GSA

404018

January 2021

1 Implement an HMM

Write a program which implements a HMM, reading the model and parameters from a file.

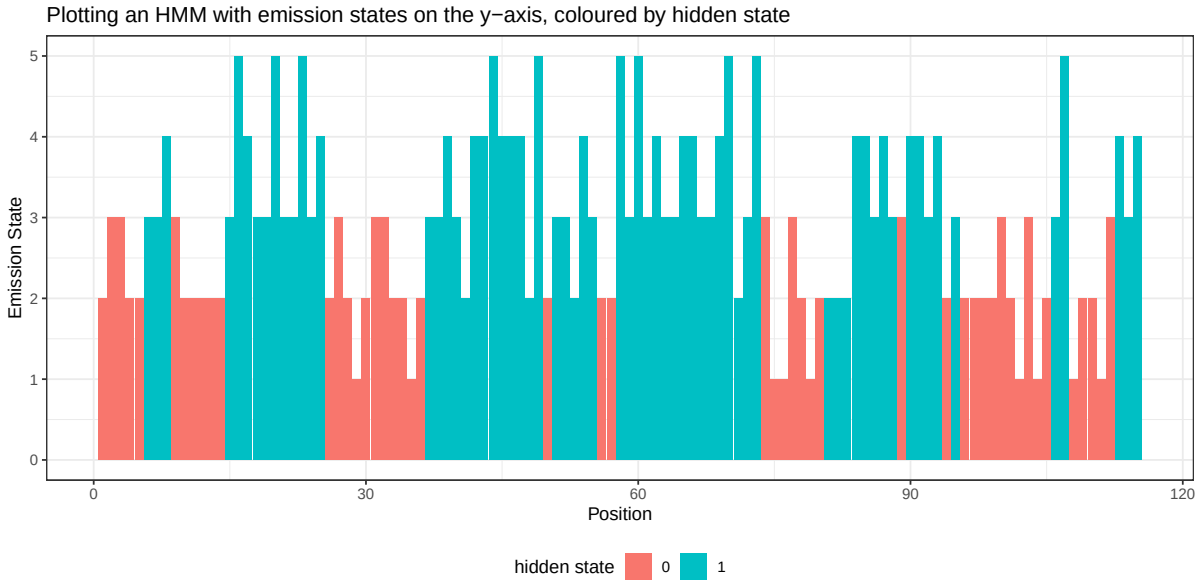
1.1 Implementation

Here you can see that I load in the HMM model and parameters and use the following function (adapted from the notes) to implement the HMM. I read in the model and parameters as a .Rdata file.

```
# Read in the data
load("~/GSA/HMM_model.Rdata")

# Make function to emit chain
hidden_markov <- function() {
    # Function to take hidden markov chain and simulate emission states

    # Set up vectors to hold output
    states <- data.frame(chain = rep(0, clen), emission = rep(0, clen))

    rownames(A) <- rownames(B) <- S

    # Set initial hidden and emission states given initial distribution and
    # emission matrix
    states$chain[1] <- sample(S, 1, prob = Mu_0)
    states$emission[1] <- sample(V, 1, prob = B[as.character(states$chain[1]),
        ])

    # Work through states and emit based on transmission probabilities
    for (i in 2:clen) {
        states$chain[i] <- sample(S, 1, prob = A[as.character(states$chain[i -
            1]), ])
        states$emission[i] <- sample(V, 1, prob = B[as.character(states$chain[i]),
            ])
    }

    return(states)
}

states <- hidden_markov()
```

1.2 Plotting

I then plotted the output.


Plotting an HMM with emission states on the y−axis, coloured by hidden state
Emission State
0
1
2
3
4
5
0
30
60
90
120
Position
hidden state
0
1

2 Implement the forward algorithm

I implemented the unscaled forward algorithm to calculate the likelihood of this sequence. Note that I have attempted to match the notation in the notes as closely as possible here and throughout to make it clear what I am doing. I read the data from a .Rdata file.

```
# Read states from a file
load("~/GSA/Emission_states.Rdata")

likelihood_forward <- function(states, S, V, A, Mu_0, B) {
    # Function to implement the forward algorithm

    # Set up dataframe
    alphas <- matrix(ncol = length(Mu_0), nrow = length(states))

    # Set up initialisation states
    alphas[1, ] <- Mu_0 * B[, states[1]]

    # Set up recursion relationship
    for (n in 2:length(states)) {

        # Multiply prev by transition, then the hidden state probabilities by the
        # emission probabilities
        alphas[n, ] <- (alphas[n - 1, ] %*% A) * B[, states[n]]
    }
    return(alphas)
}

likelihood_tab <- likelihood_forward(states, S, V, A, Mu_0, B)
```

Here you can see that the likelihood under this model was 2.29581×10^{-71}.

3 Implement this on the yeast chromosome

3.1 Download and windows

I downloaded the *S cerevisiae* genome from NCBI and split the sequence into 100bp adjacent windows. I considered a sliding window however this would make the adjacent windows at least partially dependent on each other and not just on the hidden sequence, hence I chose non-overlapping windows.

3.2 Binning scheme

I chose a binning scheme so that the proportion of the windows within each GC content range matched the proportion of the respective emitted states in the sequence I generated.

3.3 Scaled forward algorithm

Next I calculated the log likelihood of this sequence given the model using the scaled forward algorithm. This function is very similar to the unscaled forward algorith but just includes a c_n term. The log likelihood under this model was -4448.5.

