COMP 5970/6970-004
Computational Biology: Genomics and Transcriptomics
Lecture notes 8: 2/8/2022

Haynes Heaton

Spring, 2022

Lecture Objectives

- Hidden Markov Models (HMM)
- States
- Transitions
- Optimal state sequence of data
- Implementation of Global Alignment
- Implementation of HMM

Hidden Markov models (HMMs)

We have used CpG islands previously as an example for the use of Bayes' theorem and the binomial distribution. There, we asked given a sequence, what is the probability that it came from a CpG island versus from background genomic sequence. What if, instead, we want to find the CpG islands and their most likely boundaries within some sequence? Here we can use a Hidden Markov Model.

0.1 Markov process

A Markov chain, or Markov process assumes that a sequence of events is randomly generated each only depending on the state of the previous event. The **states** may be different generating functions, such as a weighted coin versus a fair coin. And there may be some probability of changing from one to the other. A Hidden Markov model is when the states of each event are not known. However, given some data, with known **transition probabilities** between states and a likelihood function for each state, we can find the state sequence that maximizes the likelihood of the data.

0.2 CpG islands

In our example, we will have two states, or models—one in which the sequence was generated from a CpG island and one in which the sequence was generated from background genomic sequence. Each of these will be modeled with a *Bernoulli* distribution in which the sequence CG occurs at position t with some probability. State 0 will represent the data being generated from a CpG island and state 1 will represent the data coming from backgrand genomic sequence. We will have a **transition probability matrix** of the

chance of being in state i at time t and state j at time $t+1$. To make all of this more clear, let's make some definitions for our notation.
Definitions

- s_i State i
- t Position t in the sequence. Hidden Markov models are often used in time series data, and while we are using them for sequence data, p for position is overloaded with probability
- D_t observed data at time t (CG or not CG)
- $s_{i,j}$ Transition probability of going from state i to state j.

So our transition matrix may look like

	$s_0(t+1)$	$s_1(t+1)$
$s_0(t)$	0.99	0.01
$s_1(t)$	0.01	0.99

indicating a 99% chance to stay in the same state as the previous state. In order to find the state sequence that maximizes the data, we can again use dynamic programming to find the maximum likelihood of the data at time t given a particular state and the maximum likelihoods of all states at time $t-1$. We start with some prior probabilities π for each state.

$$p(D_0|s_i) = \pi_i$$

$$p(D_{0..t}|s_i) = max_j(p(D_{0..t-1}|s_j)s_{i,j})p(D_t|s_i)$$

So we choose the maximum previous state times the transition probability from that state j to this state i and then multiply that time the likelihood of the data at the current position t given the current state i and keep track of which state we came from to get to this state. In practice, we keep a matrix of m by $n+1$ where m is the number of states and n is the length of the data sequence. We put the priors for each state in the first cell of that state, and then fill out this dynamic programming table hmm according to this equation.

t

s_0	π_0	$max_j(hmm[j][t-1] * s_{j,0})p(D_t|s_0)$	..	..	..	..	..
s_1	π_1	$max_j(hmm[j][t-1] * s_{j,1})p(D_t|s_1)$	..	..	..	..	..

We also keep a backtrace matrix of the state from which we came at every cell. And at the end, we choose the maximum likelihood state at the final timepoint and we run the Viterbi backtrace algorithm from that point to the beggining for the optimal state sequence for this data.

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

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

Let's load some sequence from the human genome containing a CpG island.

