
```

try first apply modal cluster and then using random distance matrix