COMP 5970/6970-004
Computational Biology: Genomics and Transcriptomics
Lecture notes 12: 2/22/2022

Haynes Heaton

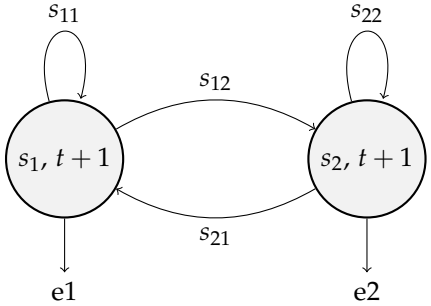
Spring, 2022

Lecture Objectives

- Finite State Automata
- Hiddem Markov models via FSAs
- Alignment as FSA
- Probabilistic alignment with pair-HMM

Finite State Automata (FSA)

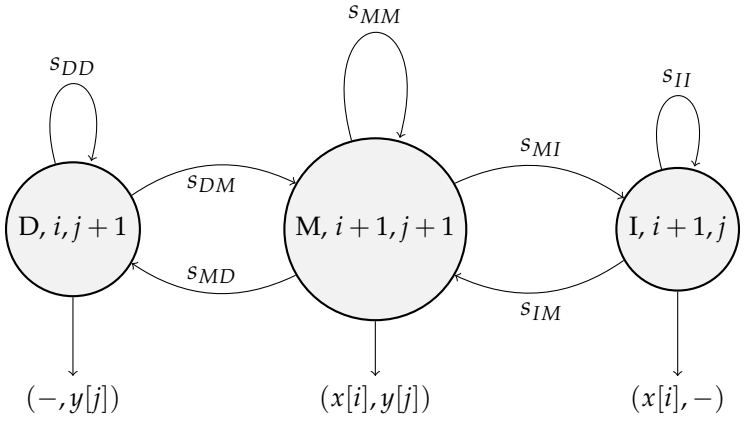
I first taught hidden Markov models from how they are programmed rather than from some mathematical formalism because I find that easier to understand. But once you get to more complicated HMMs, the formalism becomes useful. As you may recall from Formal Languages (COMP4200), Finite State Automata are a model of computation that can be in exactly one of a finite number of states at a given time, and may transition to other states at another time point given some transition rules. Doesn't this sound familiar to our HMM? The only addition we have is that depending on the state, there is an emission. In our CpG island example, this emission would either be a CG or a non-CG 2mer according to the probability of seeing a CG in that state. So the FSA looks like the following.



The emissions are called emissions because you could run this in a generative mode in which you generate sequences. In our case, we have observations.

And you keep a table of values for each state for each time. I normally drew this as a $2xn$ matrix, but we can also think of these as 2 $1xn$ arrays. The reason for this distinction should become clear soon.

For the alignment problem, we index sequence x with i and sequence y with j, so we have two time variables that can be incremented or not incremented at each step. The FSA will look like the following.



The M state is the match/mismatch state and emits $(x[i], y[j])$ and increments both i and j. The I state is the insertion of $x[i]$ with respect to sequence y and thus increments only i. The D state represents a deletion in x with respect to y (or an insertion in y with respect to x) and increments only j. It is always better to take a mismatch rather than two indels. So from the FSA, we will have three dynamic programming tables, one for each state. Each of these tables will be two dimensional, indexed by i and j for each of the two sequences x and y. And the recurrence relation can be derived from the FSA transitions and emission probabilities. For the optimal state sequence, we use max, and for the forward/backward algorithm, we sum over all possible previous states. Here I will use sum. Unlike the prior formulation of alignment, entering the match/mismatch M state, this indicates an alignment between $x[i]$ and $y[j]$. The previous formulation relied on the transition to indicate match/mismatch versus indels.

$$M[i,j] = sum \begin{cases} M[i-1,j-1]s_{MM}p(x[i],y[j]) \\ I[i-1,j-1]s_{IM}p(x[i],y[j]) \\ D[i-1,j-1]s_{DM}p(x[i],y[j]) \end{cases}$$

$$I[i,j] = sum \begin{cases} M[i-1,j]s_{MI}p(x[i],-) \\ I[i-1,j]s_{II}p(x[i],-) \end{cases}$$

$$D[i,j] = sum \begin{cases} M[i,j-1]s_{MD}p(-,y[j]) \\ D[i,j-1]s_{DD}p(-,y[j]) \end{cases}$$

And then for the backward algorithm, we have a similar update rule working in the opposite direction.

