CS281A-Problem Set#4

Juanyan Li

10/30/2015

1. K-Means

1) Keeping μ_j fixed for any i

$$\min_{z_i^j} J = \sum_{i=1}^{N} \min_{z_i^j} \sum_{j=1}^{K} z_i^j ||x_i - \mu_j||^2$$

Since $z_i = [z_i^1, \ldots, z_i^j, \ldots, z_i^K]$ with only one element to be 1, others to be zero, minimizing J with fixed μ_j yields that $z_i^j = 1$ only when $||x_i - \mu_j||$ is the smallest among all i. This is the same as:

$$z_i^j := 1[j = \arg\min_k ||x_i - \mu_k||]$$

2)Keeping z_i^j for any j

$$\min_{\mu_j} J = \sum_{j=1}^{K} \min_{\mu_j} \sum_{i=1}^{N} z_i^j ||x_i - \mu_j||^2$$

Taking first derivative of J respect to μ_j, and assigning it to zero:

$$\frac{\partial J}{\partial \mu_j} = -2 \sum_i z_i^j (x_i - \mu_j) = 0$$

$$\mu_j = \frac{\sum_i z_i^j x_i}{\sum_i z_i^j}$$

2. IPF

By implementing IPF algorithm (see *Appendix*) on `hw4data.data`, I got a result like the following:

	Model1	Model2	Model3
$\psi_{3,4}(x_3, x_4)$	$\begin{bmatrix} 11.6498 & 6.9329 \\ 0.9661 & 2.1663 \end{bmatrix}$	$\begin{bmatrix} 1.0350 & 2.0743 \\ 1.2814 & 5.4192 \end{bmatrix}$	$\begin{bmatrix} 1.9353 & 2.0758 \\ 1.9525 & 4.3465 \end{bmatrix}$
$l(\theta\|\mathcal{D})$	-3285.051	-3316.203	-3319.577

Note that initialization of $\psi_{i,j}(x_i, x_j)$ for any pair clique i, j is $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, since this would affect the final result.

Computation of the log likelihood for each of the model shows that the first one yields the largest value.**WHY?**

3. Max Likelihood Tree

(a)

(b)

(c)

(d)

(e) Maximum weight spanning tree

Implementation of Kruskal's algorithm returns a maximum weight spanning tree, the edges are shown below:

Node u	Node v
1	7
5	7
1	4
3	7
2	6
1	2

Appendix

`hw4-IPF.R`: Implementation of IPF algorithm on `hw4data.data`

```
setwd("E:/Berkeley/CS281A/ps4")
source("IPF_computation.R")


##Read data as data.frame
data <- read.table("hw4data.data")
dim(data) # 500 * 7

for(s in 1:3){
  ##Initialize
  potential <- init(s)
  nVal <- 4

  ##Compute counts for each clique(empirical dist on one node)
  count <- lapply(potential,pcount)

  ##IPF
  iter <- 40 # number of iterations
  #scene <- as.matrix(expand.grid(rep(list(c(1:nVal)),length(potential)-1)))
  #map <- matrix(0,ncol=dim(scene)[2],nrow=dim(scene)[1])
  n <- length(potential)
  uni <- unique(data)

  for(i in 1:iter){
    #Get all the marginal probabilities
    #Question: should we use the unique value to compute this?
    f <- rep(0,4)
    for(j in 1:n){
      phi_prod <- mapping(uni)
      clique <- potential[[j]][[1]] # current clique/pair index
      ind <- 1 + uni[,clique[1]] + 2*uni[,clique[2]]
      for(k in 1:nVal){
        f[k] = sum(phi_prod[which(ind==k)])
      }
      potential[[j]][[2]] = potential[[j]][[2]]*count[[j]]*(sum(f)/f)
    }
  }

  #Print potential for clique {3,4}
  for (i in 1 : n){
    if(potential[[i]][[1]][1] == 3 & potential[[i]][[1]][2] == 4){
      print(potential[[i]][[2]])
    }
  }

  #Compute the log likelihood
  prod <- mapping(data)
  pdist <- sum(log(prod/sum(prod)))

  print(pdist)
}
```

