Second assignment

Thibault Doutre, ID : 26980469

STAT 241 : Statistical Learning Theory

Notes: My code has comments incorporated but I do not describe my plotting functions. I will provide the scripts associated with this pdf file, which was compiled thanks to R Sweave.

1 Phylogenetic tree inference

1.1 SumProduct algoritm

Define the graph as a list of neighbors.

```
Neighbors=vector("list",9)
Neighbors[[1]]=c(6)
Neighbors[[2]]=c(6)
Neighbors[[3]]=c(7)
Neighbors[[4]]=c(7)
Neighbors[[5]]=c(9)
Neighbors[[6]]=c(1,2,8)
Neighbors[[7]]=c(3,4,8)
Neighbors[[8]]=c(6,7,9)
Neighbors[[9]]=c(5,8)
Neighbors
```

```
## [[1]]
## [1] 6
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 7
##
## [[4]]
## [1] 7
##
## [[5]]
## [1] 9
##
## [[6]]
```

```
## [1] 1 2 8
##
## [[7]]
## [1] 3 4 8
##
## [[8]]
## [1] 6 7 9
##
## [[9]]
## [1] 5 8
```

Define an arbitrary root (does not change the result) and an evidence set of nodes.

```
evidence=c(1,2,3,4,5)
f=9 #arbitrary root
```

Define a 4 dimensional array which defines the dual clique potentials by assigning either M1 or M2 to connected pairs of nodes. The first two index correspond to dual cliques and the two others are the 4 dimensional matrix which define the potentials of the cliques.

```
def_M = function (m){
  # M1
  M1=matrix(c(m[1],m[4],m[2],m[3],
              m[4],m[1],m[3],m[2],
              m[2],m[3],m[1],m[4],
              m[3],m[2],m[4],m[1]),4)

  # M2
  M2=matrix(c(m[1],3*m[4],4*m[2],3*m[3],
              3*m[4],m[1],3*m[3],4*m[2],
              4*m[2],3*m[3],m[1],3*m[4],
              3*m[3],4*m[2],3*m[4],m[1]),4)
  # Potentials
  M = array(dim=c(9,9,4,4))
  M[1,6,,]=M1
  M[2,6,,]=M1
  M[3,7,,]=M1
  M[4,7,,]=M1
  M[6,8,,]=M1
  M[7,8,,]=M1
  M[8,9,,]=M1
  M[5,9,,]=M2
```

```
  M[6,1,,]=M1
  M[6,2,,]=M1
  M[7,3,,]=M1
  M[7,4,,]=M1
  M[8,6,,]=M1
  M[8,7,,]=M1
  M[9,8,,]=M1
  M[9,5,,]=M2
  return(M)
}
```

Define single potentials.

```
psi=matrix(rep(1,9*4),ncol=9)
psi[,9]=c(8,9,9,8)
```

Now, compute the SumProduct algorithm for trees, as described in the textbook.

```
sum_product = function(Neighbors,psiE,evidence,x,M,f){
  # Define psiE
  psiE=psi
  psiE[,evidence]=x*psi[,evidence]
  # Messages
  mess=array(data=0,dim=c(9,9,4))
  #Marginals
  marginals=vector("list",9)

  # Send message
  send_message=function(j,i){
    # Compute product of mkj
    product_mkj=c(1,1,1,1)
    for (k in Neighbors[[j]]){
      if (k != i){
        product_mkj=product_mkj*mess[k,j,]
      }
    }
    # Compute mji
    mji=c(0,0,0,0)
    possible_xj=diag(4) # sum over all values
    for (j0 in 1:4){
      xj=possible_xj[,j0]
      # Message to be sent
      #mji=mji+ ( M[j,i,,] %*% (psiE[,j]) )* product_mkj*xj
      psiE_xj=psiE[j0,j] # real
```

