STAT542 Coding3 Bonus

Chenzi Zhang, NetID chenziz2, UIN 654728837

3/27/2019

Contents

Generate Samples from an HMM

```
T = 200
A0 = rbind(c(0.8, 0.2),
           c(0.2, 0.8))
B0 = rbind(c(0.1, 0.2, 0.7),
           c(0.4, 0.3, 0.3))
w0 = c(0.5, 0.5)
para0 = list(mz = 2, mx = 3, w = w0,
             A = A0, B = B0)
genHMM = function(para, n){
  # return n samples from HMM with parameter = para
  z = para$mz
  mx = para$mx
  w = para$w
  A = para$A
  B = para$B
  Z = rep(0, n)
  X = rep(0, n)

  ## MY CODE: generate Z[1]
  Z[1] = sample(1:2, size = 1, prob = w, replace = TRUE)
  ## MY CODE: generate Z[i]
  for(i in 2:n){
    Z[i] = sample(1:2, size = 1, prob = A[Z[i-1],], replace = TRUE)
  }
  ## MY CODE: generate X[i]
  for(i in 1:n){
    X[i] = sample(1:3, size = 1, prob = B[Z[i],], replace = TRUE)
  }
  return(X)
}
```

```
data = genHMM(para0, T)
```

The Baum-Welch (i.e., EM) Algorihtm

Define forward and backward functions

```
forward.prob = function(x, para){
  # Output the forward probability matrix alp
  # alp: T by mz, (t, i) entry = P(x_{1:t}, Z_t = i)
  T = length(x)
  mz = para$mz
  A = para$A
  B = para$B
  w = para$w
  alp = matrix(0, T, mz)

  # fill in the first row of alp
  alp[1, ] = w * B[, x[1]]
  # Recursively compute the remaining rows of alp
  for(t in 2:T){
    tmp = alp[t-1, ] %*% A
    alp[t, ] = tmp * B[, x[t]]
  }
  return(alp)
}

backward.prob = function(x, para){
  # Output the backward probability matrix beta
  # beta: T by mz, (t, i) entry = P(x_{1:t}, Z_t = i)
  T = length(x)
  mz = para$mz
  A = para$A
  B = para$B
  w = para$w
  beta = matrix(1, T, mz)

  # The last row of beta is all 1.
  # Recursively compute the previous rows of beta
  for(t in (T-1):1){
    tmp = as.matrix(beta[t+1, ] * B[, x[t+1]])  # make tmp a column vector
    beta[t, ] = t(A %*% tmp)
  }
  return(beta)
}
```

Define BW.onestep function

```
BW.onestep = function(x, para){
  # Input:
```

```r
  # x: T-by-1 observation sequence
  # para: mx, mz, and current para values for
  #    A: initial estimate for mz-by-mz transition matrix
  #    B: initial estimate for mz-by-mx emission matrix
  #    w: initial estimate for mz-by-1 initial distribution over Z_1
  # Output the updated parameters after one iteration
  # We DO NOT update the initial distribution w

  T = length(x)
  mz = para$mz
  mx = para$mx
  A = para$A
  B = para$B
  w = para$w
  alp = forward.prob(x, para)
  beta = backward.prob(x, para)

  myGamma = array(0, dim=c(mz, mz, T-1))
  ## MY CODE:
  ## Compute gamma_t(i,j) P(Z[t] = i, Z[t+1]=j),
  ## for t=1:T-1, i=1:mz, j=1:mz,
  ## which are stored an array, myGamma
  for(t in 1:(T-1)){
    for(i in 1:mz){
      for(j in 1:mz){
        myGamma[i,j,t] = alp[t,i]*A[i,j]*B[j,x[(t+1)]]*beta[(t+1),j]
      }
    }
  }


  # M-step for parameter A
  A = rowSums(myGamma, dims = 2)
  A = A/rowSums(A)
  # M-step for parameter B
  tmp = apply(myGamma, c(1, 3), sum)  # mz-by-(T-1)
  tmp = cbind(tmp, colSums(myGamma[, , T-1]))
  for(l in 1:mx){
    B[, l] = rowSums(tmp[, which(x==l)])
  }
  B = B/rowSums(B)

  para$A = A
  para$B = B
  return(para)
}
```

Define myBM function

```r
myBW = function(x, para, n.iter = 100){
  # Input:
  # x: T-by-1 observation sequence
```

```
  # para: initial parameter value
  # Output updated para value (A and B; we do not update w)

  for(i in 1:n.iter){
    para = BW.onestep(x, para)
  }
  return(para)
}
```

