Lab 08 Hidden Markov Models

The goal of Lab 08 | Hidden Markov Models is to use a hidden Markov model to identify CpG islands in a sequence. The lab is divided into three sections:

1. Review of Markov Chains
2. Hidden Markov Models
3. Viterbi Algorithm

Assignment

Follow the instructions in this document and answer the questions in the cell below each question. Submit your answers by uploading a PDF file to gradescope. To generate the pdf, first export the notebook as HTML: >File, >Export to ..., >HTML. Then, open the HTML in a browser and use your browser to print to PDF.

Check to make sure all your cells have been run and the **results** displayed in the PDF file.

Reminder, provide comments for any code you write to ensure partial credit.

Markov Chains

In Lab 07 we used a Markov Chain to simulate DNA sequences. By doing so we were able to capture an important feature of the human genome: CpG sites are depleted. We estimating the transition probabilities $P(X_t \mid X_{t-1})$ from a genome sequence and then we were able to simulate sequences using a Markov chain with the same nucleotide and di-nucleotide frequencies as those observed.

Markov Model

The following shows a graphical representation of our Markov model:

To encode this model we used a 4x4 transition matrix of going from $[A, G, C, T]$ (rows) to $[A, G, C, T]$ (columns).

-	A	G	C	T
A	0.3472	0.1971	0.1576	0.2979
G	0.2904	0.2205	0.1930	0.2959
C	0.3619	0.0425	0.2316	0.3638
T	0.2422	0.2151	0.1852	0.3572

Probability of a sequence under a Markov Model

Given the transition matrix we can simulate random sequences under the Markov model, as we did in the last lab. However, we can also calculate the probability of a sequence being generated by the Markov model. Given a sequence x:

$$P(x) = P(x_k \mid x_{k-1}) \cdot P(x_{k-1} \mid x_{k-2}) \ldots P(x_2 \mid x_1) \cdot P(x_1)$$

If x = GATC

P(x) = P(C | T) P(T | A) P(A | G) P(G)

Let's use the Markov model from Lab 07 to calculate the probability of x = GATC and assume P(G) = 0.25.

In [1]:
```
import numpy as np
P = np.matrix([
 [ 0.34722501,  0.19715488,  0.15768383,  0.29793629],
 [ 0.29046639,  0.22050754,  0.1930727,   0.29595336],
 [ 0.36191099,  0.04253927,  0.23167539,  0.36387435],
 [ 0.24229484,  0.21518752,  0.18529521,  0.35722243]])

P[3,2]*P[0,3]*P[1,0]*0.25
```

Out[1]:
```
0.004008884039213397
```

Why would this be useful?

Given a sequence and two Markov models, we can compare the probability of each model generating the sequence and determine which model is more likely to have generated it using the likelihood ratio.

First, lets generate a null model P0 based on the nucleotide frequencies alone by taking the normalized sum of the P matrix:

In [2]:
```
NucFreq = np.sum(P,axis=0)/np.sum(P)
P0 = np.vstack((NucFreq,NucFreq,NucFreq,NucFreq))
P0
```

Out[2]:
```
matrix([[0.31047431, 0.1688473 , 0.19193178, 0.32874661],
        [0.31047431, 0.1688473 , 0.19193178, 0.32874661],
        [0.31047431, 0.1688473 , 0.19193178, 0.32874661],
        [0.31047431, 0.1688473 , 0.19193178, 0.32874661]])
```

Now lets calculate the probability of a sequence 'CGCGCGC' under both models. To do this more easily, lets use a function that takes a sequence and a transition matrix as input and outputs the log10 probability of the sequences. But before doing so, lets go over underflow and list comprehension.

Arithmetic underflow

Underflow is a condition in a computer program where the result of a calculation is a number of smaller absolute value than the computer can actually represent in memory on its CPU. For long sequences the probabilities will become quite small and so to avoid underflow we should use log probabilities.To avoid underflow log probabilities are used rather than raw probabilities.