4 Implementing Baum-Welch Estimation

I implemented the Baum-Welch algorithm, building the function up in stages

4.1 Scaled backward algorithm

First I implemented the scaled backward algorithm. As you can see for the linear algebra to work the $\hat{\beta}$s must be multiplied by the emission probabilities and only then by the inverse transition matrix for the equation to output correctly.

```
scaled_backward <- function(states, S, V, A, Mu_0, B) {
    # First build function to calculate scaled backward variable

    # Set up dataframe
    beta_hats <- matrix(ncol = length(Mu_0), nrow = length(states))

    # Run scaled forward algorithm
    forward <- log_likelihood_forward(states, S, V, A, Mu_0, B)
    c_n <- forward[[1]]
    alpha_hats <- forward[[2]]

    # Set up initialisation states
    beta_hats[length(states), ] <- 1/c_n[length(c_n)]

    # Set up recursion relationship
    for (n in (length(states) - 1):1) {

        # Calculate beta_hats[n]
        beta_hats[n, ] <- ((beta_hats[n + 1, ] * B[, states[n + 1]]) %*% t(A))/c_n[n]
    }

    return(list(c_n, alpha_hats, beta_hats))
}
```

4.2 Updating theta

Here I made all the functions the functions to update theta.

4.2.1 $E(n_{ij}|Y;\theta)$

I calculated $E(n_{ij})$ for all $n = 1 : (N-1)$ separately for every i and j.

```
# Calculate A prime - first caculate E(n_ij)
exp_n_ij <- function(alpha_hats, beta_hats, states, A, B, i, j) {

    # Define function to apply across all locations
    func <- function(alpha_hats, A, beta_hats, B, n, i, j) {
        alpha_hats[n, i] * A[i, j] * B[j, states[n + 1]] * beta_hats[n + 1,
            j]
    }

    # Apply function to all 0 to N-1
    sum(sapply(1:(length(states) - 1), func, alpha_hats = alpha_hats, beta_hats = beta_hats,
        A = A, B = B, i = i, j = j))

}
```

4.2.2 A'

Next I used the $E(n_{ij})$ function to calculate A'. I worked through the matrix for each i and j and afterwards divided through by $\sum_{k=1}^{J} E(n_{ik})$.

4.2.3 B'

The function to make B' first makes a function to perform the indicator function. It next sums this from $n = 1 : (N-1)$. It then takes the sum of this for $j = 1 : J$ for each i and k. For each part of the matrix B it then divides by $\sum_{k=1}^{J} E(n_{ik})$.

```
# Next calculate B_prime
B_prime_func <- function(alpha_hats, beta_hats, states, A, B) {

    # Make B prime matrix
    B_prime <- B

    # Function to do non-summed numerator
    B_func <- function(alpha_hats, A, beta_hats, B, n, i, j, k, states) {
        if (states[n] == k) {
            return(alpha_hats[n, i] * A[i, j] * B[j, states[n + 1]] * beta_hats[n +
                1, j])
        } else {
            return(0)
        }
    }

    # Fill B prime matrix - working through the matrix by row then column
    for (trans_i in 1:dim(B)[1]) {
        for (trans_k in 1:dim(B)[2]) {
            B_prime_ik <- c()
            for (trans_j in 1:dim(B)[1]) {

                # Calculate numerator using function before, this time summing
                B_prime_ik[trans_j] <- sum(sapply(1:(length(states) - 1), B_func,
                  alpha_hats = alpha_hats, beta_hats = beta_hats, A = A, B = B,
                  i = trans_i, j = trans_j, states = states, k = trans_k))
            }

            # Normalise by the summed expected value of E(n_ij)
            B_prime[trans_i, trans_k] <- sum(B_prime_ik)/sum(sapply(1:2, exp_n_ij,
                i = trans_i, alpha_hats = alpha_hats, beta_hats = beta_hats,
                states = states, A = A, B = B))

        }

    }

    return(B_prime)
}
```

4.2.4 μ'

Next I calculated μ'. For this I changed the order of operations until it matched the equation using linear algebra. I used the inverse transition matrix to make the output correct.

Table 1: Updated parameters for A

0.9403950	0.0596050
0.1309546	0.8690454

Table 2: Updated parameters for B

0.1475381	0.4031445	0.3463567	0.1029607	0.0000000
0.0000000	0.0463246	0.2952957	0.4402619	0.2181177

4.3 Baum-Welch Function

Next I made a function to put it all together, recalculating the probabilities, $\hat{\beta}$, $\hat{\alpha}$ and updating θ at every iteration.