So now let's code it.

```
import numpy as np
import pandas as pd
from plotnine import *

## /Library/Frameworks/R.framework/Versions/4.1/Resources/library/reticulate/python/rpytools/loader.py:3

import scipy
np.seterr(divide = 'ignore')

## {'divide': 'warn', 'over': 'warn', 'under': 'ignore', 'invalid': 'warn'}
```

```
def random_sequence(n):
    return("".join(np.random.choice(["A","C","G","T"], n)))
```

```
def mutate(s, snp_rate, indel_rate):
    x = [c for c in s]
    i = 0
    while i < len(x):
        if np.random.random() < snp_rate:
            x[i] = random_sequence(1)
        if np.random.random() < indel_rate:
            length = np.random.geometric(0.5)
            if np.random.random() < 0.5: # insertion
                x[i] = x[i] + random_sequence(length)
            else:
                for j in range(i,i+length):
                    if j < len(x):
                        x[j] = ""
                    i += 1
        i += 1
    return("".join(x))
```

```
s1 = random_sequence(100)
s2 = mutate(s1,0.1,0.1)
print(s1)

## CGCGTGCATGGCGCTTTACACGTACTCGGCCTTCCGGGTAACGAGTCACTGGATCCGATAGCTTCGGTCAACCGCGCGTTGTGCTTGCGTATTGACCTAG

print(s2)

## GCGTGCATGGCGGGAACGTTTACCGTACCTACACGGCCTTCCGGGTAGACGAGTGTTCATGGTTCCGATAGCTTCCTGTTCTACGCGCGTGTGCTTGCGTA
```

```
def forward_align(s1, s2, log_emissions, log_transitions):
    # log_emissions [log_prob_match, log_prob_mismatch, log_prob_indel]
    # log_transitions 3x3 from state x to state log probabilities

    match_mismatch = np.zeros((len(s1)+1, len(s2)+1)) # always looks back i-1, j-1
    insertion = np.zeros((len(s1)+1, len(s2)+1)) # always looks back i-1,j
    deletion = np.zeros((len(s1)+1, len(s2)+1)) # always looks back i,j-1

    match_mismatch[0][0] = np.log(1) # this is treated as the start state
    insertion[0][0] = np.log(0)
    deletion[0][0] = np.log(0)
    # initialize
    for i in range(1,len(s1)+1):
        insertion[i][0] = insertion[i-1][0] + log_emissions[2] # if we get here we must go left with pro
        deletion[i][0] = np.log(0) # cannot get here
        match_mismatch[i][0] = np.log(0) # cannot get here
    for j in range(1,len(s2)+1):
        deletion[0][j] = deletion[0][j-1] + log_emissions[2] # if we get here, we must go up
        insertion[0][j] = np.log(0) # cannot get here
        match_mismatch[0][j] = np.log(0) # cannot get here
```

```
    # fill out score matrix
    for i in range(1,len(s1)+1):
        for j in range(1,len(s2)+1):
            match_emission = log_emissions[1] # mismatch
            if s1[i-1] == s2[j-1]:
                match_emission = log_emissions[0] # match

            match_scores = [match_mismatch[i-1][j-1]+log_transitions[0][0]+match_emission,
                     insertion[i-1][j-1]+log_transitions[1][0]+match_emission,
                     deletion[i-1][j-1]+log_transitions[2][0]+match_emission]
            match_mismatch[i][j] = scipy.special.logsumexp(match_scores)

            insertion_scores = [match_mismatch[i-1][j]+log_transitions[0][1]+log_emissions[2],
                                insertion[i-1][j]+log_transitions[1][1]+log_emissions[2]]
            insertion[i][j] = scipy.special.logsumexp(insertion_scores)

            deletion_scores = [match_mismatch[i][j-1]+log_transitions[0][2]+log_emissions[2],
                                deletion[i][j-1]+log_transitions[2][2]+log_emissions[2]]
            deletion[i][j] = scipy.special.logsumexp(deletion_scores)

    return(match_mismatch, insertion, deletion)
```

