SC1 Assessment 2

Sam Bowyer

2023-01-06

Hidden Markov Models

Introduction

In this document we will discuss and implement hidden Markov models (HMMs). These are an interesting type of statistical model that will allow us to utilise a variety of techniques discussed in this module, for example we will see how we can improve efficiency by using sparse matrices and ultimately we will implement the Baum-Welch algorithm which is a special case of the expectation maximisation (EM) algorithm which optimises the MAP estimates of an HMM's parameters.

Markov Chains

Recall that a sequence of random variables $X_1, X_2, ...$ taking values in a state space S is a Markov chain if it satisfies the Markov property $\forall t$:

$$\mathbb{P}(X_t = x_t | X_1 = x_1, ..., X_{t-1} = x_{t-1}) = \mathbb{P}(X_t = x_t | X_{t-1} = x_{t-1}).$$

That is, the value of X_t depends **only** on the value of X_{t-1}.

In this document we will limit ourselves to time homogeneous Markov chains with finite state spaces which we will set to $S = \{1, 2, ..., N\} = [N]$ without loss of generality. For a Markov chain to be time homogeneous we simply mean that the probability of transition between any two states is constant with respect to t, that is:

$$\mathbb{P}(X_t = j | X_{t-1} = i) = \mathbb{P}(X_{t'} = j | X_{t'-1} = i) \ \ \forall t, t'.$$

With this, we can then represent our Markov chain fully by (π, A) where $\pi \in \mathbb{R}^N$ gives us the initial distribution over state space S and $A \in \mathbb{R}^{N \times N}$ gives us the transition probabilities, meaning that

$$\pi_i = \mathbb{P}(X_1 = i)$$

and

$$a_{ij} = \mathbb{P}(X_t = j | X_{t-1} = i).$$

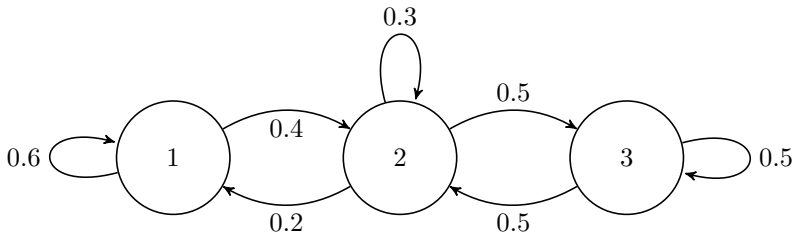
Below we provide an implementation of Markov chains in `R`.

```
runMC <- function(pi, A, max_t){
  Xs = rep(0, max_t)
  S  = 1:length(pi)

  Xs[1] = sample(S, 1, prob = pi)

  for(i in 2:max_t){
    Xs[i] = sample(S, 1, prob=A[Xs[i-1],])
  }
```

```
  return(Xs)
}

# Simple bounded walk w/ N=3 example
pi = rep(1/3, 3)
A  = matrix(c(0.6, 0.4, 0,
              0.2, 0.3, 0.5,
                0, 0.5, 0.5), nrow=3, byrow=T)
runMC(pi, A, 38)
```

```
##  [1] 3 2 1 1 1 1 1 2 3 2 2 3 2 2 3 3 2 3 2 2 3 3 2 2 3 2 3 2 2 3 2 2 3 2 2 2 3 2
```

Note in particular that we never get a 1-to-3 or 3-to-1 transition since we've set $a_{1,3} = a_{3,1} = 0$. This Markov chain can be represented by the following diagram.



Sparse Matrices For Large Random Walks

If we extend this random walk model to one with a large N number of states where transition is only possible between adjacent states (and self-loops are not allowed), we arrive at a situation where we might benefit from using R's sparse matrix implementation.

```
library(Matrix)
N = 21 # SET TO AT LEAST 201 BEFORE FINAL KNIT
pi = rep(0,N)
pi[ceiling(N/2)] = 1

getRandomWalkTransitions <- function(N, sparse=FALSE){
  a_ijs = rep(0, N*N)
  a_ijs[2] = a_ijs[N*N - 1] = 1
  for (i in 2:(N-1)){
    a_ijs[N*(i-1) + i - 1] = a_ijs[N*(i-1) + i + 1] = 0.5
  }

  if(sparse){
    return(Matrix(a_ijs, nrow=N, byrow=T, sparse=T))
  } else {
    return(matrix(a_ijs, nrow=N, byrow=T))
  }
}

A = getRandomWalkTransitions(N)
A_sparse = getRandomWalkTransitions(N, TRUE)
```

As an example, here is A for the simpler case when $N = 10$:

```
getRandomWalkTransitions(10)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0   0.0
##  [2,]  0.5  0.0  0.5  0.0  0.0  0.0  0.0  0.0  0.0   0.0
##  [3,]  0.0  0.5  0.0  0.5  0.0  0.0  0.0  0.0  0.0   0.0
##  [4,]  0.0  0.0  0.5  0.0  0.5  0.0  0.0  0.0  0.0   0.0
##  [5,]  0.0  0.0  0.0  0.5  0.0  0.5  0.0  0.0  0.0   0.0
##  [6,]  0.0  0.0  0.0  0.0  0.5  0.0  0.5  0.0  0.0   0.0
##  [7,]  0.0  0.0  0.0  0.0  0.0  0.5  0.0  0.5  0.0   0.0
##  [8,]  0.0  0.0  0.0  0.0  0.0  0.0  0.5  0.0  0.5   0.0
##  [9,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.5  0.0   0.5
## [10,]  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0   0.0
```