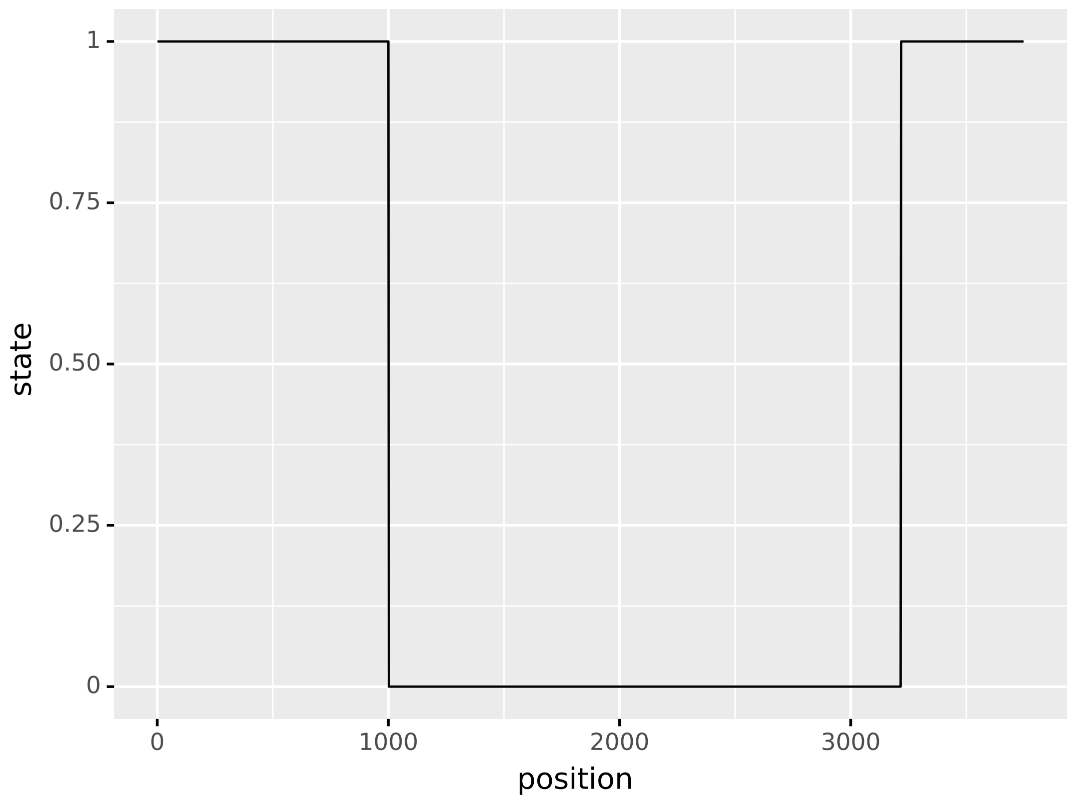
```
cpgtest = ""
with open("cpgtest.txt") as infile:
    cpgtest = infile.readline()
len(cpgtest)

## 3749
```

And write a function to find the optimal state sequence. Keep in mind that we need to compute this in log space because of numerical stability of many probabilities multiplied together.

```
def HMM(observed, priors, transitions, probabilities):
    # priors are 2 values, one for each state
    # transitions are 2x2 (from state x to state)
    # probabilities are 2x2 (state x [CpG prob, 1-CpG prob])
    hmm = np.zeros((2, len(observed))) # 2 state hmm
    backtrace = np.zeros((2, len(observed)))
    hmm[0][0] = np.log(priors[0]) # compute in log space
    hmm[1][0] = np.log(priors[1])
    log_transitions = np.zeros((2,2)) # convert inputs to log space
    log_probabilities = np.zeros((2,2))
    for i in range(2):
        for j in range(2):
            log_transitions[i][j] = np.log(transitions[i][j])
            log_probabilities[i][j] = np.log(probabilities[i][j])

    for i in range(1, len(observed)):
        CG = 1 # to index into probabilities, index 1 is not CG
        if observed[i-1:i+1] == "CG":
            CG = 0 # index 0 is for CG
        # loop over current state
        for current_state in range(2):
            # list comprehension over possible previous states
            log_probs = [hmm[previous_state][i-1] +
                         log_transitions[previous_state][current_state] +
                         log_probabilities[current_state][CG] for previous_state in range(2)]
            # find max of coming from each previous state
            max_log_prob = max(log_probs)
            hmm[current_state][i] = max_log_prob
            backtrace[current_state][i] = log_probs.index(max_log_prob)



    # viterbi backtrace
    i = len(observed) - 1
    current_state = 0
    if hmm[1][i] > hmm[0][i]:
        current_state = 1 # start backtrace at maximum likelihood state of final position
    states = []
    positions = []
    while i > 0:
        positions.append(i)
        states.append(current_state)
        current_state = int(backtrace[current_state][i])
        i -= 1
    return((states, positions))
```

```
priors = [0.5,0.5]
probabilities = [[0.04,0.96],[0.01,0.99]] # state 0 is CpG island and state 1 is background genome
transitions = [[0.99,0.01],[0.01,0.99]] # 99% chance to stay in your state
(states, positions) = HMM(cpgtest, priors, transitions, probabilities)
```

```
df = pd.DataFrame({'x': positions,'y': states})
ggplot(df) + geom_line(aes(x='x',y='y'))+ylab("state")+xlab("position")

## <ggplot: (8760584448859)>
```