Now we can use the forward algorithm to sample alignments weighted on their likelihood using a probabilistic backtrace. We start at $i = |x|$ and $j = |y|$ and sample from being in the M, I, D states by normalizing those likelihoods to sum to one and sampling among them. Then at each step of the probabilistic backtrace, we normalize the likelihoods of the possible previous states to sum to one and sample among them.

```
def sample_alignment(s1, s2, forward_match_mismatch,
                     forward_insertion, forward_deletion):
    # sampling a probabilistic alignment
    i = len(s1)
    j = len(s2)
    alignment_s1 = ""
    alignment_s2 = ""
    #possible previous states to normalize
    previous_states = np.asarray([forward_match_mismatch[i][j],
                                  forward_insertion[i][j],
                                  forward_deletion[i][j]])

    # probabilistic backtrace algorithm
    while i > 0 and j > 0:
        # normalize previous state probabilities to sum to 1 (in log space)
        normalized = np.exp(previous_states - scipy.special.logsumexp(previous_states))
        for x in range(1,3): # get cumulative probability for sampling
            normalized[x] += normalized[x-1]

        random_sample = np.random.random()

        if random_sample <= normalized[0]:
            current_state = 0
        elif random_sample <= normalized[1]:
            current_state = 1
        else:
            current_state = 2
```

```
        if current_state == 0: # match/mismatch
            i -= 1
            j -= 1
            alignment_s1 = s1[i] + alignment_s1
            alignment_s2 = s2[j] + alignment_s2
            previous_states = np.asarray([forward_match_mismatch[i][j],
                                          forward_insertion[i][j],
                                          forward_deletion[i][j]])
        elif current_state == 1: # insertion
            i -= 1
            alignment_s1 = s1[i] + alignment_s1
            alignment_s2 = "-" + alignment_s2
            previous_states = np.asarray([forward_match_mismatch[i][j],
                                          forward_insertion[i][j],
                                          np.log(0)])
        else: # current_state is 2: deletion
            j -= 1
            alignment_s1 = "-" + alignment_s1
            alignment_s2 = s2[j] + alignment_s2
            previous_states = np.asarray([forward_match_mismatch[i][j],
                                          np.log(0),
                                          forward_deletion[i][j]])

    print("sampled alignment")
    print(alignment_s1)
    print(alignment_s2)
```

And now we can define some transition and emission probabilities and sample a probabilistic alignment.

```
log_emissions = np.log(np.asarray([0.8, 0.1, 0.1]))
transitions = [[1/3, 1/3, 1/3], # match can transition to anything
               [1/2,1/2,0], # insertion can only go to insertion or match
               [1/2,0,1/2]] # deletion can only go to deletion or match
log_transitions = np.log(np.asarray(transitions))
(forward_match, forward_insertion, forward_deletion) = forward_align(s1, s2, log_emissions, log_transiti
sample_alignment(s1, s2, forward_match, forward_insertion, forward_deletion)

## sampled alignment
## CGCGTGCATGG-CG--CT-TT-A-C--ACGTACTCGGCCTTCCGGGTA-A--CGAGTCACTGGATCCGATAGCTTCGG--TCAACCGCGCGTTGTGCTTGC
## G-CGTGCATGGCGGGAACGTTTACCGTACCTACACGGCCTTCCGGGTAGACGAGTGTTCATGGTTCCGATAGCTTCCTGTTCTAC-GCGCGT-GTGCTTGC
```