When working with Markov chains it can often be useful to find the $\mathbb{P}(X_t = i|\pi, A)$ for all $i \in S$ and some arbitrary t. We can calculate a vector of these probabilities for each state $i \in S$ easily as πA^{t-1} (with π as a row vector), but note below that such a calculation is made considerably faster for large N when using the sparse matrix implementation compared to the regular implementation:

```
stateProbs <- function(pi, A, t){
  temp = A
  for (i in 1:t-2){
    temp = temp %*% A
  }
  return(pi %*% temp)
}

largeN = 200

largeN_pi = rep(0, largeN)
largeN_pi[ceiling(largeN/2)] = 1

largeN_A = getRandomWalkTransitions(largeN)
largeN_A_sparse= getRandomWalkTransitions(largeN, TRUE)

t = 1000

system.time(stateProbs(largeN_pi, largeN_A, t))
```

```
##    user  system elapsed
##   3.189   0.004   3.231
```

```
system.time(stateProbs(largeN_pi, largeN_A_sparse, t))
```

```
##    user  system elapsed
##   0.239   0.004   0.242
```

Also note that both of these do indeed produce the same result:

```
all(stateProbs(pi, A, t) == stateProbs(pi, A_sparse, t))
```

```
## [1] TRUE
```

We can also actually optimize this even further by performing $t - 1$ vector-matrix multiplications rather than the more expensive matrix-matrix multiplications:

```
stateProbs <- function(pi, A, t){
  temp = pi
  for (i in 1:t-1){
    temp = temp %*% A
```

```
  }
  return(temp)
}

system.time(stateProbs(largeN_pi, largeN_A, t))
```

```
##    user  system elapsed
##   0.024   0.000   0.023
```

```
system.time(stateProbs(largeN_pi, largeN_A_sparse, t))
```

```
##    user  system elapsed
##   0.011   0.000   0.011
```

Hidden Markov Models

A hidden Markov model consists of an unobservable (latent) Markov chain $X_1, X_2, ...$ and a sequence of corresponding observations $Y_1, Y_2, ...$ where Y_t depends only on X_t:

$$\mathbb{P}(Y_t = y_t | X_1 = x_1, ..., X_t = x_t, Y_1 = y_1, ..., Y_{t-1} = y_{t-1}, Y_{t+1} = y_{t+1}, ...) = \mathbb{P}(Y_t = y_t | X_t = x_t).$$

We'll be working with the case when each observation Y_t comes from a set of M possible observations $O = \{o_1, ..., o_M\}$ with emission probabilities:

$$b_j(o_k) = \mathbb{P}(Y_t = o_k | X_t = j)$$

for $k \in [M], j \in [N]$. Restricting O to be finite will somewhat simplify the implementation of the Baum-Welch algorithm at the end of this document, however, most of the discussion from this point on would require very few changes in order to apply to HMMs with infinite (whether countable or uncountable) observation sets O.

By writing the emission probabilities as a matrix $B \in \mathbb{R}^{N \times M}$ with entries $b_{ij} = b_i(o_j)$, we can them fully parameterise a given HMM as $\lambda = (\pi, A, B)$ along with an ordered representation of $O = \{o_1, ..., o_k\}$.

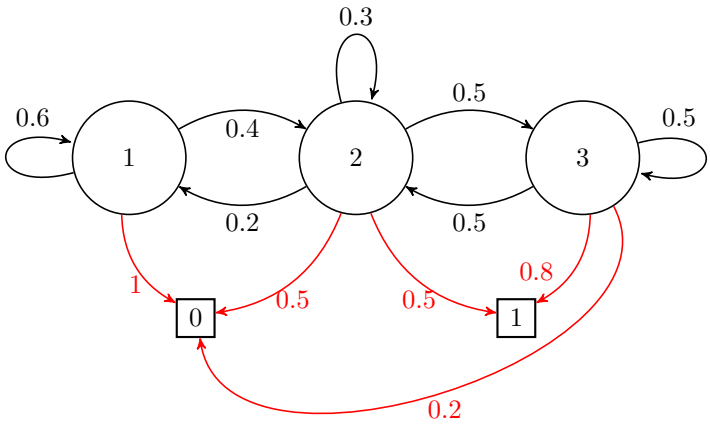
```
runHMM <- function(pi, A, B, O, max_t){
  Xs = runMC(pi, A, max_t)
  Ys = rep(0, max_t)

  for (i in 1:max_t){
    Ys[i] = sample(O, 1, prob=B[Xs[i],])
  }

  return(list(Xs=Xs, Ys=Ys))
}

# Simple bounded walk w/ N=3, M=2 example
pi = rep(1/3, 3)
A  = matrix(c(0.6, 0.4, 0,
              0.2, 0.3, 0.5,
                0, 0.5, 0.5), nrow=3, byrow=T)
B = matrix(c(  1,   0,
             0.5, 0.5,
             0.2, 0.8), nrow=3, byrow=T)
O = c(0,1)

HMM = runHMM(pi, A, B, O, 38)
HMM$Xs
```

```
##  [1] 2 3 3 2 3 3 2 2 1 1 2 3 3 3 2 3 2 3 2 1 1 2 3 3 2 3 2 3 2 3 3 3 3 3 3 2 2 2
```

```
HMM$Ys
```

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

This simple example is represented by the following diagram, where we have taken our previous Markov chain diagram and added on the emission probabilities in red (the choice of $O = \{0, 1\}$ is arbitrary—each o_k could be whatever object we want).



