GSA Assignment: Markov Chains and Hidden Markov Models

USN: 303039534

1.

A function *writemats()* has been written which takes a parameter K and writes a random KxK transition matrix and 1xK initial distribution subject to normalization. A further function *MC()* takes the filenames of two such matrices together with a chain length N, and generates N elements of a Markov Chain with the specified transition matrix and initial distribution. Code for both of these functions as well as all other functions written for this assignment can be can be found in the appendix.

As an example, we generate a random transition matrix and initial distribution with K = 3 and obtain the following parameters:

$$A = \begin{bmatrix} 0.544 & 0.239 & 0.216 \\ 0.323 & 0.216 & 0.461 \\ 0.082 & 0.431 & 0.487 \end{bmatrix}$$

$$\mu^0 = \begin{bmatrix} 0.792 \\ 0.100 \\ 0.108 \end{bmatrix}$$

Using these parameters, we generate a Markov Chain of length 1000 and plot the first 100 elements in figure 1.

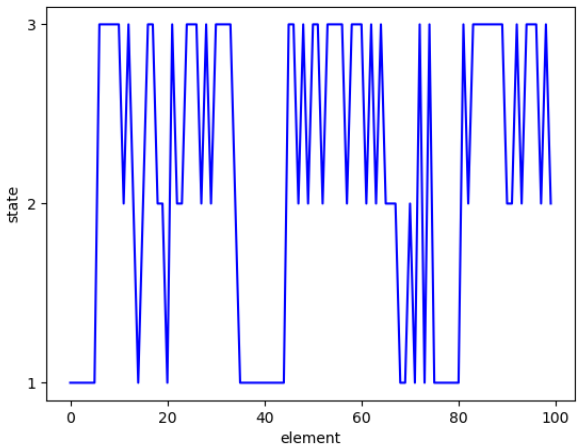

Figure 1: First 100 elements of random Markov chain

We find that the model spends 32.1%, 29.1% and 38.8% of the time in states 1, 2 and 3 respectively. We can also calculate the hypothetical equilibrium distribution, π, of the chain as the distribution for which

$$\pi^T \mathrm{A} = \pi^T \tag{1}$$

We find this as the eigenvector of A^T corresponding to an eigenvalue of 1. This equilibrium distribution is

$$\pi = \begin{bmatrix} 0.291 \\ 0.309 \\ 0.400 \end{bmatrix}$$

We thus see that our simulated distribution of states is somewhat similar to the equilibrium distribution, but with a bias towards state 1 from starting in state 1 with a high A_{11} value.

2.

For a Markov Chain with transition matrix A and initial distribution μ_0, the probability of observing a given sequence of state indices x_0^N is

$$P(x_0^N) = \mu_0(x_0) * \prod_{i,j} A_{ij}^{n_{ij}} \tag{2}$$

Where n_{ij} is the number of transitions from state i to state j.

We can calculate the maximum likelihood Markov chain transition matrix and initial distribution by maximizing the log of this probability subject to the constraint that each row of the transition matrix must add to 1 since the probability of going from any state i to some state is 1. We do this using Lagrange multipliers:

$$\mathcal{L} = \mu_0(x_0) + \sum_{i,j} n_{ij} * \log A_{ij} - \sum_i \lambda_i * (\sum_j A_{ij} - 1) \tag{3}$$

$$\frac{\partial \mathcal{L}}{\partial A_{ij}} = 0 \tag{4}$$

This gives

$$\frac{n_{ij}}{A_{ij}} = \lambda_i = \sum_j n_{ij} \tag{5}$$

From the constraint equations. We thus estimate the most likely transition matrix elements given our sequence as

$$A_{ij} = \frac{n_{ij}}{\sum_j n_{ij}} \tag{6}$$

where n_{ij} is the number of transitions from state i to state j. the Maximum likelihood transition probability from state i to state j is thus the proportion of total transitions from state i that go to state j.

We also have

$$\frac{\partial \mathcal{L}}{\partial \mu_0(i)} = 0 \tag{7}$$

This gives $\mu_0(i) = \delta(i, x_0)$. Given a single Markov chain where we only have a single data point from which to infer an initial distribution, the initial distribution is thus 1 for whichever state occurs first and zero for all other states.

A function *MLE()* has been written that performs a maximum likelihood estimation given a state space and a set of Markov chain state indices. As an example, we perform a maximum likelihood estimation on the chain generated in part 1 above and arrive at the following transition matrix and initial distribution:

$$A_{MLE} = \begin{bmatrix} 0.609 & 0.191 & 0.200 \\ 0.341 & 0.244 & 0.415 \\ 0.086 & 0.447 & 0.477 \end{bmatrix}$$

$$\mu^0_{MLE} = \begin{bmatrix} 1.000 \\ 0.000 \\ 0.000 \end{bmatrix}$$

As expected, the initial distribution is 1 for state 1 and zero for states 2 and 3 since our chain starts in state 1 (figure 1). A better estimate could be obtained by generating multiple chains and averaging the initial distribution ocer these. The estimated transition matrix looks similar to the real transition matrix from part 1., albeit with A_{11} being somewhat higher than expected. Such errors are due to the inherently stochastic nature of the chain and could be mitigated by performing maximum likelihood estimation on a longer chain or averaging over multiple chains.