List comprehension

It is not necessary, but helpful to use list comprehension. List comprehensions provide a concise way to create lists. For example, make a new list where each element is the result of some operations applied to each member of another sequence or iterable.

Say we want each 2 bp sequence from a string of nucleotides. This can easily be done using a for loop:

In [3]:
```
seq = "ACGTGGA"
dinuc = []
for i in range(len(seq)-1):
    dinuc.append(seq[i:i+2])
print(dinuc)
```

```
['AC', 'CG', 'GT', 'TG', 'GG', 'GA']
```

However, list comprehension provides an easier way. A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses.

Thus, an alternative encoding:

In [4]:
```
dinuc = [seq[i:i+2] for i in range(len(seq)-1)]
print(dinuc)
```

```
['AC', 'CG', 'GT', 'TG', 'GG', 'GA']
```

Here is a function `classify` that will output log10 probabilities as stated above:

In [5]:
```
def classify(seq, PTab):
    """ Classify seq using Markov model.  We're ignoring the
        initial probability for simplicity. """
    bits = 0
    nucmap = { 'A':0, 'C':1, 'G':2, 'T':3 }
    # using list comprehension to get dinucleotides (2 bases) by taki
ng i over range of length seq, extract to bases (i:i+2) from seq.
    # the result of which is placed in dinuc for each iteration
    for dinuc in [ seq[i:i+2] for i in range(len(seq)-1) ]:
        i, j = nucmap[dinuc[0]], nucmap[dinuc[1]]
        bits += np.log10(PTab[i, j])
    return bits
```

What is the log likelihood ratio of P/P0?

We can use `classify` to get the log probabilities and the difference between them is the log likelihood ratio.

In [6]:
```
model1 = classify("CGCGCGC",P)
model2 = classify("CGCGCGC",P0)
print("Model 1 = ",model1)
print("Model 2 = ",model2)

print("Log likelihood ratio = ", model2 - model1)
print("Likelihood ratio = ", 10**(model2-model1))
```

```
Model 1 =  -6.256467291508364
Model 2 =  -4.4680769297056395
Log likelihood ratio =  1.7883903618027244
Likelihood ratio =  61.43139277919189
```

Thus, model2 (P0) is 61 times more likely than model1 (P) to have generated the sequence. 10^LLR(P0/P) = 61.4, where LLR is the log10 likelihood ratio.

Hidden Markov Models

Hidden Markov Model (HMM) is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (i.e. hidden) states.

In a Markov chain the state is directly visible, and therefore the state transition probabilities are the only parameters. In a hidden Markov model, the state is not directly visible, but the output is visible and dependent on the state.

Graphically, the states of a Markov chain are observed, whereas in a HMM the states are unobserved but they emit observed variables. The example below shows a Markov chain for a two state chain (Heads and Tails). Below that is an HMM with two different coins (Fair and Loaded). Each coin emits either a Heads or Tails outcome that is observable. However, the coin that is used is not known (observable). This HMM provides a model of a coin toss where either a Fair or Loaded coin are used and the model switches between using the Fair or Loaded coin over time.

Trellis diagram

A trellis diagram below shows the general architecture of an instantiated HMM. Each oval shape represents a random variable that can adopt any of a number of values. The random variable $p(t)$ is the hidden state at time t (with the model from the above diagram, $p(t) \in F, L$). The random variable $x(t)$ is the observation at time t (with $x(t) \in H, T$). The arrows in the diagram (often called a trellis diagram) denote conditional dependencies.

$p = \{p_1, p_2, \ldots, p_n\}$ is a sequence of states (AKA a path). Each p_i takes a value from set Q (states).

$x = \{x_1, x_2, \ldots, x_n\}$ is a sequence of emissions. Each x_i takes a value from set ∑ (observations).

Edges (arrows) capture conditional independence:

- x_2 is conditionally independent of everything else given p_2
- p_4 is conditionally independent of everything else given p_3

The probability of being in a particular state at step i is known once we know what state we were in at step $i-1$.
Th probability of seeing a particular emission at step i is known once we know what state we were in at step i.