So as you can see, it begins in the genomic background state, eventually transitions to the CpG state, and later it transitions back to background genomic sequence thus finding the CpG island in our data.

1 Sequence alignment implementation

I think that more modalities you cover a subject on the better. So far we have described sequence alignment visually (via hand drawn out DP tables) and mathematically via the recursion relation. Now lets put it in code.

First let's generate some random sequence to align.

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

So "".join takes an array of strings and joins them separated by "", so nothing. And np.random.choice randomly chooses among an array n times. So this gives us a random string on the alphabet A,C,G,T.

Now let's mutate this sequence a bit to have something to align it to.

```
s1 = random_sequence(100)
s1

## 'GTCTTATTAGAGGAGTTCCTTGGTAAAGACGGTTTCGCGACGATCCAACCTAGCCCAGTTGGAGCGAGTGGACTCCCCAACTAACCCTCCTATAAAGAAG
```

```
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))
```

```
s2 = mutate(s1,0.1,0.1)
s2

## 'GTCTCATTCGATAGTTGAGTAAAGAGCGGTTTCGCGCTCCAACACTACAGATAGGAGCGAGTAGCTCCCAACTAACCCTCCTATAAAGACAAGG'
```

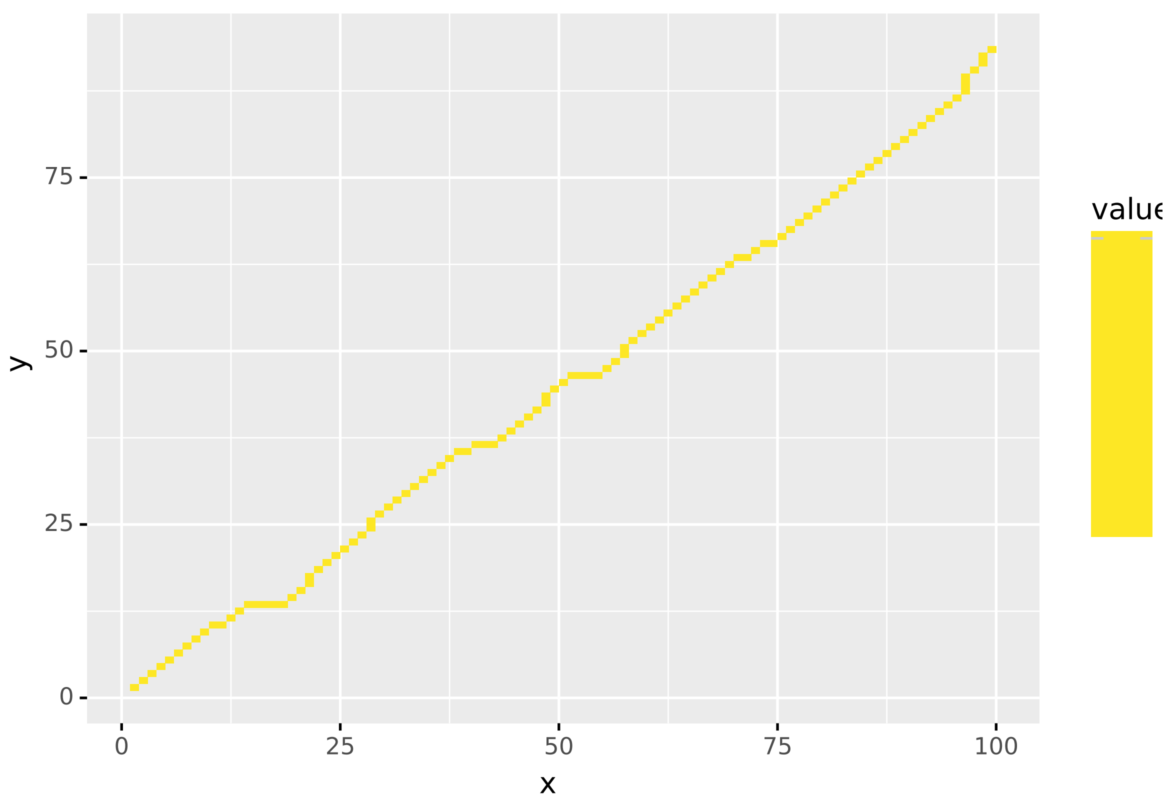
And now let's code global alignment.

```
def global_alignment(s1, s2):
    score = np.zeros((len(s1)+1, len(s2)+1))
    backtrace = np.zeros((len(s1)+1, len(s2)+1)) # encoding will be 0 is diagonal, 1 is left, 2 is up
    score[0][0] = 0
    for i in range(1, len(s1) + 1): # initialize top row
        score[i][0] = score[i-1][0] - 1
    for i in range(1, len(s2) + 1): # initialize left column
        score[0][i] = score[0][i-1] - 1

    for i in range(1, len(s1) + 1):
