1. (HMMs with mixtures of Poissons)

    (a) For time steps \( t \in \{1, \ldots, T\} \), let \( Q_t \) be the Markov state sequence, \( P_t \), the corresponding mixture component indicators, and \( Y_t \) the observations. The initial state \( Q_0 \) and the homogeneous transitions in the Markov chain are distributed according to multinomial distributions, respectively with parameters

    \[
    \pi_q := \text{P}(Q_0 = q)
    \]

    and

    \[
    \tau_{q,q'} := \text{P}(Q_{t+1} = q' | Q_t = q).
    \]

    Conditional on \( Q_t = q \), the mixture component indicator \( P_t = p \) is also a multinomial with conditional mixture proportion parameter

    \[
    \rho_{q,p} := \text{P}(P_t = p | Q_t = q).
    \]

    Finally, conditionally on the current state \( Q_t = q \) and corresponding mixture component indicator \( P_t = p \) the emission \( Y_t \) is poissonian with rate \( \lambda_{q,p} \), i.e.

    \[
    \nu_{q,p,y} := \text{P}(Y_t = y | Q_t = q, P_t = p) = \frac{e^{-\lambda_{q,p}} \lambda_{q,p}^y}{y!}.
    \]

    We therefore have the following graphical model:

    ![Graphical Model Diagram]
Homework #4 solution 

STAT C241A/CS C281A, Fall 2009

(b) The expected complete log-likelihood take the following form:

$$E_{\text{post}} \log p(\tilde{q}, \tilde{p}, \tilde{y}) = E_{\text{post}} \log \left( \pi_{q_0} \left( \prod_{t=1}^{T-1} \tau_{q_t, q_{t+1}} \right) \left( \prod_{t=1}^{T} \rho_{q_t, p_t} \frac{e^{-\lambda_{q_t, p_t}}}{y_t!} \right) \right)$$

$$= E_{\text{post}} \left[ \log \pi_{q_0} + \sum_{t=1}^{T-1} \log \tau_{q_t, q_{t+1}} + \sum_{t=1}^{T} \left( \log \rho_{q_t, p_t} - \lambda_{q_t, p_t} + y_t \log \lambda_{q_t, p_t} - \log(y_t!) \right) \right]$$

$$\propto E_{\text{post}} \left[ \sum_{i=1}^{m} 1[q_0 = i] \log \pi_i + \sum_{t=1}^{T-1} \sum_{i=1}^{m} \sum_{j=1}^{m} 1[q_t = i, q_{t+1} = j] \log \tau_{q_t, q_j} \right.$$ 

$$+ \sum_{t=1}^{T} \left( \sum_{i=1}^{m} \sum_{j=1}^{k} 1[q_t = i, p_t = j] \log \rho_{i,j} - \lambda_{q_t, p_t} + y_t \sum_{i=1}^{m} \sum_{j=1}^{k} 1[q_t = i, p_t = j] \log \lambda_{q_t, p_t} \right) \bigg].$$

Hence, we will need to compute the following expectations:

- $l_i := E_{\text{post}} 1[q_0 = i] = P(Q_0 = i|Y)$,
- $m_{i,j} := \sum_{t=1}^{T-1} P(Q_t = i, Q_{t+1} = j|Y)$,
- $n_{i,j} := \sum_{t=1}^{T} P(Q_t = i, P_t = j|Y)$,
- $o_{i,j} := \sum_{t=1}^{T} y_t P(Q_t = i, P_t = j|Y)$

(c) We now show how to construct an efficient dynamic program that compute these expectations, based on the alpha-beta recursion.

We first sum over the paths traversing an emission chains. For $t \in \{1, \ldots, T\}$ (this one can be done in any order), let:

$$L_t(q) := \sum_{p=1}^{k} \rho_{q,p} \nu_{q,p,y}.$$

Next, we do a backward pass through the chain, and let, for $t = T, T-1, \ldots, 1$,

$$M_t(q) := \left\{ \begin{array}{ll} \sum_{q'=1}^{m} \tau_{q', q} L_{t+1}(q') M_{t+1}(q') & \text{for } t \neq T \\ 1 & \text{otherwise} \end{array} \right..$$

See the figure below for a pictorial description of which parts of the graph is summed over, and how the recursion is carried.