4.4 Results

Finally the results from the Baum-Welch estimation. The log likelihood under this model was -4344.81

Table 3: Updated parameters for Mu

0	1

5 Infer the most likely sequence of hidden states

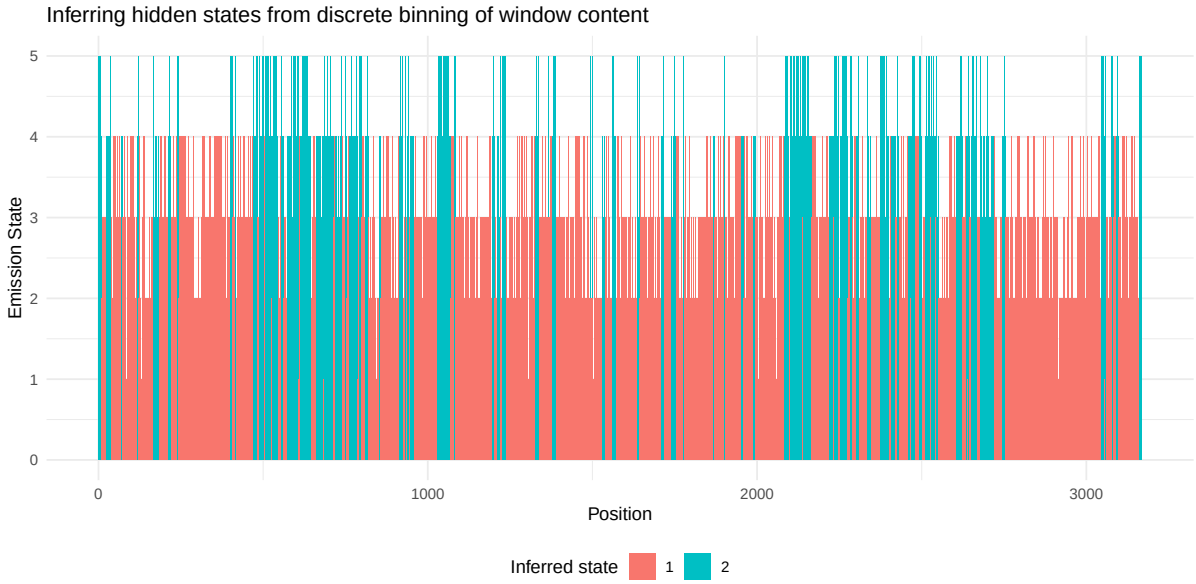
5.1 Viterbi

I used the viterbi algorithm to infer the most likely sequence of hidden states.

```
viterbi_inference <- function(states, Mu_0, S, V, B, A) {

    # Set up initial parameters
    N <- length(states)
    phi <- psi <- matrix(nrow = N, ncol = length(Mu_0))
    phi[1, ] <- log(Mu_0) + log(B[, states[1]])

    # Run recursion relationship
    for (n in 2:N) {

        # Recursion of phi
        to.max <- log(A) + phi[n - 1, ]

        # Do maximisation and add to other term
        phi[n, ] <- log(B[, states[n]]) + apply(to.max, 2, max)

        # Recursion of psi - do argmax
        psi[n, ] <- apply(to.max, 2, function(x) (which(x == max(x))))
    }

    ## Do traceback
    i_star <- rep(NA, times = N)

    # Get Nth value
    i_star[N] <- which(phi[N, ] == max(phi[N, ]))

    # Trace back along vector
    for (n in (N - 1):1) {
        i_star[n] <- psi[n + 1, i_star[n + 1]]
    }

    return(i_star)
}
```

5.2 Results

Here you can see the results from the viterbi inference algorithm. The significance of this is that you can then infer hidden states from known sequences. In the context of GC content, you could use an HMM (as here) to infer whether or not a certain section of sequence was a CpG island, and therefore by inferring the hidden sequence you can infer all of the locations of CpG islands. This is important for a number of reasons including locating promotor regions.


Inferring hidden states from discrete binning of window content
Emission State
5
4
3
2
1
0
0
1000
2000
3000
Position
Inferred state
1
2

6 Alternative choice of emission distribution

Other ways to implement this would be using more emission states, or using a continuous emission distribution. There are a number of possible emission distributions to choose from including gaussian, gaussian mixture or a poisson distribution. Here I have implemented a gaussian continuous emission distribution. Many of the functions above needed to be amended to allow a continous distribution to be used. I integrated a small portion of the emission distribution to get the probability of the emission state.

6.1 Amended forward algorithm

Here you can seee that the forward algorithm had to be amended to allow a continuous emission distribution to be used using the integrate function. I also updated the backward algorithm in the same way as the forward.

```
log_likelihood_forward_cont <- function(states, S, V, A, Mu_0, input_mean, input_sd) {
    # Function to implement the forward algorithm with continuous distribution

    # Set up dataframe - first col is c_n, rest are alpha_hat
    c_n <- vector(length = length(states))
    alpha_hats <- matrix(ncol = length(Mu_0), nrow = length(states))

    # Make integral
    b_state_1 <- integrate(dnorm, mean = input_mean[1], sd = input_sd[1], lower = states[1] -
        0.005, upper = states[1] + 0.005, abs.tol = 0)$value
    b_state_2 <- integrate(dnorm, mean = input_mean[2], sd = input_sd[2], lower = states[1] -
        0.005, upper = states[1] + 0.005, abs.tol = 0)$value
    b_states <- c(b_state_1, b_state_2)

    # Set up initialisation states
    c_n[1] <- sum(Mu_0 * b_states)
    alpha_hats[1, ] <- Mu_0 * b_states/c_n[1]

    # Set up recursion relationship
    for (n in 2:length(states)) {

        # Make integral
        b_state_1 <- integrate(dnorm, mean = input_mean[1], sd = input_sd[1],
            lower = states[n] - 0.005, upper = states[n] + 0.005, abs.tol = 0)$value
        b_state_2 <- integrate(dnorm, mean = input_mean[2], sd = input_sd[2],
            lower = states[n] - 0.005, upper = states[n] + 0.005, abs.tol = 0)$value
        b_states <- c(b_state_1, b_state_2)

        # Calculate c_n
        c_n[n] <- sum((alpha_hats[n - 1, ] %*% A) * b_states)

        # Calculate alpha_hat[n] (which is essentially alpha normalised by the sum
        # of the alphas - c_n)
        alpha_hats[n, ] <- ((alpha_hats[n - 1, ] %*% A) * b_states)/c_n[n]

    }

    # Completion - sum the logs
    return(list(c_n, alpha_hats))

}
```