Given the example of a Fair and Loaded coin, six tosses can be diagrammed below:

An HMM can thus be encoded by a transition matrix (A) and an emission matrix (E):

$|Q| \times |\sum|$ emission matrix **E** encodes $P(x_i \,|\, p_i)$

$E[p_i, x_i] = P(x_i \,|\, p_i)$

$|Q| \times |Q|$ transition matrix **A** encodes $P(p_i \,|\, p_{i-1})$

$A[p_{i-1}, p_i] = P(p_i \,|\, p_{i-1})$

$|Q|$ array **I** encodes initial probabilities of each state

$I[pi] = P(p_1)$

The probability of $P(p_1, p_2, \ldots, p_n, x_1, x_2, \ldots, x_n) =$

$\prod_{k=1}^{n} P(x_k \,|\, p_k) \;\; \prod_{k=2}^{n} P(p_k \,|\, p_{k-1}) \;\; P(p_1)$

Given this model, we can then calculate the joint probability of 'THTHHHTHTTH' & 'FFFLLLFFFFF' as:

If $P(p_1 = F)$ = 0.5, then joint probability = $0.5^9 0.8^3 0.6^8 0.4^2$ = 0.0000026874

Viterbi Algorithm

While calculating the probability of data under an HMM as done above is straightforward, we typically do not know the hidden states. For example, it was easy to calculate the probability of a sequence given two Markov models, one for generating sequence within a CpG island and the other for generating sequence outside a CpG island. HMMs are particularly useful when we do not know whether a coin is Fair or Loaded, or whether a sequence is within a CpG island or not.

The **Viterbi algorithm** is a dynamic programming algorithm for finding the most likely sequence of hidden states —called the Viterbi path—that results in a sequence of observed events in the context of a hidden Markov model. The Viterbi algorithm is known as a decoding algorithm.

To find the most likely sequence of hidden states, the Viterbi uses dynamic programming to iteratively calculated the score of the most likely path up to a certain state (k), using scores from prior calculation of the prior state (k-1). In assessing the most like likely path, it is exhaustive.

$s_{k,i}$ = score of the most likely path up to step *i* with $p_i = k$

To calculate the most likely path out of all possible paths, at each transition the Viterbi asks which path is most likely. So if we want to calculate the probability of a Fair coin at position i in the sequence of tosses, we need to consider two possibilities: the coin was fair in the prior state *or* the coin was loaded in the prior state.

Let's calculate the probability of the most likely path. This involves calculating each $s_{k,i}$, where k is the current state, and i is the current position in the sequence. Similar to the Needleman-Wunch algorithm, we also need to keep track of which path (most likely transition) at each step so that we can use a backtrace to finally calculate the most likely path.

Consider the transition (A) and emission (E) matrix below and the observation 'THTH'.

What is $s_{k,i}$?

Lets start with initial probabilities I of states $F, L = [0.5, 0.5]$. Initial state probabilities are needed, and we'll arbitrarily assume equal chances for each.
$s_{F,1}$ = *I* x *E* = 0.5 x 0.5
$s_{L,1}$ = *I* x *E* = 0.5 x 0.2

Once we have these initial values of $s_{k,1}$ we can use the recursion formulate for each subsequent position in the sequences of 'THTH' observations.
$s_{k,i}$ = *E* · max ($s_{k,i-1}$ · A)

$s_{F,2}$ = P(H | F) · max ($s_{k,1}$ · P(F | k))
The maximum is of these two possibilities:
max ($s_{F,1}$ · P(F | F) , $s_{L,1}$ · P(F | L))
Filling in with numbers we have:
max (0.5 x 0.5 x 0.6 , 0.5 x 0.2 x 0.4)
max is for a fair coin in prior state = 0.5 x 0.5 x 0.6
Thus, $s_{F,2}$ = 0.5 x 0.5 x 0.5 x 0.6