Now we return to our large N random walk and will say that $X_t = i$ leads to an observation of $Y_t = i$ with probability 0.5, and to an observation of $i - 1$ or $i + 1$ each with probability 0.25, hence $M = N + 2$ and $O = \{0, 1, ..., N + 1\}$. (This particular choice of observation procedure does give some special behaviour in that observing $Y_t = 0$ guarantees that $X_t = 1$ and $Y_t = N + 1$ guarantees $X_T = N$, however, this observation is not directly necessary for the general analysis that follows.)

```
getRandomWalkEmmisions <- function(N, sparse=FALSE){
  # One complication here is that the first column represents the probabilities
  # of observing 0, the second of observing 1 and so on.
  b_ijs = rep(0, N*(N+2))
  for (i in 1:N){
    b_ijs[(N+2)*(i-1) + i + 1] = 0.5
    b_ijs[(N+2)*(i-1) + i] = b_ijs[(N+2)*(i-1) + i + 2] = 0.25
  }

  if(sparse){
    return(Matrix(b_ijs, nrow=N, byrow=T, sparse=T))
  } else {
    return(matrix(b_ijs, nrow=N, byrow=T))
  }
}
getRandomWalkEmmisions(10, TRUE)
```

```
## 10 x 12 sparse Matrix of class "dgCMatrix"
##
##  [1,] 0.25 0.50 0.25 .    .    .    .    .    .    .    .    .
##  [2,] .    0.25 0.50 0.25 .    .    .    .    .    .    .    .
##  [3,] .    .    0.25 0.50 0.25 .    .    .    .    .    .    .
##  [4,] .    .    .    0.25 0.50 0.25 .    .    .    .    .    .
##  [5,] .    .    .    .    0.25 0.50 0.25 .    .    .    .    .
```

```
##  [6,] .    .    .    .    .    0.25 0.50 0.25 .    .    .    .
##  [7,] .    .    .    .    .    .    0.25 0.50 0.25 .    .    .
##  [8,] .    .    .    .    .    .    .    0.25 0.50 0.25 .    .
##  [9,] .    .    .    .    .    .    .    .    0.25 0.50 0.25 .
## [10,] .    .    .    .    .    .    .    .    .    0.25 0.50 0.25
```

Note that we've included the option for sparse matrix implementation of B, but since we won't be performing matrix operations with B (it is much more common to do so with A) the benefit is from a storage point of view, not in terms of time efficiency.

```
c(object.size(getRandomWalkEmmisions(200, FALSE)),
  object.size(getRandomWalkEmmisions(200, TRUE)))
```

```
## [1] 323416   9512
```

We will now run $t = 300$ time steps of the large-N random walk and use the resulting data in the proceeding discussion of filtering, smoothing and parameter estimation.

```
N = 25
M = N+2
max_t = 300

pi = rep(1/N, N)
A  = getRandomWalkTransitions(N, TRUE)
B  = getRandomWalkEmmisions(N, TRUE)
O = 0:(N+1)

HMM = runHMM(pi, A, B, O, max_t)
X = HMM$Xs
Y = HMM$Ys
X[1:20]
```

```
##  [1] 15 14 15 16 17 18 17 18 19 18 19 20 19 18 19 20 21 20 21 22
```

```
Y[1:20]
```

```
##  [1] 14 14 14 15 16 17 17 18 19 18 18 21 18 18 20 20 22 19 21 21
```

Filtering

Now suppose that we just have the observations $Y = (Y_1, ..., Y_T)$ and the parameters $\lambda = (\pi, A, B)$, and want to infer the states of the latent Markov chain $X = (X_1, ..., X_T)$. The task of predicting X_t for some $t \in [T]$ based purely on the observations up to time t (that is $Y_1, ..., Y_t$) is known as *filtering*, and we shall tackle it using the *forward algorithm*. This involves calculating what are known as *forward probabilities* $\alpha_t(i) = \mathbb{P}(Y_1, .., Y_t, X_t = i|\lambda)$. This can be done inductively:

$$\alpha_1(i) = \pi_i b_i(Y_1)$$

$$\alpha_{t+1}(j) = \underbrace{\left[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\right]}_{\substack{\text{All of the ways to get to state } j \\ \text{from any state } i \text{ at time step } t}} \cdot b_j(Y_{t+1})$$

With these values we can then calculate

$$\mathbb{P}(X_t|Y_1, ..., Y_t, \lambda) = \frac{\mathbb{P}(X_t, Y_1, ..., Y_t|\lambda)}{\mathbb{P}(Y_1, ..., Y_t|\lambda)} = \frac{\alpha_t(X_t)}{\sum_{i=1}^{N} \alpha_t(i)}.$$