3.

We proceed to implement a function *HMC()* which generates a Markov chain of length N using *MC()* and uses this as a hidden chain to generate an observed chain given an emission matrix **B**. We then use *HMC()* to generate a chain of 115 hidden and emitted states using the parameters given in the assignment (θ_{ass}) and plot these in figure 2.

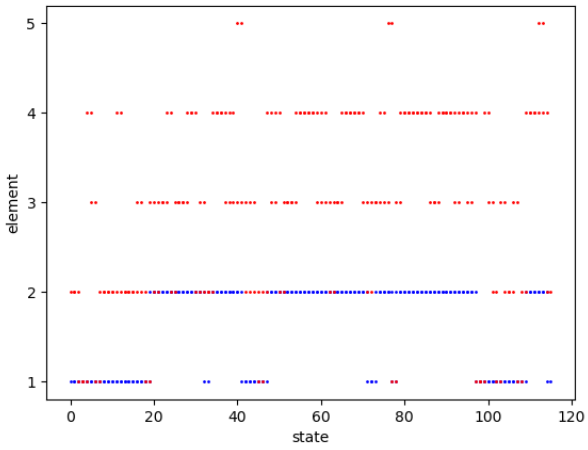

Figure 2: 115 hidden (blue) and emitted (red) states for a Markov chain generated with the parameters given in the assignment (θ_{ass}).

θ_{ass} is defined by

$$S_{ass} = 0, 1, V_{ass} = 1, 2, 3, 4, 5 \tag{8}$$

$$\mathrm{A}_{ass} = \begin{bmatrix} 0.8 & 0.2 \\ 0.1 & 0.9 \end{bmatrix} \mathrm{B}_{ass} = \begin{bmatrix} 0.2 & 0.5 & 0.2 & 0.1 & 0.0 \\ 0.0 & 0.1 & 0.4 & 0.4 & 0.1 \end{bmatrix} \mu_{0,ass} = [0.5\ 0.5] \tag{9}$$

We see in figure 2 that there is a bias towards hidden state 2 over hidden state 1 as we expect from inspection of A_{ass}. We also see that hidden state 1 leads to a higher proportion of emitted states 1 and 2 whereas hidden state 2 leads to a higher proportion of emitted states 3, 4 and 5 as expected from inspection of B_{ass}.

4.

Writing the emitted sequence from part 3. to a file emitted.txt, we further implement the scaled forward algorithm in the function *forward()* to calculate the probability of observing this sequence y_0^N given θ_{ass}.

We first define the parameters $\{\alpha_n(i)\}$ as

$$\alpha_n(i) = P(y_0^n, x_n = i) = \sum_j \alpha_{n-1}(j) * B_i(V_n) * A_{ji} \tag{10}$$

This recursion is initialized by

$$\alpha_0(i) = \mu_0(i) * B_i(V_0) \tag{11}$$

After calculating α at every step of the chain, we can proceed to calculate the probability of observing a sequence of emitted states as

$$P(y_0^N) = \sum_i (P(y_0^N, x_N = i)) = \sum_i \alpha_N(i) \tag{12}$$

Since α_n becomes progressively smaller with n, underflow can become a problem and we therefore implement a scaled version of the forward algorithm where we normalize each set of α values using normalization constants $c_n = P(y_n | y_0^{n-1})$ such that

$$\hat{\alpha}_n(i) = \frac{\alpha_n(i)}{\prod_{k=0}^n c_k} = \frac{\alpha_n(i)}{\sum_j \alpha_n(j)} \tag{13}$$

This prevents underflow and allows us to calculate the total log likelihood of observing y_0^N as

$$\log P = \log \prod_n c_n = \sum_n \log c_n \tag{14}$$

For the chain in figure 2 we obtain log10p = -69.545 and thus p = 2.851e-70. In the present case, the chain is sufficiently small that underflow is not an issue in Julia, and we thus validate our implementation using the un-scaled algorithm which gives the same probability.

We also implement the backward algorithm for future use in the Baum-Welch algorithm. Here we define

$$\beta_n(i) = P(y_{n+1}^N, x_n = i) = \sum_j \beta_{n+1}(j) * B_j(V_{n+1}) * A_{ij} \tag{15}$$

This recursion is initialized by

$$\beta_N(i) = 1 \tag{16}$$

To prevent underflow, we use the same scaling factors as in the forward algorithm such that

$$\hat{\beta}_n(i) = \frac{\beta_n(i)}{\prod_{k=n+1}^N c_k} \tag{17}$$

We also use the β values from the backward algorithm to check that for all n

$$\sum_i \alpha_n(i) * \beta_n(i) = \sum_i P(Y_0^n, x_n = i|\Theta) * P(Y_(n+1)^N|x_n = i, \Theta) = \sum_i P(Y_0^N, x_n = i|\Theta) = 1 \tag{18}$$

This further validates the present implementation of the forward-backward algorithm.