$s_{L,2}$ = P(H | L) · max ($s_{k,1}$ · P(L | k))
max ($s_{F,1}$ · P(L | F) , $s_{L,1}$ · P(L | L))
max (0.5 x 0.5 x 0.4 , 0.5 x 0.2 x 0.6)
max is for a fair coin in prior state = 0.5 x 0.5 x 0.4
Thus, $s_{L,2}$ = 0.8 x 0.5 x 0.5 x 0.4

In addition to recording $s_{F,2}$ and $s_{L,2}$, we also need to record which was the most likely path for traceback. In both cases the most likely path to $s_{F,2}$ and $s_{L,2}$ was from $s_{F,1}$. Subsequent calculations of $s_{k,i}$ can be made in similar ways using the preceeding $s_{k,i-1}$ values.

Question 1

What is $s_{F,3}$ and $s_{L,3}$?
Use the A and E matrix given above and use $s_{F,2} = 0.17$ and $s_{L,2} = 0.28$ in your calculation.
(4 points)

In [27]:
```
# Answer
```

Encoding the viterbi algorithm

Lets write a function to calculate all $s_{k,i}$ for any sequence of Heads and Tails.

Lets start by encoding the I, A and E matrices:

In [8]:
```
# The transition probabilities of Fair (F) to Loaded (L), emission pr
obabilities of Heads (H) and Tails (T)
# are stored in the A, E and I matrices
import numpy as np
Amat = np.matrix([[0.7, 0.3], [0.4, 0.6]]) # transition matrix A
Emat = np.matrix([[0.5, 0.5], [0.8, 0.2]]) # emission matrix E
Imat = np.array([0.5,0.5]) #  initial probabilities I

# To work in log10 space multiple probabilities by log10
Amat = np.log10(Amat)
Emat = np.log10(Emat)
Imat = np.log10(Imat)
```

Now lets encode the Viterbi algorithm

In [9]:
```
# Viterbi function with input of the string of observations and hmm m
odel (E, A, I matrices)

def viterbi(obs, Amat, Emat, Imat):

    # use a character state to row/column ID map and observations to
 row id map
    stmap = {
        "F": 0,
        "L": 1,
        "H": 0,
        "T": 1
    }
    # split observations into list
    char = list(obs)

    # initialize a matrix of s(k,i) and pointers to prior most likely
state.
    nrow, ncol = 2, len(obs)
    matS  = np.zeros(shape=(nrow, ncol), dtype=float) # s(k,i) where
 dtype indicates the datatype
    matTb = np.zeros(shape=(nrow, ncol), dtype=int)   # tb(k,i) where
tb points to the prior case

    # Fill in first column
    for k in range(0, nrow):
        # map first obs to row
        obs0 = stmap[ obs[0] ]
        matS[k, 0] = Emat[k,obs0]+Imat[k] #addition since we are usin
g log(P) rather than P

    # Fill in rest of s(k,i) and Tb tables
    for i in range(1, ncol):
            for k in range(0, nrow):
                # map current observed value to row
                obsj = stmap[ obs[i]]
                # s(F, i-1) * P(F|F) * E(obs|F) BUT use addition sinc
e we are using log(P) rather than P
                sf = matS[0,i-1] + Amat[0,k] + Emat[k,obsj] # s(k,i-
1) * transition prob. * emission prob.
                # s(L, i-1) * P(F|L) * E(obs|F)
                sl = matS[1,i-1] + Amat[1,k] + Emat[k,obsj]
                # Find max of sl and sf and asign probability to matS
                # and assign state to matTb
                if (sf < sl):
                    matS[k,i] = sl
                    matTb[k,i] = 1
                else:
                    matS[k,i] = sf
                    matTb[k,i] = 0
    # Find final state with maximal log probability
    if ( matS[0,ncol-1] > matS[1,ncol-1] ):
        final = 0
        p = 'F'
    else:
        final = 1
```

```
        p = 'L'
    # Backtrace
    k = final
    backmap = ["F","L"]
    for i in range(ncol-1, 0, -1):
        k = matTb[k, i]
        p = backmap[k] + p
    return matS, matTb, p
```