From which *maximum a posteriori* (MAP) estimates of X_t can be taken:

$$X_t^{\text{MAP}} = \underset{i}{\text{argmax}} \frac{\alpha_t(i)}{\sum_{j=1}^{N} \alpha_T(j)} = \underset{i}{\text{argmax}}\, \alpha_t(i)$$

Implementing this in `R` is fairly straightforward, however, it is better to store the logs of the entries of *alpha* since they get very small very quickly.

```
forwardProbabilities <- function(pi, A, B, Y){
  N = length(pi)
  max_t = length(Y)

  # cols index time steps, t
  # rows index states, i  (these are arbitrary choices really)
  log_alpha = matrix(rep(0, N*max_t), nrow=N, ncol=max_t)

  log_alpha[,1] = log(pi*B[,Y[1]+1]) # note we add 1 to observation Y to get the
                                     # desired index within B

  for (t in 1:(max_t-1)){
    for (j in 1:N){
      log_alpha[j, t+1] = as.vector(log(sum(exp(log_alpha[,t]+log(A[,j])))
                                    + log(B[j, Y[t+1]+1]))
    }
  }
  return(log_alpha)
}

log_alpha = forwardProbabilities(pi, A, B, Y)

X_filterEstimate = apply(log_alpha, 2, which.max)
X_filterEstimate; X
```

```
##   [1] 14 14 14 15 16 17 16 17 18 18 18 20 19 18 19 20 21 20 21 20 23 24 25 24 23
##  [26] 22 21 22 21 20 19 18 21 20 19 20 19 20 21 22 21 22 23 22 21 20 19 20 19 20
##  [51] 19 20 19 20 19 20 19 20 19 18 17 16 15 16 17 16 15 16 17 16 17 18 19 20 19
##  [76] 18 19 18 19 18 19 18 19 18 17 16 17 16 15 16 17 16 15 14 17 16 15 14 13 12
## [101] 11 12 13 14 13 14 15 14 13 14 15 14 13 14 15 14 13 12 13 12 15 14 15 14 17
## [126] 18 17 18 19 20 19 20 19 20 21 22 23 22 21 22 23 22 23 24 25 24 25 24 25 24
## [151] 23 22 21 22 23 22 23 24 23 22 21 20 19 20 19 18 19 20 21 20 19 20 19 20 19
## [176] 20 19 18 21 20 19 18 17 18 19 18 17 18 17 16 15 14 17 18 19 18 17 16 17 16
## [201] 17 18 17 18 17 16 19 18 21 20 19 18 19 18 19 18 19 16 15 14 13 14 15 14 15
## [226] 16 15 14 15 14 13 12 11 10  9 10 11 12 13 14 15 14 15 16 17 18 17 16 17 18
## [251] 17 18 17 16 19 20 19 20 19 20 19 18 17 16 15 14 15 12 11 10  9 10 11 12 13
## [276] 14 13 14 15 16 17 16 15 14 15 14 17 16 17 16 17 16 17 16 17 18 19 18 19 18
```

```
##   [1] 15 14 15 16 17 18 17 18 19 18 19 20 19 18 19 20 21 20 21 22 23 24 25 24 23
##  [26] 22 21 22 21 20 19 20 21 20 19 20 19 20 21 22 21 22 23 22 21 20 19 20 21 20
##  [51] 19 20 19 18 19 18 19 20 19 18 17 16 17 18 17 16 17 18 17 16 17 18 19 20 19
##  [76] 18 19 18 19 18 19 18 19 18 17 16 17 16 15 16 17 16 15 16 17 16 15 14 13 12
## [101] 11 12 13 14 15 14 15 14 13 14 15 14 13 14 15 14 13 12 13 14 15 14 15 16 17
## [126] 18 19 18 19 20 19 20 21 20 21 22 23 22 21 22 23 22 23 24 25 24 25 24 25 24
## [151] 23 22 21 22 23 22 23 24 23 22 21 20 19 20 19 18 19 20 21 20 21 20 19 20 21
## [176] 20 19 20 21 20 19 18 17 18 19 18 17 18 17 16 15 16 17 18 19 18 17 16 17 16
## [201] 17 18 19 18 17 18 19 20 21 20 19 18 19 18 19 18 17 16 15 14 13 14 15 14 15
```

```
## [226] 16 15 14 15 14 13 12 11 10 11 12 13 14 15 14 15 14 15 16 17 18 17 16 17 18
## [251] 17 18 17 18 19 20 21 20 21 20 19 18 17 16 15 14 13 12 11 10  9 10 11 12 13
## [276] 14 13 14 15 16 17 16 15 14 15 16 17 18 17 16 17 16 17 18 17 18 19 18 19 18
```

```
sum(X_filterEstimate == X)/max_t
```

```
## [1] 0.8533333
```

We can see that 85% of the MAP estimates are correct, but we can hopefully improve even further this by using the information of the observations **after** time t as well as those up to time t. This then becomes the problem known as *smoothing*.