5.

We download the *S. cerevisiae* chromosome III from Ensembl and quantify the GC content in 100bp windows using the functions *get_brackets()* and *calc_GC()*.

To convert the calculated GC content to emission states from the model in part 3, we need to generate a binning scheme with five bins. We therefore generate a Hidden Markov Chain from θ_{ass} with 100,000 elements and calculate the frequency of each state [1:5]. The cumulative frequencies are 0.065, 0.297, 0.631, 0.932 and 1.0. We then plot a cumulative distribution of the GC content in the yeast chromosome (figure 3).

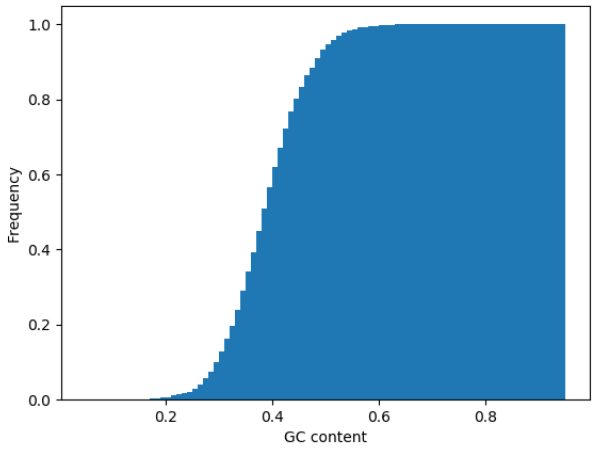

Figure 3: Cumulative distribution of GC content

By matching these two cumulative distributions, we arrive at the following upper bounds of GC content in a window to correspond to each state 1-5: [0.275, 0.345, 0.405, 0.495, 1.0].

We feed these parameters to the function *GC_to_state()* together with the sequence of 3167 calculated GC contents for the 3167 windows and return a sequence of 3167 observed states $s_n \in [1:5]$.

We use this sequence of emitted states as an input to *forward()* together with θ_{ass} to calculate the probability of observing this particular sequence. In this case underflow does become a problem, and we therefore use the scaled algorithm to calculate the probability which gives a log likelihood of log10(P) = -1864.385.

6.

In the Baum-Welch algorithm, we start with a set of initial parameters and define

$$\gamma_n(i) = P(x_n = i|y_0^N) = \frac{P(x_n = i, y_0^N)}{P(y_0^N)} = \frac{\alpha_n(i) * \beta_n(i)}{\sum_j \alpha_n(j) * \beta_n(j)} \tag{19}$$

$$\xi_n(i,j) = P(x_n = i, x_{n+1} = j|y_0^N) = \frac{\alpha_n(i) * A_{ij} * \beta_{n+1}(j) * B_j(V_{n+1})}{\sum_i \sum_j \alpha_n(i) * A_{ij} * \beta_{n+1}(j) * B_j(V_{n+1})} \tag{20}$$

After calculating $\{\alpha_n\}$ and $\{\beta_n\}$ for each step in the chain of emitted states using the forward algorithm with θ_{init}, we also calculate all $\{\gamma_n\}$ and $\{\xi_n\}$. Using these quantities, we can calculate expected transition frequencies and thus expected parameters $\tilde{\text{A}}$, $\tilde{\text{B}}$ and $\tilde{\mu_0}$:

$$\tilde{\mu}_0(i) = \gamma_1(i) \tag{21}$$

$$\tilde{\text{A}}_{ij} = \frac{\sum_{n=0}^{N-1} \xi_n(i,j)}{\sum_{n=0}^{N-1} \gamma_n(i)} \tag{22}$$

$$\tilde{\text{B}}_i(V_k) = \frac{\sum_{n=0}^{N} \delta(v_k, v_n) * \gamma_n(i)}{\sum_{n=0}^{N} \gamma_n(i)} \tag{23}$$

Where δ is the delta function.

$\tilde{\mu}_0$, $\tilde{\text{A}}$ and $\tilde{\text{B}}$ will generally be different from the initial parameters A, B and μ_0 used to estimate $\{\alpha_n\}$, $\{\beta_n\}$, $\{\gamma_n\}$ and $\{\xi_n\}$. We can therefore calculate new values for these quantities, which in turn give us a new set of model parameters etc. By iteratively updating the model parameters until they remain stationary, we find a local maximum in parameter space and thus a set of new optimum parameters for the given model and emitted sequence.

We implement the Baum-Welch algorithm in the function *baum_welch()* with a default convergence threshold of 0.0001 for the 2-norm of a vector of the change in all model parameters. That is, we measure convergence by the parameters becoming stationary rather than the log likelihood becoming stationary, but these are of course related.