Example

Lets call the function and print the most likely path along with the observed states

In [10]:
```
obs = "HTTHHH"
m1, m2, p = viterbi(obs, Amat, Emat, Imat)
print(m1)
print(m2)
print(p)
print(obs)
```

```
[[-0.60205999 -1.05799195 -1.5139239  -1.96985586 -2.42578781 -2.8817
1977]
 [-0.39794001 -1.31875876 -2.23957752 -2.13371266 -2.45247142 -2.7712
3019]]
[[0 0 0 0 0 0]
 [0 1 1 0 1 1]]
FFFLLL
HTTHHH
```

Question 2

Above, the viterbi found "FFFLLL" to be the most likely sequence of hidden states. In this question, we will confirm the answer by an exhaustive search.

Write a function to calculate all possible paths of hidden states, FFFFFF through LLLLLL (2**6 = 64 paths) and their likelihoods given the A and E matrices above and the observed states "HTTHHH"?

Print the path and the log10 probability of each path. At the end, print the most likely path and its log10 probability.

Use the Amat, Emat and Imat matrices defined in the example above. You can use `import itertools` for enumeration but it is not necessary, it can also be done with loops or list comprehension.

(4 points)

In [26]:
```
# Answer
obs = "HTTHHH"
```

CpG island HMM

Our next goal is to find CpG islands in a sequence. To do this we will need to generate an A, E and I matrix for the HMM. The A and E matrix can be obtained empirically given sequences with known CpG islands within them. However, in this case lets start by assuming this has already been done to produce the following:

For I let's assume equal chances at the start [0.5, 0.5].

Question 3

Write a new viterbi function to calculate the most likely path of a sequence given A, E, and I matrices.

What is the most likely path of the following sequence:
GTTACTTATATAGCTTACTTCAAGGCGAGTCTCGGAGCCGCGCCCAACCGATGCTAATAAAATTTCTGTTTTG

Your function should output the sequence with the hidden states (I,O) above it.
(4 points)

In [25]:
```
obs = "GTTACTTATATAGCTTACTTCAAGGCGAGTCTCGGAGCCGCGCCCAACCGATGCTAATAAAA
TTTCTGTTTTG"

Amat = np.matrix([[0.8, 0.2], [0.2, 0.8]]) # transition matrix A
Emat = np.matrix([[0.1,0.4, 0.4, 0.1], [0.25, 0.25, 0.25, 0.25]]) # e
mission matrix E
Imat = np.array([0.5,0.5]) #  initial probabilities I

# To work in log10 space multiple probabilities by log10
Amat = np.log10(Amat)
Emat = np.log10(Emat)
Imat = np.log10(Imat)

# Answer
```

Given the CpG HMM described above, the most likely path for the following sequence is:

```
 OOOOOOOIIIIIIIIIIIIOOOOOOOOOOOOOOIIIIIIIIOOOOOOOOOIIIIIIII
 ATATATATCGCGCGCGCGATATATATATATATCCCCCCCCCATATATATATGGGGGGGG
```

where I and O indicate the hidden states for being inside and outside a CpG island. As you can see the 'CG' rich region is found correctly, but the 'C' rich or 'G' rich region is also called a CpG island.

This can be fixed by changing the number of states from two (I,O) to eight (IA, IG, IC, IT, OA, OG, OC, OT):

As also described in lecture, the orange cells are for transitions inside a CpG island (IA, IG, IC, IT) or outside a CpG (OA, OG, OC, OT), and the yellow cells are for transitioning from outside to inside states and vice-versa. White cells have zero probability, which means the only way to go from an inside to outside state is from IG and from outside to inside is going to a IC state.

Question 4

Write a new viterbi function to calculate the most likely path of a sequence. Instead of the 2 state model, use the 8 state model described above. For convenience the A, E, and I matrices are given below.