Smoothing

We shall find the smoothed estimates for X_t using the Forward-Backward algorithm which uses the forward probabilities $\alpha_i(t)$ that we've already implemented alongside new *backward* probabilities

$$\beta_t(i) = \mathbb{P}(Y_{t+1}, ..., Y_T | X_t = i, \lambda)$$

for each $i \in [N]$ and $t \in [T]$.

Again, we can calculate these inductively:

$$\beta_T(i) = 1$$

$$\beta_t(i) = \underbrace{\sum_{j=1}^{N} a_{ij} b_j(Y_{t+1}) \beta_{t+1}(j)}_{\substack{\text{All of the ways to get to some state } j \\ \text{from state } i \text{ and observe } Y_{t+1}}}$$

```
backwardProbabilities <- function(pi, A, B, Y){
  N = length(pi)
  max_t = length(Y)

  # cols index time steps, t
  # rows index states, i  (these are arbitrary choices)
  log_beta = matrix(rep(0, N*max_t), nrow=N, ncol=max_t)

  log_beta[,max_t] = 0

  for (t in (max_t-1):1){
    for (i in 1:N){
      temp = 0
      for (j in 1:N){
         temp = temp + as.vector(exp(log(A[i,j] * B[j, Y[t+1]+1]) + log_beta[j, t+1]))
      }
      log_beta[i, t] = log(temp)
    }
  }
  return(log_beta)
}

log_beta = backwardProbabilities(pi, A, B, Y)
```

The last part of the Forward-Backward algorithm combines these two sets of probabilities to calculate a third—the smoothed probabilities:

$$\gamma_t(i) = \mathbb{P}(X_t = i | Y, \lambda) = \frac{\mathbb{P}(X_t = i, Y | \lambda)}{\mathbb{P}(Y | \lambda)} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N} \alpha_t(j)\beta_t(j)}$$

The implementation here is simplified by noting that the denominator can be written as $\mathbb{P}(Y|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$.

```
smoothedProbabilities <- function(log_alpha, log_beta){
  max_t = ncol(log_alpha)
  return((log_alpha + log_beta)-log(sum(exp(log_alpha[,max_t]))))
}
log_gamma = smoothedProbabilities(log_alpha, log_beta)
```

We can then get improved MAP estimates via $X_t^{\text{MAP}} = \text{argmax}_i\, \gamma_t(i)$:

```
X_smoothEstimate = apply(log_gamma, 2, which.max)
X_smoothEstimate; X
```

```
##   [1] 13 14 15 16 15 16 17 18 19 18 19 20 19 18 19 20 21 20 21 22 23 24 25 24 23
##  [26] 22 21 22 21 20 19 20 21 20 19 20 19 20 21 22 21 22 23 22 21 20 19 20 19 20
##  [51] 19 20 19 20 19 20 19 20 19 18 17 16 17 16 17 16 17 16 17 16 17 18 19 20 19
##  [76] 18 19 18 19 18 19 18 19 18 17 16 17 16 15 16 17 16 15 16 17 16 15 14 13 12
## [101] 11 12 13 14 13 14 15 14 13 14 15 14 13 14 15 14 13 12 13 14 15 14 15 16 17
## [126] 18 19 18 19 20 19 20 21 20 21 22 23 22 21 22 23 24 23 24 25 24 25 24 25 24
## [151] 23 22 21 22 23 22 23 24 23 22 21 20 19 20 19 18 19 20 21 20 19 20 19 20 19
## [176] 20 19 20 21 20 19 18 17 18 19 18 17 18 17 16 15 16 17 18 19 18 17 16 17 16
## [201] 17 18 17 18 17 18 19 20 21 20 19 18 19 18 19 18 17 16 15 14 13 14 15 14 15
## [226] 16 15 14 15 14 13 12 11 10 11 12 11 12 13 14 15 14 15 16 17 18 17 16 17 18
## [251] 17 18 17 18 19 20 19 20 19 20 19 18 17 16 15 14 13 12 11 10  9 10 11 12 13
## [276] 14 13 14 15 16 17 16 15 14 15 16 17 16 17 16 17 16 17 16 17 18 19 18 19 18
```

```
##   [1] 15 14 15 16 17 18 17 18 19 18 19 20 19 18 19 20 21 20 21 22 23 24 25 24 23
##  [26] 22 21 22 21 20 19 20 21 20 19 20 19 20 21 22 21 22 23 22 21 20 19 20 21 20
##  [51] 19 20 19 18 19 18 19 20 19 18 17 16 17 18 17 16 17 18 17 16 17 18 19 20 19
##  [76] 18 19 18 19 18 19 18 19 18 17 16 17 16 15 16 17 16 15 16 17 16 15 14 13 12
## [101] 11 12 13 14 15 14 15 14 13 14 15 14 13 14 15 14 13 12 13 14 15 14 15 16 17
## [126] 18 19 18 19 20 19 20 21 20 21 22 23 22 21 22 23 22 23 24 25 24 25 24 25 24
## [151] 23 22 21 22 23 22 23 24 23 22 21 20 19 20 19 18 19 20 21 20 21 20 19 20 21
## [176] 20 19 20 21 20 19 18 17 18 19 18 17 18 17 16 15 16 17 18 19 18 17 16 17 16
## [201] 17 18 19 18 17 18 19 20 21 20 19 18 19 18 19 18 17 16 15 14 13 14 15 14 15
## [226] 16 15 14 15 14 13 12 11 10 11 12 13 14 15 14 15 14 15 16 17 18 17 16 17 18
## [251] 17 18 17 18 19 20 21 20 21 20 19 18 17 16 15 14 13 12 11 10  9 10 11 12 13
## [276] 14 13 14 15 16 17 16 15 14 15 16 17 18 17 16 17 16 17 18 17 18 19 18 19 18
```

```
sum(X_smoothEstimate == X)/max_t
```

```
## [1] 0.9333333
```