To update the parameters for the model in part 3, we use θ_{ass} as initial parameters. After running the Baum-Welch algorithm, we arrive at an optimum log10p = -1821.103 for the emission sequence in part 5. That is, after running the Baum-Welch algorithm, we obtain a set of parameters that makes this particular emitted sequence 43 orders of magnitude more likely than it was with our initial parameters. The optimum parameters are:

$$A = \begin{bmatrix} 0.939 & 0.061 \\ 0.120 & 0.880 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.085 & 0.334 & 0.399 & 0.181 & 0.000 \\ 0.000 & 0.032 & 0.197 & 0.567 & 0.204 \end{bmatrix}$$

$$\mu_0 = \begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix}$$

However, the Baum-Welch algorithm can be sensitive to initial parameters, and we therefore validate our result by generating 100 random sets of initial parameters for the algorithm with the function *test_welch()*. This gives an optimum log10p of -1818.783, which is slightly better than what we observed using θ_{ass} as initial parameters. The optimum parameters calculated by *test_welch* are:

$$A = \begin{bmatrix} 0.954 & 0.046 \\ 0.083 & 0.917 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.082 & 0.335 & 0.400 & 0.182 & 0.001 \\ 0.011 & 0.047 & 0.206 & 0.544 & 0.191 \end{bmatrix}$$

$$\mu_0 = \begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix}$$

A likely reason for the discrepancy is that with B_{ass}, we force B1(5) and B2(1) to be zero, thus decreasing the number of degrees of freedom available for optimization. By relaxing this constraint, we arrive at both of these parameters being slightly higher than zero, and this results in a slightly better fit.

When inspecting the list of 100 log10p values converged to, we observe that all of our simulations converge to one of two local maxima at logp = -1818.78 and logp = -1946.23. This illustrates the potentially large effect of the choice of initial conditions and thus the importance of verifying results with different initial conditions. Even after running 100 simulations, we do not know for sure if we have in fact found the global maximum or merely local maxima that are commonly converged to in a Baum-Welch framework. One way to investigate this further would be to implement a metadynamics-like algorithm where we prevent reconvergence to maxima already found, but that is beyond the scope of the present assignment

We can also alter the binning procedure from the one in part 5 where we match the state distribution expected from θ_{ass}. Another straightforward approach would be to bin GC content such that states 1-5 become equally frequent in the yeast chromosome. Using this binning procedure and running *test_welch()* leads to a best fit with optimum log10p = -2070.062 from 100 simulations, which is significantly different from what we observed with the previous binning scheme. This suggests that the choice of binning can strongly affect the result, calling for a potential optimization of the binning scheme. However, this is a 4-parameter optimization that is beyond the scope of the present assignment (although it could be implemented relatively simply using steepest descent with empirical gradients or quasi-newton optimx ization with empirical gradient and hessian if the log10p landscape is well-behaved).

We could also achieve better fits by altering the number of free parameters further and allowing for e.g. more than 2 hidden states and more than 5 observed states. This is a rather difficult hyperparameter optimization problem which balances overfitting and predictive power and is beyond the scope of the present assignment. However, it is an interesting and important problem both in the field of Hidden Markov Models and in machine learning more generally.

7.

To find the most likely sequence of hidden states (the Viterbi path) given an emitted sequence and a set of parameters, we implement the Viterbi algorithm in the function *viterbi()*. In the viterbi algorithm, we define

$$\omega_{n,i} = \log P(y_0^N, x_0^N) = \log P(y_n|x_n = i) + \max_k [\omega_{n-1,k} + \log P(x_n = i|x_{n-1} = k)] \tag{24}$$

This recursion is initialized by

$$\omega_{1,i} = \log \mu_0(i) + \log B_i(V_0) \tag{25}$$

We also store for every element n and state i which previous state lead to ω_{ni}, such that

$$\text{traceback}_{n,i} = \text{argmax}_k[\log P(y_n|i) + \omega_{n-1,k} + \log P(i|k)] \tag{26}$$

After calculating all $\{\omega_{n,i}\}$, we can find the Viterbi path using traceback

$$\text{path}[N] = \text{argmax}_i(\omega_{N,i}) \tag{27}$$

$$\text{path}[n] = \text{traceback}[n+1, \text{path}[n+1]] \tag{28}$$

We proceed to find the Viterbi path using the optimized parameters from the Baum-Welch optimization with initial parameters θ_{ass}. This results in a Viterbi path with a log likelihood of -1896.813 for the most likely sequence of hidden states.

We plot the Viterbi path together with the actual GC content for 100 windows along the yest chromosome in figure 4. In this figure, the GC content has been scaled to a range of [0:3] for plotting together with the hidden Markov states. This corresponds to a real range of GC contents of [0.11:0.63]. As also observed in figure 2 we see that state 1 generally corresponds to lower GC contents and state 2 to higher GC contents. We also see that there are generally longer stretches of state 1 and state 2 rather than rapid oscillations between the two. This is a result of the transition matrix having large diagonal elements.