`IPF_computation.R`: Functions used in IPF algorithms

```
#Initialization
init <- function(model=1){ # Graph models selection, default to be model 1
  nVal <- 4 # number of possible values in each clique potential
  phi <- c(1:nVal) # Initial assignment for clique(pair) potential
  if(model%%1==0) { # Check input argument
    if(model == 1){
      pairlist <- list(c(1,5),c(1,6),c(2,5),c(3,4),c(3,6),c(4,5),c(4,7),c(6,7)) # 8 pairs
      potential <- lapply(pairlist,list,phi)
    }
    else if(model==2){
      pairlist <- list(c(1,2),c(1,3),c(1,4),c(1,7),c(2,4),c(2,6),c(3,4),
                       c(3,5),c(3,7),c(4,6),c(5,7))  # 11 pairs
      potential <- lapply(pairlist,list,phi)
    }
    else if(model==3){
      pairlist <- list(c(1,2),c(1,3),c(1,4),c(1,5),c(1,6),c(1,7),c(2,3),
                       c(2,4),c(2,5),c(2,6),c(2,7),c(3,4),c(3,5),c(3,6),
                       c(3,7),c(4,5),c(4,6),c(4,7),c(5,6),c(5,7),c(6,7))  # 21 pairs
      potential <- lapply(pairlist,list,phi)
    }
    else{
      stop("Only 1,2 and 3 are accepted!")
    }
    return(potential)
  }
  else{
    stop("Invalid input!")
  }
}

#Computation of counts
pcount <- function(p){
  nObs <- dim(data)[1]
  count <- as.data.frame(table(data[p[[1]]]))[,3]/nObs
  return(count)
}

#Computation of marginal probability distribution
mapping<- function(dat){
  map <- matrix(0,ncol=n,nrow=dim(dat)[1])
  for(i in 1 : n){
    ind <- as.matrix(dat[,potential[[i]][[1]][1]] + 2*dat[,potential[[i]][[1]][2]]+1)
    map[,i] = potential[[i]][[2]][ind]
  }
  phi_prod <- apply(map,1,prod)
  return(phi_prod)
}
```

`hw4-mwst.R`: Implementation of Kruskal's algorithm to find maximum spanning tree based on `hw4data.data`

```r
setwd("E:/Berkeley/CS281A/ps4")
source("kruskal.R")

##Read data as data.frame
data <- read.table("hw4data.data")
dim(data) # 500 * 7
nObs <- dim(data)[1]
nNodes <- dim(data)[2]
nPairs <- choose(nNodes,2)
pairlist <- list(c(1,2),c(1,3),c(1,4),c(1,5),c(1,6),c(1,7),c(2,3),
                 c(2,4),c(2,5),c(2,6),c(2,7),c(3,4),c(3,5),c(3,6),
                 c(3,7),c(4,5),c(4,6),c(4,7),c(5,6),c(5,7),c(6,7))  # 21 pairs

##Compute empirical marginals on nodes and pairs
pairCount <- lapply(pairlist,function(vec){
                                as.data.frame(table(data[vec]))[,3]/nObs
                             })
nodeCount <- lapply(c(1:nNodes),function(val){
                                as.data.frame(table(data[,val]))[,2]/nObs
                             })

##Compute the weight for each edge
weight <- rep(0,nPairs)
for(i in 1 : nPairs){
  first <- pairlist[[i]][1]
  second <- pairlist[[i]][2]
  denominator <- c(nodeCount[[first]][1]*nodeCount[[second]][1],
                   nodeCount[[first]][2]*nodeCount[[second]][1],
                   nodeCount[[first]][1]*nodeCount[[second]][2],
                   nodeCount[[first]][2]*nodeCount[[second]][2])
  weight[i] = sum(pairCount[[i]]*log(pairCount[[i]]/denominator))
}

df <- data.frame(cbind(t(matrix(unlist(pairlist),nrow=2)),weight))
order <- order(df$weight,decreasing=TRUE)

##Find the maximum spanning tree
s <- init(nNodes)
parent <- s[[1]]
rank <- s[[2]]

msTree <- list()

for (i in 1 : nPairs){
  u <- df[order[i],1]
  v <- df[order[i],2]
  if(find(u) != find(v)){
    msTree[[i]] <- c(u,v)
    union(u,v)
  }
}
```

```
msTree <- t(matrix(unlist(msTree),nrow=2))
```

`kruskal.R`: Source code for Kruskal’s algorithm

```
#Kruskal's algorithm

init <- function(n){
  return(list(c(1:n),rep(0,n)))
}

find <- function(u){
  while(.GlobalEnv$parent[u] != u)
    u = .GlobalEnv$parent[u]
  return(u)
}

union <- function(u,v){
  ru <- find(u)
  rv <- find(v)
  if(ru == rv) {
    return
  }
  if(.GlobalEnv$rank[ru] > .GlobalEnv$rank[rv]){
    .GlobalEnv$parent[rv] = ru
  }
  else{
    .GlobalEnv$parent[ru] = rv
    if(.GlobalEnv$rank[ru] == .GlobalEnv$rank[rv]){
    }
  }
}
```