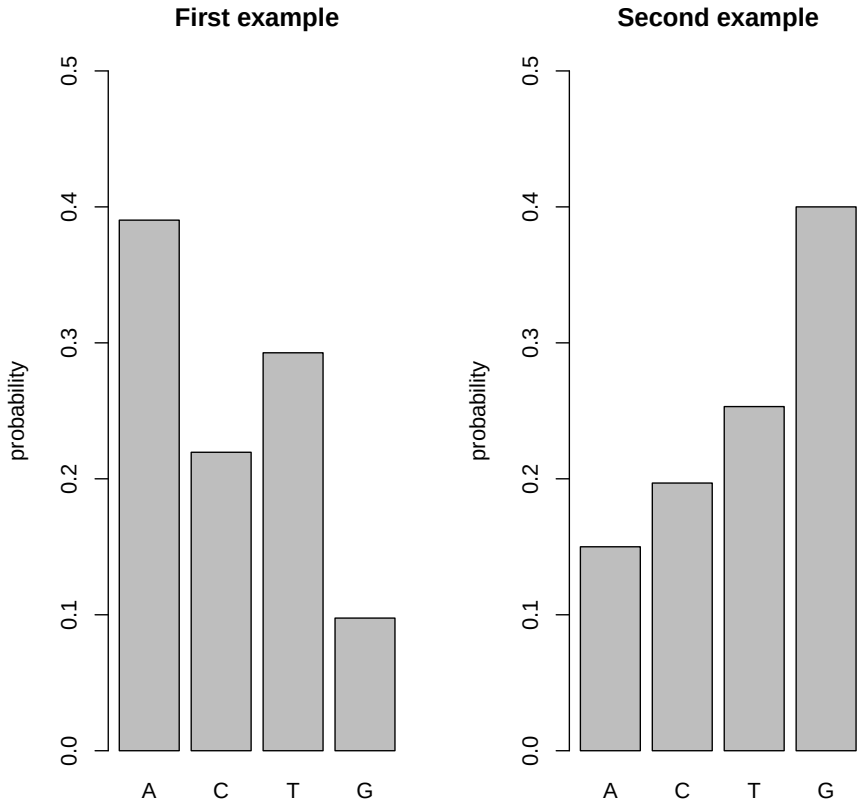
```
    mkj_xj=product_mkj[j0] # real
    psi_xi_xj=M[j,i,,j0] # vector
    mji=mji+ psiE_xj*psi_xi_xj*mkj_xj
  }
  mess[j,i,] <<- mji
}

# Collect
collect = function (i,j){
  for (k in Neighbors[[j]]){
    if ( k !=i ){
      collect(j,k)
    }
  }
  send_message(j,i)
}

# Distribute
distribute = function (i,j){
  send_message(i,j)
  for (k in Neighbors[[j]]){
    if ( k !=i ){
      distribute(j,k)
    }
  }
}

# Marginal
compute_marginal = function(i){
  product_mji=1
  for (j in Neighbors[[i]]){
    product_mji=product_mji*mess[j,i,]
  }
  marginals[[i]] <<- psiE[,i]*product_mji
}

# Algorithm
for (e in Neighbors[[f]]){
  collect(f,e)
}
for (e in Neighbors[[f]]){
  distribute(f,e)
}
for (i in 1:9){
  compute_marginal(i)
```

```
  }
  # Normalize marginals
  return(lapply(marginals,function (x) x/sum(x)))
}
```

1.2 Example

Run the algorithm for both example provided in the problem. Plot the corresponding values of ACTG with the barplot function.

```
par(mfrow=c(1,2))
############
# Exemple 1
############

m=c(8,3,2,1)
M=def_M(m)
x=matrix(c(0,0,0,1,0,1,0,0,1,0,0,0,0,0,1,0,0,0,1,0),ncol=5)
s1=sum_product(Neighbors,psiE,evidence,x,M,f)
barplot(s1[[9]],names.arg =c("A","C","T","G"),ylab="probability",
        ylim = c(0,.5),
        main= "First example")

############
# Exemple 2
############

m=c(7,4,3,2)
M=def_M(m)
x=matrix(c(0,1,0,0,1,0,0,0,0,0,0,1,0,0,1,0,0,1,0,0),ncol=5)
s2=sum_product(Neighbors,psiE,evidence,x,M,f)
barplot(s2[[9]],names.arg =c("A","C","T","G"),ylab="probability",
        ylim = c(0,0.5),
        main= "Second example")
```




```
s1[[9]] # probabilities of node 9 for the first example

## [1] 0.39024390 0.21951220 0.29268293 0.09756098

s2[[9]] # probabilities of node 9 for the second example

## [1] 0.150000 0.196875 0.253125 0.400000
```

1.3 Third question

See scanned paper.