6.2 λ'

I calculated λ' using the equations from the notes as below

```
# Function to do non-summed numerator
lambda_func <- function(alpha_hats, A, beta_hats, input_mean, input_sd, n, i,
    j, states) {

    # Do integral
    b_state <- integrate(dnorm, mean = input_mean[j], sd = input_sd[j], lower = states[n +
        1] - 0.005, upper = states[n + 1] + 0.005, abs.tol = 0)$value
    # Return numerator
    states[n] * alpha_hats[n, i] * A[i, j] * b_state * beta_hats[n + 1, j]

}

# Next calculate new mean
lambda_prime_func <- function(alpha_hats, beta_hats, states, A, input_mean,
    input_sd) {

    lambda_prime <- c()

    # Fill B prime matrix - working through the matrix by row then column
    for (trans_i in 1:length(Mu_0)) {
        lambda_prime_ij <- c()
        for (trans_j in 1:length(Mu_0)) {

            # Calculate numerator using function before, this time summing
            lambda_prime_ij[trans_j] <- sum(sapply(1:(length(states) - 1), lambda_func,
                alpha_hats = alpha_hats, beta_hats = beta_hats, A = A, input_mean = input_mean,
                input_sd = input_sd, i = trans_i, j = trans_j, states = states))
        }

        # Normalise by the summed expected value of E(n_ik)
        lambda_prime[trans_i] <- sum(lambda_prime_ij)/sum(sapply(1:2, exp_n_ij_cont,
            i = trans_i, alpha_hats = alpha_hats, beta_hats = beta_hats, states = states,
            A = A, input_mean = input_mean, input_sd = input_sd))


    }

    return(lambda_prime)
}
```

6.3 σ'

I used the equation from page 35 in the notes to derive σ' as this was incompletely defined.

```
# make functions to go inside sigma_prime
sigma_func_numerator <- function(states, alpha_hats, beta_hats, c_n, input_mean,
    n, i) {

    # This is just the unsummed numerator derived from the MAP bit:
    # c_n*alpha^hat_n(i)*beta^hat_n(i) = P(X_n = s_i | Y)
    (states[n] - input_mean)^2 * c_n[n] * alpha_hats[n, i] * beta_hats[n, i]
}

sigma_func_denominator <- function(alpha_hats, beta_hats, c_n, n, i) {
```

```
    # This is just the unsummed denominator as above for P(X_n = s_i | Y)
    c_n[n] * alpha_hats[n, i] * beta_hats[n, i]
}

# Next calculate the new standard deviation
sigma_prime_func <- function(states, alpha_hats, beta_hats, c_n, input_mean) {

    sigma_prime <- c()

    for (i in 1:length(Mu_0)) {

        # Calculate the numerator
        numerator <- sum(sapply(1:(length(states) - 1), sigma_func_numerator,
            alpha_hats = alpha_hats, beta_hats = beta_hats, c_n = c_n, input_mean = input_mean,
            i = i, states = states))

        # Calculate the denominator
        denominator <- sum(sapply(1:(length(states) - 1), sigma_func_denominator,
            alpha_hats = alpha_hats, beta_hats = beta_hats, c_n = c_n, i = i))

        sigma_prime[i] <- (numerator/denominator)^0.5

    }

    return(sigma_prime)

}
```