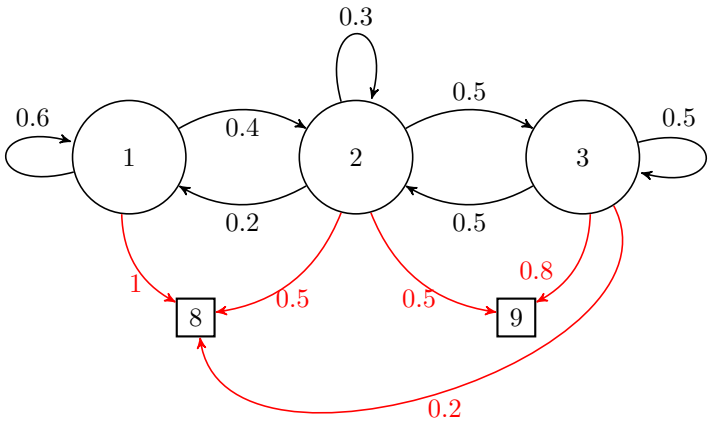
Here it is also useful to note that the entire chain of MAP estimates, both for the smoothed and filtered case, does not give you the most likely full chain of states for X, it is choosing the most likely state for each time step t without considering the estimates at any other time step. Because of this we may arrive at impossible chains of estimates including transitions where none are possible: you can see that the first three states estimated by our filtered values are `14` despite the fact that we know our Markov chain X cannot stay in the same state in consecutive time steps. To deal with this problem and find the most likely entire chain we might use the Viterbi algorithm, which works using similar machinery to the Forward-Backward algorithm, however, it will not be covered in this document.

Parameter Estimation

So far we have dealt with the case where we know the HMM parameters $\lambda = (\pi, A, B)$ beforehand, however, this is very often not the case. Supposing we only receive the set of observations $Y = (Y_1, ..., Y_T)$ (and we know N and O), can we estimate λ?

This problem can be tackled in a variety of ways, however, we will present the Baum-Welch algorithm (which is a special case of the expectation maximisation algorithm) which makes use of the forward, backward and smoothed probabilities we've already been calculating.

To make it easier to see what the algorithm is doing (and to lower the storage requirements), we will return to the simpler example we've used before with the following diagram and supposing that X_1 is equally likely to be any of the states in $S = \{1, 2, 3\}$.




```
pi = rep(1/3, 3)
A  = matrix(c(0.6, 0.4, 0,
              0.2, 0.3, 0.5,
                0, 0.5, 0.5), nrow=3, byrow=T)
B = matrix(c(  1,   0,
             0.5, 0.5,
             0.2, 0.8), nrow=3, byrow=T)
O = c(0,1)
max_t = 100
N = 3
M = 2

HMM = runHMM(pi, A, B, O, 100)
X = HMM$Xs
Y = HMM$Ys
X; Y
```

```
##   [1] 1 1 2 3 2 2 3 3 3 3 2 1 1 2 1 1 1 2 2 2 3 3 2 1 1 1 1 2 3 3 2 3 2 3 3 2 3
##  [38] 2 2 3 2 3 2 3 2 3 3 2 3 2 3 2 2 2 3 2 3 3 2 2 3 2 2 1 2 2 3 2 2 1 1 1 2 2
##  [75] 3 3 2 3 3 2 1 2 3 2 3 2 3 3 2 1 1 1 1 2 3 2 2 3 2 3
```

```
##   [1] 0 0 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 1 1 0 1 0 1 1 1 1
##  [38] 1 1 0 0 1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1 0 0 0 0 0 0
##  [75] 0 1 0 1 1 0 0 0 1 1 1 0 1 1 1 0 0 0 0 1 0 1 1 1 0 1
```

We can see that although this model is simpler than the previous one (in that it has fewer parameters), the more varied behaviour of the Markov chain X and the emission probabilities leads to the MAP estimates of states being less accurate than before.

```
log_alpha = forwardProbabilities(pi, A, B, Y)
log_beta = backwardProbabilities(pi, A, B, Y)
log_gamma = smoothedProbabilities(log_alpha, log_beta)
```

```
X_filterEstimate = apply(log_alpha, 2, which.max)
X_smoothEstimate = apply(log_gamma, 2, which.max)

sum(X_filterEstimate == X)/max_t; sum(X_smoothEstimate == X)/max_t
```

```
## [1] 0.62
```

```
## [1] 0.69
```