And now for the backward algorithm applied to this pair-HMM.

```
def backward_align(s1, s2, log_emissions, log_transitions):
    reverse_match = np.zeros((len(s1)+1, len(s2)+1))
    reverse_insertion = np.zeros((len(s1)+1, len(s2)+1))
    reverse_deletion = np.zeros((len(s1)+1, len(s2)+1))

    # initialization
    reverse_match[len(s1)][len(s2)] = np.log(1) # final state we must end up in for global alignment
    reverse_insertion[len(s1)][len(s2)] = np.log(0) # cant get heree
    reverse_deletion[len(s1)][len(s2)] = np.log(0) # cant get here
    for i in reversed(range(0,len(s1))):
        reverse_match[i][len(s2)] = np.log(0) # cant get here
```

```
        reverse_insertion[i][len(s2)] = reverse_insertion[i+1][len(s2)] + log_emissions[2] # must go rig
        reverse_deletion[i][len(s2)] = np.log(0) # cant get here
    for j in reversed(range(1,len(s2))):
        reverse_match[len(s1)][j] = np.log(0) # cant get here
        reverse_insertion[len(s1)][j] = np.log(0) # cant get here
        reverse_deletion[len(s1)][j] = reverse_deletion[len(s1)][j+1] + log_emissions[2] # must go down

    for i in reversed(range(0,len(s1))):
        for j in reversed(range(0, len(s2))):
            match_emission = log_emissions[1] # mismatch
            if s1[i] == s2[j]:
                match_emission = log_emissions[0] # match
            reverse_match_scores = [reverse_match[i+1][j+1] + log_transitions[0][0] + match_emission,
                        reverse_insertion[i+1][j+1]+log_transitions[0][1]+match_emission,
                        reverse_deletion[i+1][j+1]+log_transitions[0][2]+match_emission]
            reverse_match[i][j] = scipy.special.logsumexp(reverse_match_scores)

            reverse_insertion_scores = [reverse_match[i+1][j]+log_transitions[1][0]+log_emissions[2],
                                        reverse_insertion[i+1][j] + log_transitions[1][1] + log_emissions
            reverse_insertion[i][j] = scipy.special.logsumexp(reverse_insertion_scores)

            reverse_deletion_scores = [reverse_match[i][j+1]+log_transitions[2][0]+log_emissions[2],
                                       reverse_deletion[i][j+1] + log_transitions[2][2]+log_emissions[2]]
            reverse_deletion[i][j] = scipy.special.logsumexp(reverse_deletion_scores)

    return(reverse_match, reverse_insertion, reverse_deletion)
```

And to bring it together for posteriors for character $x[i]$ for a particular i to be aligned to each character $y[j]$ across all j. To get the likelihood of each state at each pair-timepoint i, j, we multiply the forward likelihoods by the backward likelihoods for each state for each pair timepoint. Then to get the posterior probabilities of a particular character in x is aligned to each character in y, we normalize by the sum of that column across all 3 states.

```
def forward_backward_align(s1, s2,
                           forward_match,
                           forward_insertion,
                           forward_deletion,
                          backward_match,
                          backward_insertion,
                          backward_deletion):
    log_likelihood_match = np.zeros((len(s1),len(s2)))
    log_likelihood_insertion = np.zeros((len(s1), len(s2)))
    log_likelihood_deletion = np.zeros((len(s1), len(s2)))
    for i in range(len(s1)):
        for j in range(len(s2)):
            log_likelihood_match[i][j] = (forward_match[i+1][j+1] +
                                          backward_match[i][j])
            log_likelihood_insertion[i][j] = (forward_insertion[i+1][j+1] +
                                            backward_insertion[i][j])
            log_likelihood_deletion[i][j] = (forward_deletion[i+1][j+1]+
                                          backward_deletion[i][j])

    log_likelihoods = np.asarray([log_likelihood_match,
                                  log_likelihood_insertion,
```

```
                                    log_likelihood_deletion])

    column_denoms = scipy.special.logsumexp(log_likelihoods, axis=(0,2))

    posteriors_match = np.exp(log_likelihood_match.T - column_denoms).T
    return(posteriors_match)
```

Now to run this and plot the posteriors.