2 LMS Algorithm

2.1 Theta star

Load the data, solve the normal equations, extract eigen values and eigen vectors with basics functions in R.

```
# Load data
setwd('~/Documents/STAT241')
data=read.table('./hw2.data',col.names = c('x1','x2','y'))
X=as.matrix(data[,1:2])
Y=as.matrix(data[,3])

# solve(A,b) solve b = Ax
theta_star = solve(t(X)%*%X,t(X) %*% Y)
row.names(theta_star)=NULL
theta_star

##             [,1]
## [1,] 0.5353682
## [2,] 0.2357154

# Eigen vectors / values
eigen_values = eigen(t(X)%*%X)$values
eigen_vectors = eigen(t(X)%*%X)$vectors
lambda_max=max(eigen_values)
eigen_vectors

##             [,1]        [,2]
## [1,] 0.6264609  -0.7794528
## [2,] 0.7794528   0.6264609

eigen_values

## [1] 15.983740  2.418214
```

2.2 Cost function

Implement a cost function which takes the vector theta into argument. Then, display the optimal cost.

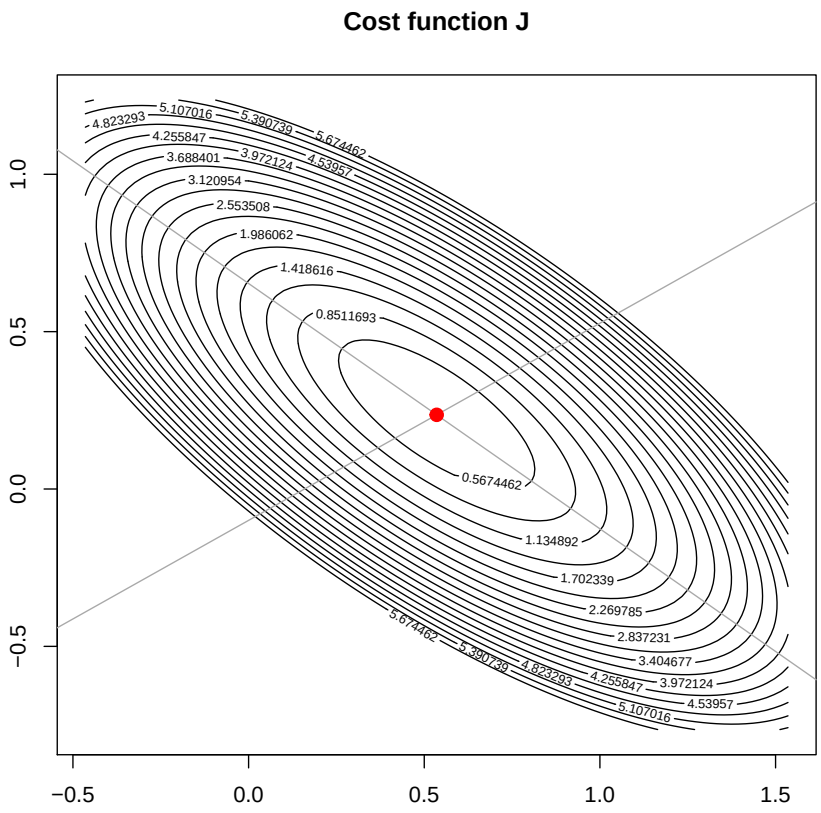
```
# Cost function with vector input theta
J = function (theta){
  a= Y - X %*% theta
  return( (t(a)%*%a) [1,1])
}
J_opt=J(theta_star)
J_opt

## [1] 0.2837231
```

Create the equivalent of the J function with two input arguments which correspond to the composants of theta.

```
# J when arguments are sequences of numbers
J_seq = function(theta1_seq,theta2_seq){
  theta_seq=matrix(c(theta1_seq,theta2_seq),nrow=2,byrow=T)
  return(apply(theta_seq,2,J))
}
```