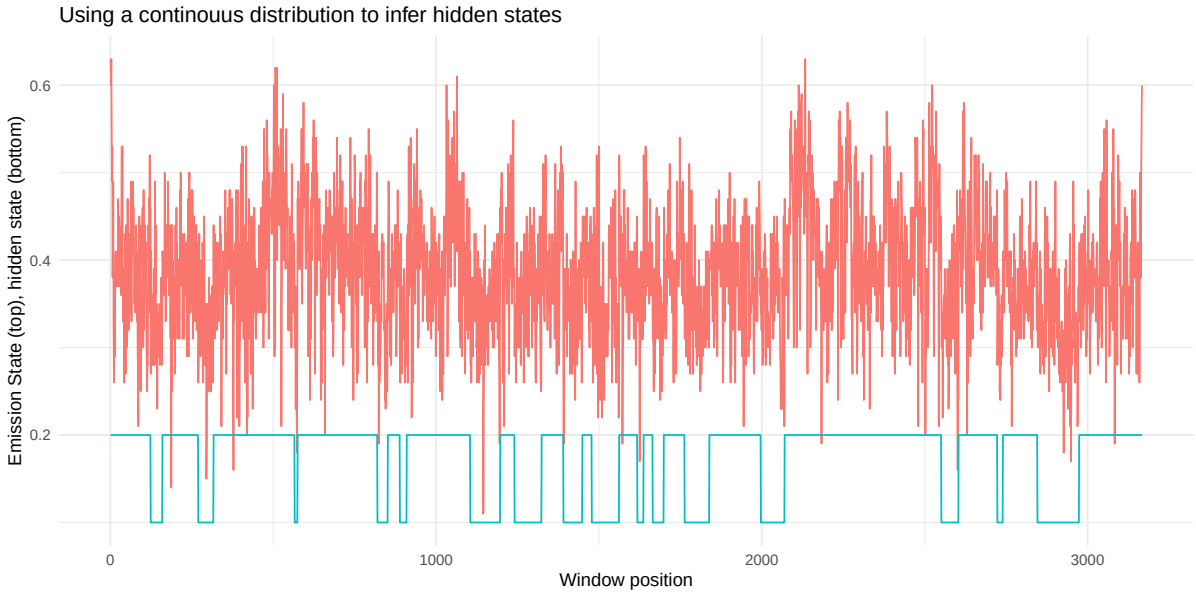
6.4 Other equations and Baum-Welch

I also had to redefine some of the other equations used for the Baum-Welch estimation, which I did in a similar way to the forward algorithm using the integrate function.

6.5 Updated Viterbi and Results

I updated the Viterbi algorithm to include a continous distribution using the itegrate function as before.

Finally I used the updated Viterbi algorithm to infer the hidden states from the raw GC content of the windows. You can seee that the regions inferred to be in the hidden state are similar to those inferred by the discrete distribution, however are somewhat larger. This is possibly because the continuous sequence does not depend on the binning scheme which was set according to the distribution of states from the original HMM produced.



The log likelihood under this model was -1.06796×10^4. The means for this model were 0.3479608 for the first hidden state, 0.4115032 for the second hidden state. The standard deviation for this model were 0.1097298 for the first hidden state and 0.1034529 for the second.