To perform the Baum-Welch algorithm we first make some initial guess at the parameter values $\lambda = (\pi, A, B)$. If we have some previous knowledge of the problem that could help with this guess we can use that (this might arrive at the true values faster), however, we will simply make our initial guesses at random.

```
getNormalisedDist <- function(n){
  ps = runif(n)
  return(ps/sum(ps))
}

getRandomLambda <- function(N,M){
  pi = getNormalisedDist(N)
  A  = matrix(rep(0, N*N), nrow=N)
  B  = matrix(rep(0, N*M), nrow=N)

  for(i in 1:N){
    A[i,] = getNormalisedDist(N)
    B[i,] = getNormalisedDist(M)
  }
 lambda = list(pi=pi, A=A, B=B)
 return(lambda)
}
```

The Baum-Welch requires us to calculate one final set of probabilities (whose use will become clear shortly):

$$\xi_t(i,j) = \mathbb{P}(X_t = i, X_{t+1} = j | Y, \lambda) = \frac{\overbrace{\alpha_t(i)}^{\substack{\text{being in state} \\ i \text{ at time } t}} \cdot \overbrace{a_{ij} b_j(Y_{t+1})}^{\substack{\text{moving from } i \text{ to } j \\ \text{and observing } Y_{t+1}}} \cdot \overbrace{\beta_{t+1}(j)}^{\substack{\text{being in state } j \\ \text{at time } t+1}}}{\mathbb{P}(Y|\lambda)}$$
$$= \frac{\alpha_t(i) \cdot a_{ij} b_j(Y_{t+1}) \cdot \beta_{t+1}(j)}{\sum_{k=1}^{N} \sum_{l=1}^{N} \alpha_t(k) \cdot a_{kl} b_l(Y_{t+1}) \cdot \beta_{t+1}(l)}$$

There are $(T-1)N^2$ of these values, which is a lot, hence why we're now using the smaller HMM example. We also provide a function here that will calculate the sums $\sum_{t=1}^{T-1} \xi_t(i,j)$ as is the only form in which these probabilities are used within the Baum-Welch algorithm.

```
smoothedTransitionProbabilities <- function(A, B, Y, log_alpha, log_beta){
  N = nrow(A)
  max_t = length(Y)

  log_xi = list()
  log_denom = log(sum(exp(log_alpha[,max_t])))

  for (t in 1:(max_t-1)){
    log_xi_t = matrix(rep(0, N*N), nrow=N)
    for (i in 1:N){
      for (j in 1:N){
        log_xi_t[i,j] = log_alpha[i,t] + log(A[i,j] * B[j, Y[t+1]+1]) + log_beta[j, t+1] - log_denom
```

```
      }
    }
    log_xi[[t]] = log_xi_t
  }
  return(log_xi)
}

summedSmoothesTransitionProbabilities <- function(log_xi){
  N = nrow(log_xi[[1]])

  xi_sums = matrix(rep(0, N*N), nrow=N)

  for (i in 1:N){
    for (j in 1:N){
      for (t in 1:length(log_xi)){
        xi_sums[i,j] = xi_sums[i,j] + exp(log_xi[[t]][i,j])
      }
    }
  }

  return(xi_sums)
}
```

The final step of the Baum-Welch algorithm, given some parameter guesses $\lambda = (\pi, A, B)$, is to update them as follows to obtain $\lambda' = (\pi', A', B')$:

$$\bar{\pi}_i = \mathbb{P}(X_1 = i | Y, \lambda) = \gamma_1(i)$$

$$\bar{a}_{ij} = \frac{\text{expected number of } i\text{-to-}j \text{ transitions}}{\text{expected number of } i\text{-to-}k \text{ transitions } \forall k} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

$$\bar{b_j}(o_k) = \frac{\text{expected number of } o_k \text{ observations from state } j}{\text{expected number of time steps in state } j} = \frac{\sum_{t=1}^{T} \mathbf{1}_{\{Y_t = o_k\}} \cdot \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