        for j in range(1, len(s2) + 1):
            diagonal = -1
            if s1[i-1] == s2[j-1]:
                diagonal = 1
            scores = [score[i-1][j-1] + diagonal, score[i-1][j] - 1, score[i][j-1] - 1]
            max_score = max(scores)
            score[i][j] = max_score
            backtrace[i][j] = scores.index(max_score) # index matches up with our encoding above
    # viterbi backtrace
    i = len(s1)
    j = len(s2)
    x = [] # these are just for visualization
    y = []
    s1_alignment = ""
    s2_alignment = ""
    while i > 0 or j > 0:
        x.append(i)
        y.append(j)
        if backtrace[i][j] == 0:
            i -= 1
            j -= 1
            s1_alignment = s1[i] + s1_alignment # emit both
            s2_alignment = s2[j] + s2_alignment
        elif backtrace[i][j] == 1:
            i -= 1
            s1_alignment = s1[i] + s1_alignment # emit s1 character and - for s2
            s2_alignment = "-" + s2_alignment
        else:
            j -= 1
            s1_alignment = "-" + s1_alignment # emit - for s1 and s2 character
            s2_alignment = s2[j] + s2_alignment
    return(s1_alignment, s2_alignment, x, y)
```

```
(s1_aligned, s2_aligned, x, y) = global_alignment(s1, s2)
print(s1_aligned)

## GTCTTATTAGAGGAGTTCCTTG-GTAAAGA-CGGTTTCGCGACGATCCAAC-CTAGCCCAG-TTGGAGCGAGTGGACTCCCCAACTAACCCTCCTATAAAG

print(s2_aligned)

## GTCTCATTCGA-TAG----TTGAGTAAAGAGCGGTTTCGCG-C--TCCAACACTA---CAGATAGGAGCGAGTAG-CT-CCCAACTAACCCTCCTATAAAG
```

And we can also visualize the Viterbi backtrace.

```
df = pd.DataFrame({'x':x,'y':y,'value':[1]*len(x)})
ggplot(df)+geom_bin2d(aes(x='x',y='y',fill='value'),binwidth=1)

## <ggplot: (8760584528707)>
```



y
x
0
25
50
75
100
value