Utility: e.g. euchromatin vs. heterochromatin. Transcriptionally active vs passive. ORF finding.

gene finding; higher GC content due to evolutionary constraints?

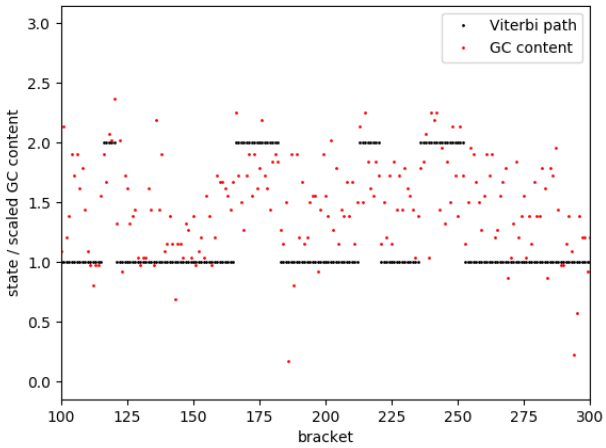

Figure 4: Viterbi path (black) and GC content scaled to a range of 0 to 3 for position 10,000 to 30,000 of *S. cerevisiae* chromosome III

Appendix (code)

File: gsa.jl

```
###code for GSA

using Random, PyCall, PyPlot, LinearAlgebra, DelimitedFiles, StatsBase
include("forward.jl")
include("viterbi.jl")
include("markov_chains.jl")
include("anal_GC.jl")

    #1.
#Kmat, Kvec = writemats(K = 3)
#chain = MC( N = 1000, Plot=true ) #generate example chain

    #2.
#A, u0 = MLE(chain, 1:3) #infer parameters (have to repeat this to get u0)

    #3.
#construct model from assignment
A = [0.8 0.2; 0.1 0.9]; writedlm("A.txt", A)
u0 = [0.5 ; 0.5]; writedlm("u0.txt", u0)
B = [0.2 0.5 0.2 0.1 0; 0 0.1 0.4 0.4 0.1]; writedlm("B.txt", B)
#plot_HMC() #generate and plot chain for hidden markov model

    #4.
#use forward algorithm to get probability of emitted sequence
#fwdh, bwdh, cs, log10p = forward(emitted = "emitted.txt"))

    #5.
bs = get_brackets() #load yeast genome and divide into 100 bp windows
seq = calc_GC(bs) #calculate GC content of bins

#plot cumulative distribution
#PyPlot.plt[:hist](seq, bins = 0.05:0.01:0.95, cumulative = true, density=true)
    #xlabel("GC content")
    #ylabel("Frequency")
    #savefig("cumulative_GC.png")
#close()

#state_seq = GC_to_state(seq) #infer discrete states 1:5
#writedlm("GC_seq.txt", state_seq); writedlm("GC_percent,txt", seq)
#get log likelihood of GC sequence
#fwdh, bwdh, cs, log10p = forward(emitted = "GC_seq.txt"))

    #6.
```

```
#Use Baum-Welch to infer parameters with initial parameters from assignment
#A, B, u0, logp = baum_welch(emitted = "GC_seq.txt")
#use uniform binning
#Aunif, Bunif, u0unif, logpunif = baum_welch(emitted = "GC_seq_unif.txt")

#test 100 different initial parameters and find best possible parameters
#logps, Am, Bm, um = test_welch()
#use uniform binning
#logps, Am, Bm, um = test_welch(emitted = "GC_seq_unif.txt")

    #7.
#p, o = viterbi() #test viterbi for N=115 simulated HMM
#generate Viterbi path for Baum-Welch-inferred parameters
#path, opt = viterbi(emitted = "GC_seq.txt", A = "Abw.txt", B="Bbw.txt", u0="ubw.txt")
#writedlm("viterbi_path_bw.txt", path)

#plot_GCs() #plot viterbi path together with GC content
```