This function is useful for plotting the cost function J.

```
# Plot
plot_J=function(npoints=100,nlevels=20,xlim=c(theta_star[1]-1,
                                              theta_star[1]+1),
                ylim=c(theta_star[2]-1,theta_star[2]+1),
                title="",col="black",eigen=T){
  x=seq(xlim[1],xlim[2],length.out = npoints)
  y=seq(ylim[1],ylim[2],length.out = npoints)
  z=outer(x,y,function(x,y) J_seq(x,y))
  contour(x,y,z,main= title,levels= seq(J_opt,J_opt*20,
                                        length.out = nlevels),col=col)
  if (eigen)
    abline(-eigen_vectors[1,2]*theta_star[1]+theta_star[2],
           eigen_vectors[1,2],col="darkgray")
    abline(-eigen_vectors[2,2]*theta_star[1]+theta_star[2],
           eigen_vectors[2,2],col="darkgray")
  points(theta_star[1],theta_star[2],col="red",cex=2,pch=20)
}
plot_J(title="Cost function J")
```



2.3 Path

Then, in order to compute the LMS algorithm, create a recursive function which arguments are :

- t = t-th iteration
- rho = step size
- theta0 = initial point
- Y = output value
- X = input value
- init-res = initial construction of the theta sequence of the algorithm. I put all the theta values into this 2 rows matrix. Will be initialize later.

The function returns a list with two arguments :

- theta = last theta value
- res = history of all thetha values calculated

```
theta_t = function(t,rho,theta0,Y,X,init_res){
  if (t==0)
    return(list(theta=theta0,res=init_res))
  else
    # Choose random sample
    i=sample(1:length(Y),1)
    yn=Y[i]
    xn=as.matrix(X[i,],nrow=2)
    # Precedent values obtained by induction
    old=theta_t(t-1,rho,theta0,Y,X,init_res)
    old_theta=old$theta
    old_res=old$res
    # Update theta
    new_theta=old_theta+rho*(yn - (t(old_theta) %*% xn)[1,1]) * xn
    # Update res
    new_res=old_res
    new_res[,t+1]=new_theta
    return(list(theta=new_theta,res=new_res))
}
```

Now, create lms function which initialize the result matrix (res) with the good number of columns, i.e. t+1.

```
lms=function(t,rho,theta0,Y,X){
  # Initialize matrix
  init_res=matrix(nrow=2,ncol=t+1)
  init_res[,1]=theta0
  # Return res argument from theta_t function
  return(theta_t(t,rho,theta0,Y,X,init_res)$res)
}
```

Create a plotting function.

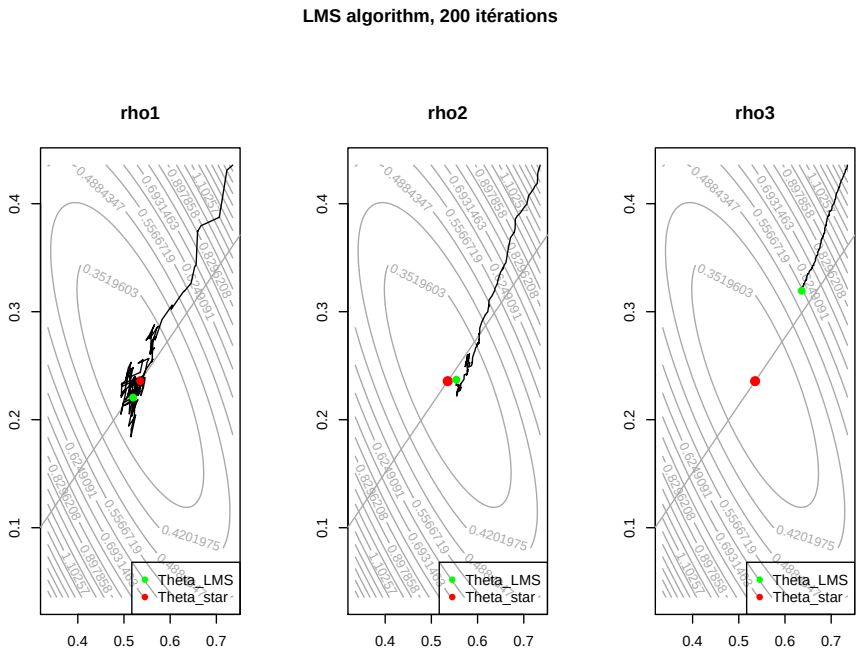
```
lms_plot=function(t,rho,theta0,Y,X,it=1000,title=""){
  plot_J(xlim=theta_star[1]+c(-.2,.2),ylim=theta_star[2]+c(-.2,.2),
         nlevels=80,col="darkgray",eigen=F,title=title)
  l=lms(it,rho,theta0,Y,X)
  lines(l[1,],l[2,],type="l",xlab="x",ylab="y")
  points(theta_star[1],theta_star[2],col="red",pch=19)
  points(l[1,length(l)/2],l[2,length(l)/2],col="green",pch=19)
  legend( "bottomright",legend=c("Theta_LMS","Theta_star"),
          col=c("green","red"),pch=c(16,16) )
}
```