7 Appendix with all code used

```
# Read in the data
load("~/GSA/HMM_model.Rdata")

# Make function to emit chain
hidden_markov <- function() {
    # Function to take hidden markov chain and simulate emission states

    # Set up vectors to hold output
    states <- data.frame(chain = rep(0, clen), emission = rep(0, clen))

    rownames(A) <- rownames(B) <- S

    # Set initial hidden and emission states given initial distribution and
    # emission matrix
    states$chain[1] <- sample(S, 1, prob = Mu_0)
    states$emission[1] <- sample(V, 1, prob = B[as.character(states$chain[1]),
        ])

    # Work through states and emit based on transmission probabilities
    for (i in 2:clen) {
        states$chain[i] <- sample(S, 1, prob = A[as.character(states$chain[i -
            1]), ])
        states$emission[i] <- sample(V, 1, prob = B[as.character(states$chain[i]),
            ])
    }

    return(states)
}

states <- hidden_markov()

ggplot(states) + geom_col(aes(x = 1:115, y = emission, fill = as.factor(chain))) +
    theme_bw() + theme(legend.position = "bottom") + xlab("Position") + ylab("Emission State") +
    labs(title = "Plotting an HMM with emission states on the y-axis, coloured by hidden state",
        fill = "hidden state")

hidden.states <- states$chain
states <- states$emission
save(states, file = "~/GSA/Emission_states.Rdata")

# Read states from a file
load("~/GSA/Emission_states.Rdata")

likelihood_forward <- function(states, S, V, A, Mu_0, B) {
    # Function to implement the forward algorithm

    # Set up dataframe
    alphas <- matrix(ncol = length(Mu_0), nrow = length(states))

    # Set up initialisation states
    alphas[1, ] <- Mu_0 * B[, states[1]]

    # Set up recursion relationship
    for (n in 2:length(states)) {
```

```
        # Multiply prev by transition, then the hidden state probabilities by the
        # emission probabilities
        alphas[n, ] <- (alphas[n - 1, ] %*% A) * B[, states[n]]
    }
    return(alphas)
}

likelihood_tab <- likelihood_forward(states, S, V, A, Mu_0, B)

S_cerevisiae_III <- readLines("~/GSA/S_cereviciae_III.fasta")
S_cerevisiae_III <- S_cerevisiae_III[-c(1:3)]
S_cerevisiae_III <- paste(S_cerevisiae_III, collapse = "")
S_cerevisiae_III <- strsplit(S_cerevisiae_III, split = "")[[1]]

GC_content <- function(input, loc) {
    # Function to find GC content in any 100bp window

    # Get window
    window.vec <- input[seq(loc, loc + 99)]
    # Return GC content as percentage
    length(which(window.vec %in% c("G", "C")))/length(which(is.na(window.vec) ==
        FALSE))
}

# Apply function to get GC content for all windows
GC_window_content <- sapply(seq(1, length(S_cerevisiae_III), by = 100), GC_content,
    input = S_cerevisiae_III)

# Get emission state probabilities for HMM at beginning
bin_prop <- sapply(sort(unique(states)), function(x, y) (length(which(y == x))),
    y = states)
names(bin_prop) <- sort(unique(states))
bin_prop <- bin_prop/length(states)

# Get GC contents corresponding to each bin
bin_cumulative <- sapply(1:length(bin_prop), function(x, y) (sum(y[1:x])), y = bin_prop)
gc_bins <- c()
for (i in 1:length(bin_cumulative)) {
    gc_bins[i] <- max(sort(GC_window_content)[seq(1, floor(bin_cumulative[i] *
        length(GC_window_content)))])
}

# Calculate emission sequence under this model
GC_emission_seq <- rep(NA, length(GC_window_content))
GC_emission_seq[which(GC_window_content <= gc_bins[1])] <- 1
for (i in 2:length(gc_bins)) {
    GC_emission_seq[which(GC_window_content <= gc_bins[i] & GC_window_content >
        gc_bins[i - 1])] <- i
}



# Calculate A prime - first caculate E(n_ij)
exp_n_ij <- function(alpha_hats, beta_hats, states, A, B, i, j) {

    # Define function to apply across all locations
    func <- function(alpha_hats, A, beta_hats, B, n, i, j) {
        alpha_hats[n, i] * A[i, j] * B[j, states[n + 1]] * beta_hats[n + 1,
```

```
            j]
    }

    # Apply function to all 0 to N-1
    sum(sapply(1:(length(states) - 1), func, alpha_hats = alpha_hats, beta_hats = beta_hats,
        A = A, B = B, i = i, j = j))

}

# Function to make A prime
A_prime_func <- function(alpha_hats, beta_hats, states, A, B, Mu_0) {

    # Make A prime matrix
    A_prime <- matrix(nrow = length(Mu_0), ncol = length(Mu_0))

    # Fill A prime matrix
    for (i in 1:length(Mu_0)) {
        for (j in 1:length(Mu_0)) {
            A_prime[i, j] <- exp_n_ij(alpha_hats, beta_hats, states, A, B, i,
                j)
        }
    }

    # Normalise A prime matrix by row sums and return it - this poss should be
    # col? - previously tried with inverse matrix
    t(apply(A_prime, 1, function(x) (x/sum(x))))
}

# Next calculate B_prime
B_prime_func <- function(alpha_hats, beta_hats, states, A, B) {

    # Make B prime matrix
    B_prime <- B

    # Function to do non-summed numerator
    B_func <- function(alpha_hats, A, beta_hats, B, n, i, j, k, states) {
        if (states[n] == k) {
            return(alpha_hats[n, i] * A[i, j] * B[j, states[n + 1]] * beta_hats[n +
                1, j])
        } else {
            return(0)
        }
    }

    # Fill B prime matrix - working through the matrix by row then column
    for (trans_i in 1:dim(B)[1]) {
        for (trans_k in 1:dim(B)[2]) {
            B_prime_ik <- c()
            for (trans_j in 1:dim(B)[1]) {

                # Calculate numerator using function before, this time summing
                B_prime_ik[trans_j] <- sum(sapply(1:(length(states) - 1), B_func,
                  alpha_hats = alpha_hats, beta_hats = beta_hats, A = A, B = B,
                  i = trans_i, j = trans_j, states = states, k = trans_k))
            }

            # Normalise by the summed expected value of E(n_ij)
```

```