File: markov_chains.jl

```
using Random, PyCall, PyPlot, LinearAlgebra, DelimitedFiles, StatsBase

function writemats(;K=3, filename = "K")
    Kmat = rand(K,K) #normalize rows!
    Kmat ./= sum(Kmat,dims = 2)
    writedlm(filename*"mat.txt", Kmat)

    Kvec = rand(K)
    Kvec = Kvec ./ sum(Kvec)
    writedlm(filename*"vec.txt", Kvec)
    return Kmat, Kvec
end

#writemats()

function MC(;N=100, A = "Kmat.txt", u0 = "Kvec.txt", Plot = false)
    A = readdlm(A)
    u0 = readdlm(u0)
    S = 1:length(u0)
    chain = zeros(N)
    chain[1] = sample(S, aweights(u0))
    for i in 2:N
        chain[i] = sample(S, aweights(A[Int(chain[i-1]),:]))
    end
    if Plot
        plot(chain[1:100], "b-")
        yticks(1:size(A)[1])
        xlabel("element")
        ylabel("state")
        savefig("MC.png")
        close()
    end
    writedlm("chain1000.txt", chain)
    return chain
end

function MLE(seq, states)
    #sequence is a sequence of state indices
    #states is a set of possible state indices

    N = length(states)
    A = zeros(N, N)

    for ind in 2:length(seq)
        A[Int(seq[Int(ind-1)]), Int(seq[Int(ind)])] += 1
    end

    println("N: ", length(seq), " transitions: ", sum(A))
    A ./= sum(A, dims = 2)

    #Aij = nij / sum_i[nij]
    #where nij is #transitions from i to j
    u0 = zeros(N)
```

```
    u0[Int(seq[1])] = 1 # we only have a single data point...
    return A, u0
end

function HMC(;N=115, A = "A.txt", B = "B.txt", u0 = "u0.txt")
    hidden = MC(N=N, A=A, u0=u0)
    B = readdlm(B)
    V = 1:size(B)[2] #number of rows
    observed = zeros(N)
    for i in 1:N
        observed[i] = sample(V, aweights(B[Int(hidden[i]),:]))
    end
    return hidden, observed

end

function plot_HMC()
    h, o = HMC()
    writedlm("emitted.txt", o)
    L = length(h)
    xs = vcat(0, repeat(1:(L-1), inner=2), L)
    figure()
    psize = 1
    plot(xs, repeat(h, inner=2), "bo", MarkerSize=psize)
    plot(xs, repeat(o, inner=2), "ro", MarkerSize=psize)
    #legend(["Hidden", "Observed"])
    yticks(1:5, 1:5)
    xlabel("state")
    ylabel("element")
    savefig("ho.png")
    #show()
    close("all")
end
```

File: anal_GC.jl

```
using Random, PyCall, PyPlot, LinearAlgebra, DelimitedFiles, StatsBase

function get_brackets(;file = "S.cerevisiae.III.fa")
    #calculate GC content of file
    #convert to number 1:5 by binning for emission
    #character count is multiple of 60
    f = open(file)
    lines = readlines(f)
    genome = ""
    for line in lines
        genome = genome*line
    end
    println(length(genome), " bases" )
    n = 1
    brackets = []
    while n <= length(genome)
        brackets = vcat(brackets,
                        genome[n:min((n+99), length(genome))])
        n += 100
    end
    close(file)
    return brackets
end

function calc_GC(brackets)
    seq = zeros(length(brackets))
    for i in 1:length(seq)
        seq[i] = length( collect( #search for any of G, C, g, c
                eachmatch(r"[GCgc]", brackets[i], overlap=false) ) ) /
                length(brackets[i])
    end
    return seq
end

function anal_GC(;N=100000)
#for binning purposes, we aim to match the probability
#of getting a particular observable in the model with the probability
```

```
#of getting a particular GC content in the chromosome
#i.e. match cumulative distributions
#consider if we should just do 20% each so we can make a good HMM whenw e optimize it
 h, o = HMC(N=N)
 cum = 0
 for i in 1:5
     cum += sum(o .== i)/N
     println(i, ": ", sum(o .== i)/N, "  ", cum )
 end
 bins = [0.275, 0.345, 0.405, 0.495, 1.0]
 #bins_eq = [0.325, 0.365, 0.405, 0.445, 1.0]
 for i in 1:5
     println(sum( seq .<= bins[i] )/length(seq))
 end
end

function GC_to_state(seq, bins = [0.275, 0.345, 0.405, 0.495, 1.0])
    #convert from GC content to state 1:5
    #for use with HMM
    N = length(seq)
    state_seq = zeros(N)
    for i in 1:N
        for j = 1:5
            if seq[i] <= bins[j]
                state_seq[i] = j
                break
            end
        end
    end
    return state_seq
end

function plot_GCs(;viterbi = "viterbi_path_bw.txt", GC_obs = "GC_seq.txt",
                    GC_percent = "GC_percent.txt")

    Range = [100, 300]

    viterbi = readdlm(viterbi)
    GC_obs = readdlm(GC_obs)
    GC_percent = readdlm(GC_percent)
    min, max = minimum(GC_percent), maximum(GC_percent)
    GC_percent = (GC_percent .- min) * 3/(max-min)

    figure()
    psize = 1
    plot(viterbi, "ko", MarkerSize = psize)
    #plot(GC_obs, "bo", MarkerSize = psize)
    plot(GC_percent, "ro", MarkerSize = psize)
    xlim(Range)
    xlabel("bracket")
    ylabel("state / scaled GC content")
    legend(["Viterbi path", "GC content"])
    savefig("viterbi_path.png")
    close()

end
```