This update procedure is then repeated with the new parameter estimates until they stabilise into a local optimum. There's no guarantee of finding the global optimum, however, given enough time the Baum-Welch algorithm (being a special case of the EM algorithm) will find a local optimum.

```
baumwelch <- function(N, M, Y, max_iter){
  max_t = length(Y)

  lambda = getRandomLambda(N,M)
  pi = lambda$pi
  A  = lambda$A
  B  = lambda$B

  for (l in 1:max_iter){
    # E step
    log_alpha = forwardProbabilities(pi, A, B, Y)
    log_beta  = backwardProbabilities(pi, A, B, Y)
    log_gamma = smoothedProbabilities(log_alpha, log_beta)
    log_xi    = smoothedTransitionProbabilities(A, B, Y, log_alpha, log_beta)
    xi_sums   = summedSmoothesTransitionProbabilities(log_xi)

    # M step
```

```
    pi_new = exp(log_gamma[,1])

    A_new = matrix(rep(0, N*N), nrow=N)
    B_new = matrix(rep(0, N*M), nrow=N)

    for (i in 1:N){
      gamma_sum = sum(exp(log_gamma[i,1:max_t]))
      for (j in 1:N){
        A_new[i,j] = xi_sums[i,j] / (gamma_sum - exp(log_gamma[i,max_t]))
      }

      for (k in 1:M){
        numerator = 0
        for (t in 1:max_t){
          if (Y[t]==k-1){
            numerator = numerator + exp(log_gamma[i,t])
          }
        }
        B_new[i, k] = numerator / gamma_sum
      }
    }

    pi = pi_new
    A  = A_new
    B  = B_new
    lambda = list(pi=pi, A=A, B=B)
  }

  lambda = list(pi=pi, A=A, B=B)
  return(lambda)
}

lambda = baumwelch(N, M, Y, 50)
lambda
```

```
## $pi
## [1] 1.000000e+00 1.406180e-58 1.548412e-40
##
## $A
##            [,1]      [,2]       [,3]
## [1,] 0.72084307 0.2282737 0.05088326
## [2,] 0.02795314 0.4160196 0.55602728
## [3,] 0.17651909 0.6762170 0.14726391
##
## $B
##           [,1]         [,2]
## [1,] 1.0000000 2.358644e-10
## [2,] 0.0857058 9.142942e-01
## [3,] 0.4972550 5.027450e-01
```

```
pi; A; B
```

```
## [1] 0.3333333 0.3333333 0.3333333
```

```
##      [,1] [,2] [,3]
```

```
## [1,]  0.6  0.4  0.0
## [2,]  0.2  0.3  0.5
## [3,]  0.0  0.5  0.5
```

```
##      [,1] [,2]
## [1,]  1.0  0.0
## [2,]  0.5  0.5
## [3,]  0.2  0.8
```

We can see that the estimates produced, whilst not perfect, manage to capture lots of the behaviour present in our HMM. In particular, the algorithm has managed to get very impressive estimates for the emission probabilities B when $X_t \in \{1, 3\}$, even if the $b_2(o_k)$ probabilities are very far off the reality. Similarly for the transition probabilities A we see somewhat accurate estimates for the first two rows (though by no means perfect), but the third row is quite wrong—we can see here that we've ended up in a local optimum.

We also see that π has been estimated as essentially $[1, 0, 0]$, which agrees with the fact that $X_1 = 1$:

```
X[1]
```

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

However, clearly this isn't a good estimate of π, for that we really need multiple observed sequences $\{Y^k\}_{k=1}^K$ where each $Y^k = (Y_1^K, ..., Y_{T_k})$ corresponds to a different sequence of Markov chain states $X^k = (X_1^k, ..., X_{t_k}^k)$, each of (potentially different) length T_k. With these, we can simply run the Baum-Welch algorithm K times (once on each observation-sequence) and then generate a final estimate of π by averaging each corresponding entry in the K different estimates we obtain. (Sadly doing the same for A and B doesn't lead to much improvement as the local optima they get stuck in more complex than π's one-hot vectors and so averaging out doesn't work quite so well. We have written the code for A and B here anyway so that this behaviour can be observed.)

```
K = 100

pi_estimates = list()
A_estimates  = list()
B_estimates  = list()

# Create a list of partial-means for each parameter
pi_partialMeans = list()
A_partialMeans  = list()
B_partialMeans  = list()

getPartialMean <- function(Zs){
  partial_K = length(Zs)
  temp = Zs[[1]]
  if (partial_K > 1){
    for (k in 2:partial_K){
      temp = temp + Zs[[k]]
    }
  }
  return(temp/partial_K)
}

for (k in 1:K){
  max_t_k = sample(50:150, 1)

  HMM = runHMM(pi, A, B, O, max_t_k)
  Y = HMM$Ys
```

```
  lambda = baumwelch(N, M, Y, 100)
  pi_estimates[[k]] = lambda$pi
  A_estimates[[k]]  = lambda$A
  B_estimates[[k]]  = lambda$B

  pi_partialMeans[[k]] = getPartialMean(pi_estimates)
  A_partialMeans[[k]]  = getPartialMean(A_estimates)
  B_partialMeans[[k]]  = getPartialMean(B_estimates)
}

pi_partialMeans[K]; A_partialMeans[K]; B_partialMeans[K]
```

```
## [[1]]
## [1] 0.3897003 0.3003291 0.3099706
```

```
## [[1]]
##           [,1]      [,2]      [,3]
## [1,] 0.4231670 0.3179532 0.2588798
## [2,] 0.3147531 0.3949246 0.2903224
## [3,] 0.2855859 0.2747773 0.4396368
```

```
## [[1]]
##           [,1]      [,2]
## [1,] 0.5729684 0.4270316
## [2,] 0.4763946 0.5236054
## [3,] 0.4841607 0.5158393
```

```
pi; A; B
```

```
## [1] 0.3333333 0.3333333 0.3333333
```

```
##      [,1] [,2] [,3]
## [1,]  0.6  0.4  0.0
## [2,]  0.2  0.3  0.5
## [3,]  0.0  0.5  0.5
```

```
##      [,1] [,2]
## [1,]  1.0  0.0
## [2,]  0.5  0.5
## [3,]  0.2  0.8
```

From this we see an improved overall estimate for π—by repeating the Baum-Welch algorithm multiple times with random initial guesses each time we manage to explore more of the search space. However, as mentioned before, the estimates for A and B are actually worse, each row seeming to move towards a uniform distribution.