Define parameters.

```
theta0=theta_star+.2
rho1=2/max(eigen_values)
rho2=1/(2*max(eigen_values))
rho3=1/(8*max(eigen_values))
```

Plot for different rho, the LMS algorithm with 200 iterations.

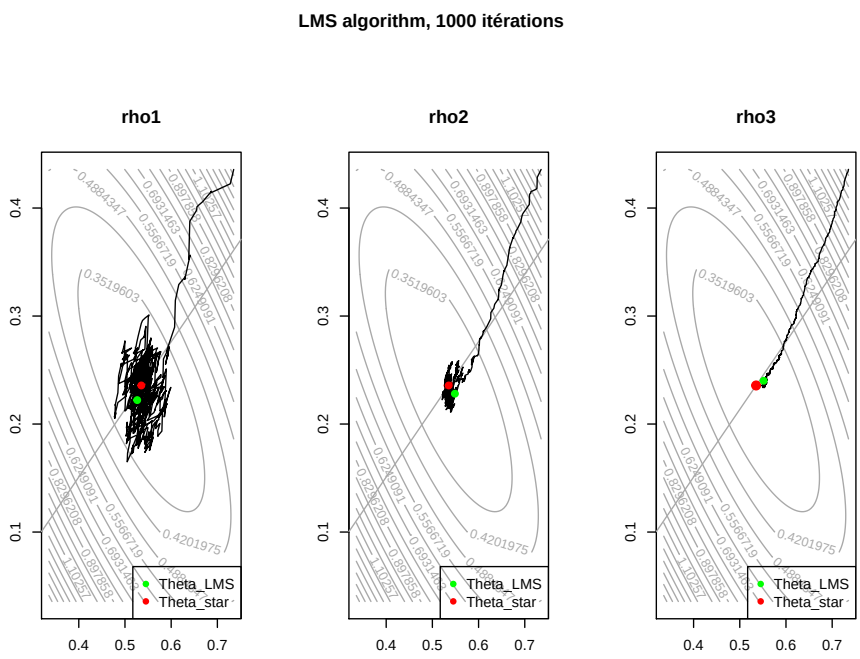
```
set.seed(4)
it=200
par(mfrow=c(1,3),oma = c(10, 0, 7, 0))
lms_plot(t,rho1,theta0,Y,X,it,title="rho1")
lms_plot(t,rho2,theta0,Y,X,it,title="rho2")
lms_plot(t,rho3,theta0,Y,X,it,title="rho3")
title("LMS algorithm, 200 itrations",outer=T,lwd=3,cex=3)
```



We can observe that with rho1, the algorithm does not converge, the step is too large as the theorical boundary for rho is not respected. For rho2, the algorithm

seems to converge but with rho3, the algorithm is to slow: we need to compute more iterations. For example, with 1000 iterations:

```
set.seed(15)
it=1000
par(mfrow=c(1,3),oma = c(10, 0, 7, 0))
lms_plot(t,rho1,theta0,Y,X,it,title="rho1")
lms_plot(t,rho2,theta0,Y,X,it,title="rho2")
lms_plot(t,rho3,theta0,Y,X,it,title="rho3")
title("LMS algorithm, 1000 itrations",outer=T,lwd=3,cex=3)
```



Now, the parameter rho3 seems to be efficient enough to approximate theta star.