The Viterbi Algorihtm

```
myViterbi = function(x, para){
  # Output: most likely sequence of Z (T-by-1)

  T = length(x)
  mz = para$mz
  A = para$A
  B = para$B
  w = para$w
  log.A = log(A)
  log.w = log(w)
  log.B = log(B)

  # Compute delta (in log-scale)
  delta = matrix(0, T, mz)
  # fill in the first row of delta
  delta[1, ] = log.w + log.B[, x[1]]

  ## MY CODE:
  ## Recursively compute the remaining rows of delta
  for(t in 1:(T-1)){
    tmp = NULL
    for(j in 1:mz){
      tmp = rbind(tmp, delta[t,j] + log.A[j,])
    }
    log.max = apply(tmp, 2, function(x) max(x))
    delta[(t+1), ] = log.max + log.B[ ,x[(t+1)]]
  }

  # Compute most prob sequence Z
  Z = rep(0, T)
  # start with the last entry of Z
  Z[T] = which.max(delta[T, ])

  ## MY CODE:
  ## Recursively compute the remaining entries of Z
  for(t in T:2){
    Z[t-1] = which.max(delta[(t-1), ] + log.A[, Z[t]])
  }

  return(Z)
```

```
}
```

Test My Algorithm

```
data = genHMM(para0, T)
mz = 2
mx = 3
ini.w = rep(1, mz); ini.w = ini.w / sum(ini.w)
ini.A = matrix(1, 2, 2); ini.A = ini.A / rowSums(ini.A)
ini.B = matrix(1:6, 2, 3); ini.B = ini.B / rowSums(ini.B)
ini.para = list(mz = 2, mx = 3, w = ini.w,
                A = ini.A, B = ini.B)

myout = myBW(data, ini.para, n.iter = 100)
myout.Z = myViterbi(data, myout)
myout.Z[myout.Z==1] = 'A'
myout.Z[myout.Z==2] = 'B'
```

```
library(HMM)
hmm0 =initHMM(c("A", "B"), c(1, 2, 3),
              startProbs = ini.w,
              transProbs = ini.A,
              emissionProbs = ini.B)
Rout = baumWelch(hmm0, data, maxIterations=100, delta=1E-9, pseudoCount=0)
Rout.Z = viterbi(Rout$hmm, data)
```

```
myout$A
```

```
##           [,1]      [,2]
## [1,] 0.4526640 0.5473360
## [2,] 0.4564762 0.5435238
```

```
Rout$hmm$transProbs
```

```
##     to
## from         A         B
##    A 0.4526640 0.5473360
##    B 0.4564762 0.5435238
```

They're the same.

```
myout$B
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.2454993 0.2414352 0.5130655
## [2,] 0.3639114 0.2388004 0.3972881
```

```
Rout$hmm$emissionProbs
```

```
##       symbols
## states         1         2         3
##      A 0.2454993 0.2414352 0.5130655
##      B 0.3639114 0.2388004 0.3972881
```

They're the same.

```r
cbind(Rout.Z, myout.Z)[c(1:10, 180:200), ]
```

```
##       Rout.Z myout.Z
##  [1,] "A"    "A"
##  [2,] "B"    "B"
##  [3,] "A"    "A"
##  [4,] "B"    "B"
##  [5,] "A"    "A"
##  [6,] "A"    "A"
##  [7,] "B"    "B"
##  [8,] "B"    "B"
##  [9,] "B"    "B"
## [10,] "A"    "A"
## [11,] "A"    "A"
## [12,] "B"    "B"
## [13,] "B"    "B"
## [14,] "B"    "B"
## [15,] "A"    "A"
## [16,] "A"    "A"
## [17,] "B"    "B"
## [18,] "A"    "A"
## [19,] "B"    "B"
## [20,] "B"    "B"
## [21,] "A"    "A"
## [22,] "B"    "B"
## [23,] "B"    "B"
## [24,] "B"    "B"
## [25,] "A"    "A"
## [26,] "B"    "B"
## [27,] "B"    "B"
## [28,] "A"    "A"
## [29,] "B"    "B"
## [30,] "B"    "B"
## [31,] "A"    "A"
```

```r
sum(Rout.Z != myout.Z)
```

```
## [1] 0
```

The output from my Viterbi algorithm and the one from HMM are exactly the same.