File: forward.jl

```
#implement forward, backwards and Baum-Welch algorithms

using Random, PyCall, PyPlot, LinearAlgebra, DelimitedFiles
using StatsBase

function get_mats(emitted, A, B, u0)
    #if provided as strings, returns as matrices
    if (typeof(A) == String) A = readdlm(A) end
    if (typeof(B) == String) B = readdlm(B) end
    if (typeof(u0) == String) u0 = readdlm(u0) end
    if (typeof(emitted) == String) emitted = readdlm(emitted) end
    return emitted, A, B, u0
end
```

```
function forward(;emitted = "emitted.txt", A="A.txt", B="B.txt",
                 u0 = "u0.txt", Print = false)

    #load matrices if provided as files
    emitted, A, B, u0 = get_mats(emitted, A, B, u0)

    Nh, No = size(B) #number of hidden and observed states
    states_h = 1:Nh ; states_o = 1:No #state spaces
    L = size(emitted)[1]

    fwdh = zeros(length(emitted), Nh) #create array of alpha values
    #rows are steps in the chain, columns are hidden states
    cs = zeros(length(emitted)) #store scaling factors
    alphah_prev = zeros(Nh) #scaled
    #initialize alpha_0(j) as u0(j)*B[j, V0]

    c1 = sum( u0 .* B[:, Int(emitted[1])] ); cs[1] = c1 #scaling factor
    for state in states_h
        alphah_prev[Int(state)] = B[Int(state), Int(emitted[1])] * u0[Int(state)] / c1
    end
    fwdh[1,:] = alphah_prev #store scaled values

    for (n, obs_n) in enumerate(emitted[2:end])
        #for each observed element in the chain
        deltas = zeros(Nh) #delta_ni = cn*alphah_ni = B[i, Vn]*sum_j[ alphah_n-1(j) * A[j, i] ]

        for state in states_h
            deltas[state] = B[Int(state), Int(obs_n)] .*
                            sum(alphah_prev .* A[:,Int(state)])
        end

        cn = sum(deltas) #sum_i(alphah_ni * cn) = cn * 1 since alpha_h normalized
        alphah_curr = deltas ./ cn #normalized

        fwdh[n+1,:] = alphah_curr #store alpha hat
        alphah_prev = alphah_curr #store alphah
        cs[n+1] = cn #store scaling factors
        Print && println("new_n:_", n+1, "__alpha:_", alphah_curr) #print result
    end

    #p = sum(alpha_curr)  #total p is sum of final alphas
    log10p = sum( log10.( cs ) ) #P = product_n(cn)
    Print && println("probability_is_", 10^log10p, "__log10:_", log10p)


  # backward part of the algorithm

    betah_prev = repeat([1], Nh)
    bwdh = zeros(length(emitted), Nh) #create array of alpha values
    bwdh[1,:] = betah_prev

    for (n, obs_Lm) in enumerate(reverse(emitted[2:end]))

        epsilons = zeros(Nh) #epsilon_ni = cn+1*betah_ni = sum_j[ B[j, Vn+1]* betah_n+1(j) * A[i, j] ]

        for state in states_h
            epsilons[state] = sum( B[:, Int(obs_Lm)] .*
                                betah_prev .* A[Int(state),:] )
        end

        betah_curr = epsilons ./ cs[Int(L-n+1)] #normalized

        bwdh[n+1,:] = betah_curr #store beta hat
        betah_prev = betah_curr #store betah
        Print && println("new_n:_", L-n, "__betah:_", betah_curr) #print result
    end

    bwdh = reverse(bwdh, dims=1)
    Print && println( sum(fwdh .* bwdh, dims=2) ) #P/P = 1 for all n
    #println(sum(fwdh .* bwdh) )
    #println(sum(fwdh .* bwdh, dims=2)[1:10])
```

```
    return fwdh, bwdh, cs, log10p
end


function estimate_params(emitted, fwdh, bwdh, cs, A0, B0; Print = false)
    #given alphas, betas and emitted sequence, calculate expected A, B, u0
    #everything is ratios of products of alphas and betas
    #so the scaling cancels
    #(alpha_i)*(beta_i) scaled by prod(cs)

    L = size(fwdh)[1]; Nh = size(fwdh)[2]; No = size(B0)[2]
    A1 = copy(A0); B1 = copy(B0)

    #gamma_ni = alpha_n(i)*beta_n(i) / sum_i(alpha_n(i) * beta_n(i))
    gammas = fwdh .* bwdh ./ sum(fwdh .* bwdh, dims = 2)

    xis = zeros((L-1), Nh, Nh) #initialize array

    for n in 1:(L-1)
        #transitions from n to n+1
        for i in 1:Nh
            for j in 1:Nh
                xis[n, i, j] =
                    fwdh[n, i] * A0[i,j] * bwdh[n+1, j] * B0[j, Int(emitted[n+1])]

            end
        end
        xis[n, :, :] /= sum(xis[n, :, :]) #normalize xis
        Print && println(xis[n, :, :])
    end

    #println("validation")
    #println(gammas[1:10, :], "\n", sum(xis, dims=3)[1:10,:])
    #println(sum(emitted .== 1), " ", sum(emitted .== 2))
    #println("size: ", size(gammas), "  ", sum(gammas[1:(end-1), :], dims=1))
    #println("size  ", size(xis), "  ", sum(xis, dims = [1,3]))

    u0 = gammas[1,:] #get initial distribution
    for i in 1:Nh
        for j in 1:Nh
            #update transition matrix as P(xn=i, xn+1=j) / P(xn=i)
            A1[i, j] = sum( xis[:, i, j] ) / sum( gammas[1:(end-1),i] )
        end
        for k in 1:No
            #update emission matrix as p(xn=i, bn=k) / p(xn=i)
            B1[i, k] = sum( gammas[reshape(emitted .== k, L), i] ) / sum(gammas[:,i])
        end
    end
    #println(sum(A1, dims=2))
    #A1 = A1 ./ sum(A1, dims=2) #prevent drift in normalization from iterations and long chain

    return A1, B1, u0
end

function baum_welch(;emitted = "emitted.txt", A="A.txt", B="B.txt",
                u0 = "u0.txt", thresh = 0.0001, maxiter = 10000)
    #

    #println("\n\nstarting new Baum Welch\n")
    emitted, A_0, B_0, u0_0 = get_mats(emitted, A, B, u0)

    error = thresh+1
    niter = 0

    println(A_0)

    while (error > thresh) & (niter < maxiter)
        niter += 1
        #println("A0: ", A_0)
        fwdh, bwdh, cs, logp = forward(emitted=emitted, A=A_0, B=B_0, u0=u0_0)
        A_1, B_1, u0_1 = estimate_params(emitted, fwdh, bwdh, cs, A_0, B_0)
        error = norm( hcat(A_1, B_1, u0_1) - hcat(A_0, B_0, u0_0) )
```

```
        println("n: ", niter, "  error: ", error, "  logp: ", logp, "  u0_1")
        #println(A_0, "    ", A_1)
        #println(B_0, "    ", B_1)
        #println(u0_0, "    ", u0_1)
        A_0, B_0, u0_0 = copy(A_1), copy(B_1), copy(u0_1)
    end

    println(A_0)

    fwdh, bwdh, cs, logp = forward(emitted=emitted, A=A_0, B=B_0, u0=u0_0)
    println("Final log10 Probability: ", logp)

    return A_0, B_0, u0_0, logp

end

function test_welch(;N = 100, emitted = "GC_seq.txt")

    logps = zeros(N)
    logp_max = -Inf
    Am, Bm, um = 0, 0, 0

    for i in 1:N
        writemats(K = 2, filename = "test")
        Btest = rand(2,5) #normalize rows!
        Btest ./= sum(Btest,dims = 2)

        A, B, u0, logp = baum_welch(A = "testmat.txt", B = Btest,
                        u0 = "testvec.txt", emitted = emitted)

        logps[i] = logp

        if logp > logp_max
            println("New max logp: ", logp)
            Am, Bm, um = A, B, u0
            logp_max = logp
        end
    end

    return logps, Am, Bm, um
end
```