What is the most likely path of the following sequence:

```
 ATGTTGCTATTACCAGACGAAAACGCGTAGGCATACGTTTGCATAATGTCTGGGGGGGACAAGTTGCAATTTACAAGTCCGTT
```

Your answer should be a string of "I" and "O" characters indicating whether the sequence is inside (I) or outside (O) a CpG island.

(4 points)

In [13]:
```
# Matrices
Amat = np.matrix([
    [0.18, 0.40, 0.27, 0.15, 0.00, 0.00, 0.00, 0.00],
    [0.17, 0.35, 0.32, 0.16, 0.05, 0.05, 0.05, 0.05],
    [0.19, 0.24, 0.36, 0.21, 0.00, 0.00, 0.00, 0.00],
    [0.09, 0.36, 0.34, 0.21, 0.00, 0.00, 0.00, 0.00],
    [0.00, 0.00, 0.05, 0.00, 0.34, 0.19, 0.15, 0.29],
    [0.00, 0.00, 0.05, 0.00, 0.29, 0.22, 0.19, 0.29],
    [0.00, 0.00, 0.05, 0.00, 0.36, 0.04, 0.23, 0.36],
    [0.00, 0.00, 0.05, 0.00, 0.24, 0.21, 0.18, 0.35]])

Emat = np.matrix([
    [1,0,0,0],
    [0,1,0,0],
    [0,0,1,0],
    [0,0,0,1],
    [1,0,0,0],
    [0,1,0,0],
    [0,0,1,0],
    [0,0,0,1]])
Imat = np.array([0.125,0.125,0.125,0.125,0.125,0.125,0.125,0.125])
# Make sure row sums to 1
Amat = Amat / np.sum(Amat, axis = 1)
# To work in log10 space multiple probabilities by log10
# To avoid taking log10(0) add a small amount to each cell in the mat
rix
Amat = np.log10(Amat+.0001)
Emat = np.log10(Emat+.0001)
Imat = np.log10(Imat)

obs = "ATGTTGCTATTACCAGACGAAAACGCGTAGGCATACGTTTGCATAATGTCTGGGGGGGACAA
GTTGCAATTTACAAGTCCGTTGAACGATTCCACCGGTAGGAATGCGTGGCAACTTTAAATGTAACAGGA
CTC"
```

In [24]:
```
# Answer
```

Training an HMM

Guessing values for the A and E matrix *can* work, but an HMM will work much better if these matrices are learned from real data where we know the hidden states. The A and E matrices can be found when we have sequences along with the hidden states annotated through experimental data or other sources of information. We simply count transitions between states and count emisions, and then divide by the row sums to get probabilities.

Question 5

Count transitions in the provided files to generate an A matrix. Two files are provided: `chr11.sub.fasta` is 24 Mbp of chromosome 11 from the human genome, and `CPG.sub.states` is a file in the same format indicating outside CpG site by 0, and inside CpG site by 1.

Important points about these files:
--they are large, and so reading the entire file into memory may be problematic. Reading and processing one line at a time is more memory effecient (although you may be able to do it with the sequence in memory).
--the fasta file contains "N" which is an unknown base (gap in assembly).
--both files contain 70 sites per line, and so line 147, position 11 of both files is information for the base (fasta file) and state (state file) at the same position

Use the sequence file and states to fill out the A matrix used in Question 4. But, add one additional state `N` so that it is a 9x9 matrix. You don't need to fill out an emission matrix because each state only emits one base (as shown in Question 4) and the state `N` emits an N. To fill out the matrix, start with a matrix of zeros, add counts for each transition that is observed in the real data, then finally divide by row sums to make probabilities.

Also, use precision printing as indicated below.

Finally, state or print to screen what the most likely and least likely transition is from an outside (OA, OG, OC, OT) to an inside state (IA, IG, IC, IT).

(4 points)

In [23]:
```
Amat = np.zeros([9,9])

# Answer
print(np.array_str(Amat, precision=6, suppress_small=True))
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

In [ ]:
```
```