            B_prime[trans_i, trans_k] <- sum(B_prime_ik)/sum(sapply(1:2, exp_n_ij,
                i = trans_i, alpha_hats = alpha_hats, beta_hats = beta_hats,
                states = states, A = A, B = B))

        }

    }

    return(B_prime)
}

# Next calculate Mu_prime
Mu_prime_func <- function(alpha_hats, beta_hats, states, A, B) {

    # Order set to output correctly
    alpha_hats[1, ] * ((B[, states[2]] * beta_hats[2, ]) %*% t(A))

}

baum_welch <- function(states, S, V, A, Mu_0, B, threshold) {
    # Function to calculate the expected values of theta

    # First get backward and forward variables
    alpha_beta <- scaled_backward(states, S, V, A, Mu_0, B)
    c_n <- alpha_beta[[1]]
    alpha_hats <- alpha_beta[[2]]
    beta_hats <- alpha_beta[[3]]

    # Now iterate over t until threshold < value set above
    L_t <- sum(log(log_likelihood_forward(states, S, V, A, Mu_0, B)[[1]]))
    delta <- 10000

    while (delta > threshold) {

        # First get backward and forward variables
        alpha_beta <- scaled_backward(states, S, V, A, Mu_0, B)
        c_n <- alpha_beta[[1]]
        alpha_hats <- alpha_beta[[2]]
        beta_hats <- alpha_beta[[3]]

        # Get updated parameters
        B_prime <- B_prime_func(alpha_hats, beta_hats, states, A, B)

        A_prime <- A_prime_func(alpha_hats, beta_hats, states, A, B, Mu_0)

        Mu_prime <- Mu_prime_func(alpha_hats, beta_hats, states, A, B)

        L_next <- sum(log(log_likelihood_forward(states, S, V, A_prime, Mu_prime,
            B_prime)[[1]]))

        delta <- abs(L_t - L_next)

        # Now update the parameters (theta_t -> theta_t+1)
        B <- B_prime
        A <- A_prime
        Mu_0 <- Mu_prime
        L_t <- L_next
```

```
    }

    return(list(A, B, Mu_0, L_t))
}

updated_theta <- baum_welch(GC_emission_seq, S, V, A, Mu_0, B, 1e-10)
updated_A <- updated_theta[[1]]
kable(updated_A, caption = "Updated parameters for A")
updated_B <- updated_theta[[2]]
kable(updated_B, caption = "Updated parameters for B")
updated_Mu <- updated_theta[[3]]
kable(updated_Mu, caption = "Updated parameters for Mu")

inferred.states <- viterbi_inference(GC_emission_seq, updated_Mu, S, V, updated_B,
    updated_A)
ggplot() + geom_col(aes(x = 1:length(GC_emission_seq), y = GC_emission_seq,
    fill = as.factor(inferred.states))) + theme_minimal() + labs(fill = "Inferred state") +
    theme(legend.position = "bottom") + xlab("Position") + ylab("Emission State") +
    labs(title = "Inferring hidden states from discrete binning of window content")

log_likelihood_forward_cont <- function(states, S, V, A, Mu_0, input_mean, input_sd) {
    # Function to implement the forward algorithm with continuous distribution

    # Set up dataframe - first col is c_n, rest are alpha_hat
    c_n <- vector(length = length(states))
    alpha_hats <- matrix(ncol = length(Mu_0), nrow = length(states))

    # Make integral
    b_state_1 <- integrate(dnorm, mean = input_mean[1], sd = input_sd[1], lower = states[1] -
        0.005, upper = states[1] + 0.005, abs.tol = 0)$value
    b_state_2 <- integrate(dnorm, mean = input_mean[2], sd = input_sd[2], lower = states[1] -
        0.005, upper = states[1] + 0.005, abs.tol = 0)$value
    b_states <- c(b_state_1, b_state_2)

    # Set up initialisation states
    c_n[1] <- sum(Mu_0 * b_states)
    alpha_hats[1, ] <- Mu_0 * b_states/c_n[1]

    # Set up recursion relationship
    for (n in 2:length(states)) {

        # Make integral
        b_state_1 <- integrate(dnorm, mean = input_mean[1], sd = input_sd[1],
            lower = states[n] - 0.005, upper = states[n] + 0.005, abs.tol = 0)$value
        b_state_2 <- integrate(dnorm, mean = input_mean[2], sd = input_sd[2],
            lower = states[n] - 0.005, upper = states[n] + 0.005, abs.tol = 0)$value
        b_states <- c(b_state_1, b_state_2)

        # Calculate c_n
        c_n[n] <- sum((alpha_hats[n - 1, ] %*% A) * b_states)

        # Calculate alpha_hat[n] (which is essentially alpha normalised by the sum
        # of the alphas - c_n)
        alpha_hats[n, ] <- ((alpha_hats[n - 1, ] %*% A) * b_states)/c_n[n]

    }
```

```
    # Completion - sum the logs
    return(list(c_n, alpha_hats))

}

# Function to make scaled backward with continuous distribution
scaled_backward_cont <- function(states, S, V, A, Mu_0, input_mean, input_sd) {
    # First build function to calculate scaled backward variable

    # Set up dataframe
    beta_hats <- matrix(ncol = length(Mu_0), nrow = length(states))

    # Run scaled forward algorithm
    forward <- log_likelihood_forward_cont(states, S, V, A, Mu_0, input_mean,
        input_sd)
    c_n <- forward[[1]]
    alpha_hats <- forward[[2]]

    # Set up initialisation states
    beta_hats[length(states), ] <- 1/c_n[length(c_n)]

    # Set up recursion relationship
    for (n in (length(states) - 1):1) {

        # Make integral
        b_state_1 <- integrate(dnorm, mean = input_mean[1], sd = input_sd[1],
            lower = states[n + 1] - 0.005, upper = states[n + 1] + 0.005, abs.tol = 0)$value
        b_state_2 <- integrate(dnorm, mean = input_mean[2], sd = input_sd[2],
            lower = states[n + 1] - 0.005, upper = states[n + 1] + 0.005, abs.tol = 0)$value
        b_states <- c(b_state_1, b_state_2)

        # Calculate beta_hats[n]
        beta_hats[n, ] <- ((beta_hats[n + 1, ] * b_states) %*% t(A))/c_n[n]
    }

    return(list(c_n, alpha_hats, beta_hats))

}

# Function to do non-summed numerator
lambda_func <- function(alpha_hats, A, beta_hats, input_mean, input_sd, n, i,
    j, states) {

    # Do integral
    b_state <- integrate(dnorm, mean = input_mean[j], sd = input_sd[j], lower = states[n +
        1] - 0.005, upper = states[n + 1] + 0.005, abs.tol = 0)$value
    # Return numerator