File: viterbi.jl

```
using Random, PyCall, PyPlot, LinearAlgebra, DelimitedFiles
using StatsBase

function viterbi(;emitted = "emitted.txt", A="A.txt", B="B.txt",
                u0 = "u0.txt")


    emitted, A, B, u0 = get_mats(emitted, A, B, u0)

    L = length(emitted); Nh, No = size(B)
    V = zeros(L, Nh)
    traceback = zeros(L, Nh); traceback[1,:] = zeros(Nh)


    omegas = zeros(L, Nh)
    traceback = zeros(L, Nh); traceback[1,:] = zeros(Nh)
    omegas[1,:] = log10.(u0) .+ log10.(B[:, Int(emitted[1])])
    for n = 2:L
        #calculate probabilities of possible transitions
        #println(size(log10.(B[:, Int(emitted[n])])'), " ", size(omegas[n-1, :]), " ", size(log10.(A)))
        params = (log10.(A) .+ log10.(B[:, Int(emitted[n])])' ).+ omegas[n-1,:]
        #println(params)
        a, b = findmax(params, dims=1) #find optimum step to each state
        omegas[n,:] = a #store new probability
        traceback[n,:] = [b[1][1], b[2][1]] #store state from which we came
    end

    for n in 1:L println(omegas[n,:],"  ",traceback[n, :]) end
```

```
    path = zeros(L)
    # The highest probability
    opt, ind = findmax(omegas[end,:])
    path[L] = ind
    for n in reverse(1:(L-1))
        path[n] = traceback[(n+1), Int(path[n+1])]
    end

    println("Most probable path:\n", ind)
    for state in path println(state) end
    println("max prob: ", opt)

    return path, opt
end
```