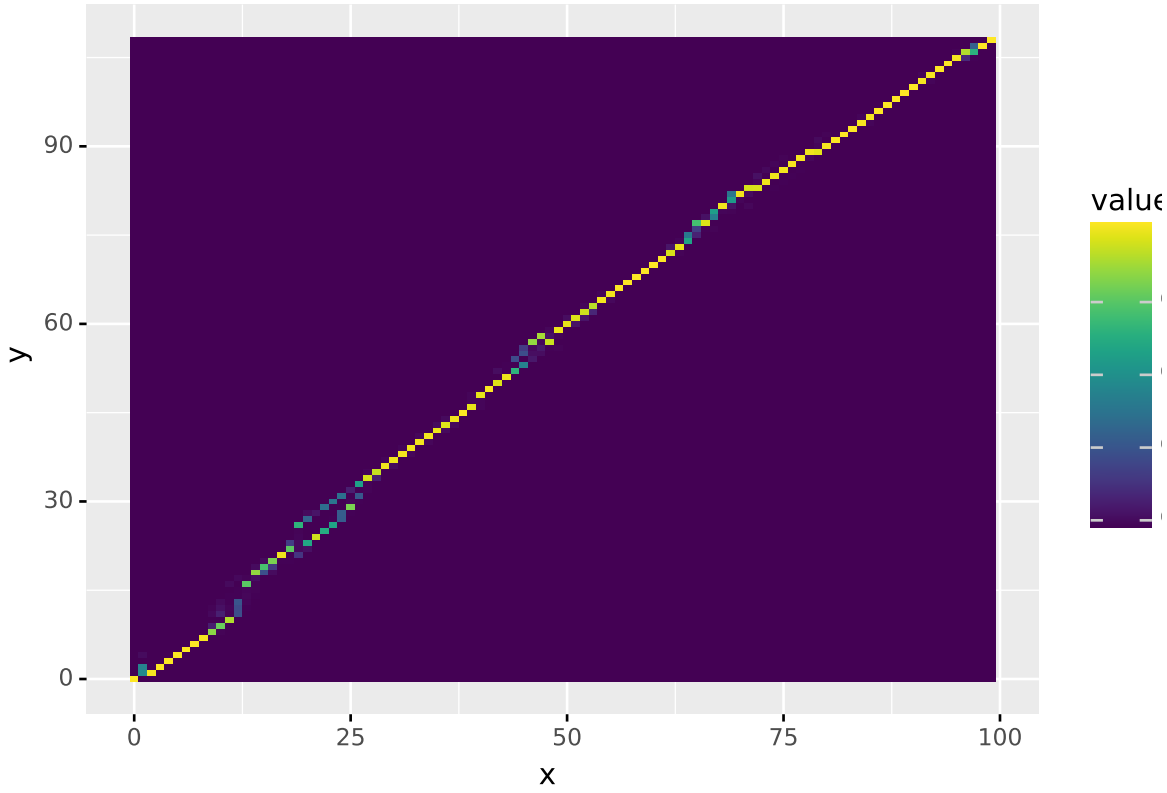
```
log_emissions = np.log(np.asarray([0.8, 0.1, 0.1]))
transitions = [[1/3, 1/3, 1/3], # match can transition to anything
               [1/2,1/2,0], # insertion can only go to insertion or match
               [1/2,0,1/2]] # deletion can only go to deletion or match
log_transitions = np.log(np.asarray(transitions))
(forward_match, forward_insertion, forward_deletion) = forward_align(s1, s2, log_emissions, log_transiti
(backward_match, backward_insertion, backward_deletion) = backward_align(s1, s2, log_emissions, log_tran
posteriors_match = forward_backward_align(s1,s2,forward_match, forward_insertion,
                                          forward_deletion, backward_match,
                                          backward_insertion, backward_deletion)
```

And plot them.

```
x = []
y = []
value = []
for i in range(posteriors_match.shape[0]):
    for j in range(posteriors_match.shape[1]):
        x.append(i)
        y.append(j)
        value.append(posteriors_match[i][j])
df = pd.DataFrame({'x':x,'y':y,'value':value})

(ggplot(df)+geom_tile(mapping=aes(x='x',y='y',fill='value'),stat="identity"))

## <ggplot: (8770365485562)>
```


90
60
30
0
y
0
25
50
75
100
x
value