    states[n] * alpha_hats[n, i] * A[i, j] * b_state * beta_hats[n + 1, j]

}

# Next calculate new mean
lambda_prime_func <- function(alpha_hats, beta_hats, states, A, input_mean,
    input_sd) {

    lambda_prime <- c()

    # Fill B prime matrix - working through the matrix by row then column
```

```
    for (trans_i in 1:length(Mu_0)) {
        lambda_prime_ij <- c()
        for (trans_j in 1:length(Mu_0)) {

            # Calculate numerator using function before, this time summing
            lambda_prime_ij[trans_j] <- sum(sapply(1:(length(states) - 1), lambda_func,
                alpha_hats = alpha_hats, beta_hats = beta_hats, A = A, input_mean = input_mean,
                input_sd = input_sd, i = trans_i, j = trans_j, states = states))
        }

        # Normalise by the summed expected value of E(n_ik)
        lambda_prime[trans_i] <- sum(lambda_prime_ij)/sum(sapply(1:2, exp_n_ij_cont,
            i = trans_i, alpha_hats = alpha_hats, beta_hats = beta_hats, states = states,
            A = A, input_mean = input_mean, input_sd = input_sd))


    }

    return(lambda_prime)
}

# Calculate A prime - first caculate E(n_ij)
exp_n_ij_cont <- function(alpha_hats, beta_hats, states, A, input_mean, input_sd,
    i, j) {

    # Define function to apply across all locations
    func <- function(alpha_hats, A, beta_hats, n, i, j, input_mean, input_sd) {

        # Get integral
        b_state <- integrate(dnorm, mean = input_mean[j], sd = input_sd[j],
            lower = states[n + 1] - 0.005, upper = states[n + 1] + 0.005, abs.tol = 0)$value

        alpha_hats[n, i] * A[i, j] * b_state * beta_hats[n + 1, j]
    }

    # Apply function to all 0 to N-1
    sum(sapply(1:(length(states) - 1), func, alpha_hats = alpha_hats, beta_hats = beta_hats,
        A = A, input_mean = input_mean, input_sd = input_sd, i = i, j = j))

}

# Function to make A prime
A_prime_func_cont <- function(alpha_hats, beta_hats, states, A, input_mean,
    input_sd, Mu_0) {

    # Make A prime matrix
    A_prime <- matrix(nrow = length(Mu_0), ncol = length(Mu_0))

    # Fill A prime matrix
    for (i in 1:length(Mu_0)) {
        for (j in 1:length(Mu_0)) {
            A_prime[i, j] <- exp_n_ij_cont(alpha_hats, beta_hats, states, A,
                input_mean, input_sd, i, j)
        }
    }

    # Normalise A prime matrix by row sums and return it - this poss should be
```

```
    # col? - previously tried with inverse matrix
    t(apply(A_prime, 1, function(x) (x/sum(x))))
}

# Next calculate Mu_prime
Mu_prime_func_cont <- function(alpha_hats, beta_hats, states, A, input_mean,
    input_sd) {

    # Get integral
    b_state_1 <- integrate(dnorm, mean = input_mean[1], sd = input_sd[1], lower = states[2] -
        0.005, upper = states[2] + 0.005, abs.tol = 0)$value
    b_state_2 <- integrate(dnorm, mean = input_mean[2], sd = input_sd[2], lower = states[2] -
        0.005, upper = states[2] + 0.005, abs.tol = 0)$value
    b_states <- c(b_state_1, b_state_2)

    alpha_hats[1, ] * ((b_states * beta_hats[2, ]) %*% t(A))
}

# Now implement the viterbi
viterbi_inference_cont <- function(states, Mu_0, S, V, input_mean, input_sd,
    A) {

    # Set up initial parameters
    N <- length(states)
    phi <- psi <- matrix(nrow = N, ncol = length(Mu_0))

    # Get integral
    b_state_1 <- integrate(dnorm, mean = input_mean[1], sd = input_sd[1], lower = states[1] -
        0.005, upper = states[1] + 0.005, abs.tol = 0)$value
    b_state_2 <- integrate(dnorm, mean = input_mean[2], sd = input_sd[2], lower = states[1] -
        0.005, upper = states[1] + 0.005, abs.tol = 0)$value
    b_states <- c(b_state_1, b_state_2)

    phi[1, ] <- log(Mu_0) + log(b_states)

    # Run recursion relationship
    for (n in 2:N) {

        # Recursion of phi ****is this a mistake?****
        to.max <- log(A) + phi[n - 1, ]

        # Get integral
        b_state_1 <- integrate(dnorm, mean = input_mean[1], sd = input_sd[1],
            lower = states[n] - 0.005, upper = states[n] + 0.005, abs.tol = 0)$value
        b_state_2 <- integrate(dnorm, mean = input_mean[2], sd = input_sd[2],
            lower = states[n] - 0.005, upper = states[n] + 0.005, abs.tol = 0)$value
        b_states <- c(b_state_1, b_state_2)

        # Do maximisation and add to other term
        phi[n, ] <- log(b_states) + apply(to.max, 2, max)

        # Recursion of psi - do argmax
        psi[n, ] <- apply(to.max, 2, function(x) (which(x == max(x))))
    }

    ## Do traceback
    i_star <- rep(NA, times = N)
```

```
    # Get Nth value
    i_star[N] <- which(phi[N, ] == max(phi[N, ]))

    # Trace back along vector
    for (n in (N - 1):1) {
        i_star[n] <- psi[n + 1, i_star[n + 1]]
    }

    return(i_star)
}

# Arbitrarily set the mean and sd
input_sd <- c(sd(GC_window_content), sd(GC_window_content) - 0.01)
input_mean <- c(mean(GC_window_content) - 0.02, mean(GC_window_content) + 0.02)

# Get updated theta and infer the variables
updated_cont_theta <- baum_welch_cont(GC_window_content, S, V, A, Mu_0, input_mean,
    input_sd, 1e-04, 10000)
A_prime <- updated_cont_theta[[1]]
lambda_prime <- updated_cont_theta[[2]]
sigma_prime <- updated_cont_theta[[3]]
Mu_prime <- updated_cont_theta[[4]]
continuous_inferred_states <- viterbi_inference_cont(GC_window_content, Mu_prime,
    S, V, lambda_prime, sigma_prime, A_prime)

# Make a plot (woo)
ggplot() + geom_line(aes(x = 1:length(GC_window_content), y = GC_window_content,
    col = "blue")) + geom_line(aes(x = 1:length(continuous_inferred_states),
    y = continuous_inferred_states/10, col = "red")) + labs(title = "Using a continouus distribution
    theme_minimal() + theme(legend.position = "none") + xlab("Window position") +
    ylab("Emission State (top), hidden state (bottom)")
```