A similar forward pass is also performed, this time for $t = 1, 2, \ldots, T$:

$$N_t(q) := \left\{ \begin{array}{ll} \sum_{q'=1}^{m} \tau_{q', q} N_{t-1}(q') L_{t-1}(q') & \text{for } t \neq 1 \\ \pi_q & \text{otherwise} \end{array} \right..$$

Finally, these quantities can be used to find the required expectation. For example:

$$P(Q_t = q, Q_{t+1} = q'|Y) \propto N_t(q) L_t(q) \tau_{q,q'} L_{t+1}(q') M_{t+1}(q')$$

$$P(Q_t = q, P_t = p|Y) \propto N_t(q) M_t(q) \rho_{q,p} \nu_{q,p,y}.$$

(d) Using the fact that the model under study is an exponential family (provided we restrict the parameters of the multinomials to be in the open interval $(0, 1)$), we get that the updates for the
M-step are:

\[
\hat{\tau}_{q,q'} := \frac{m_{q,q'}}{\sum_{q''} m_{q,q''}}
\]
\[
\hat{\rho}_{q,p} := \frac{n_{q,p}}{\sum_{p'} n_{q,p'}}
\]
\[
\hat{\lambda}_{q,p} := \frac{o_{q,p}}{n_{q,p}}
\]
\[
\hat{\pi}_q := l_q.
\]

2. (EM for HMMs)

(a) An implementation in python is included in Appendix A.

Note that depending on the programming environment and floating point library used, computations of the probabilities for the Poisson likelihood may be badly behaved numerically. Taking the logs helps in this case. Moreover, in applications the chain under study can be much longer (\(T >> 50\)) and it is often necessary to perform the entire forward-backward algorithm in log-space to avoid underflows.

(b) The log-likelihood can be computed efficiently using the datastructures resulting from forward-backward.

As expected, the training likelihood increases monotonically. The test likelihood, on the other hand, reaches a peak after three iterations and then decreases, due to overfitting.

The true parameters (those used to generate the data) were:

\textbf{Lambda}:

0.5 17.0
35.0 125.0
185.0 255.0

\textbf{Rho}:
Figure 1: Log-likelihood of the HMM model on training and test data.

0.5 0.5
0.5 0.5
0.5 0.5

Tau:
0.05 0.9 0.05
0.05 0.05 0.9
0.9 0.05 0.05

(c) Performance of the iid model is poorer on the training set since the model was generated using a sequence model. However, the HMM model overfit the data (see Figure 2).

3. (EM)

(a) We call \( I_t(q_t) \) the sum of the probabilities of the states in the hidden subtree rooted at node indexed \( t \) and with state \( q \) at that state. We call \( O_t(q_t) \) the sum of the probabilities of the states in the complement of this subtree. We have the following recursions: for \( I_t \),

\[
I_t(q_t) := \begin{cases} 
\sum_{q_u} \sum_{q_v} \tau_{q_v,q_u} \sigma_{q_v,y_t} \nu_{q_v,y_t} \nu_{q_u,y_u} I_v(q_v)I_u(q_u) & \text{for } t \text{ non-leaf} \\
1.0 & \text{otherwise}
\end{cases}
\]  

where \( u, v \) are the children of \( t \), \( \tau \) is the transition potential, \( y_t \) is the observation emitted by node \( t \) and \( \nu_{q,y} \), the normal density with mean and variance \( \mu_q, \sigma_q^2 \) evaluated at \( y \); for \( O_t \),

\[
O_t(q_t) := \begin{cases} 
\sum_{q_p} \sum_{q_b} \tau_{q_b,q_p} \sigma_{q_b,y_t} \nu_{q_b,y_t} \nu_{q_p,y_p} O_p(q_p)O_b(q_b) & \text{for } t \text{ not root} \\
1.0 & \text{otherwise}
\end{cases}
\]

where \( p \) is the parent of \( t \) and \( b \), its brother. Notice that \( I \) must be computed bottom up, while \( O \) must be computed top down and after \( I \) is done.
Figure 2: Log-likelihoods of HMM and mixture model on training and test data.

(b) Note that the expected complete loglikelihood decompose and the derivation of the M step for \( \mu_1, \sigma^2_1 \) is identical than that for an HMM (see text). We show the MLE for \( \alpha \): we want to find \( \alpha \in (0, 1) \) that maximizes:

\[
\sum_{t \neq \text{root}} \left( \frac{x_t x_{p(t)} + 1}{2} \log \alpha + \frac{1 - x_t x_{p(t)}}{2} \log(1 - \alpha) \right),
\]

where \( p(t) \) is the parent of \( t \). Taking the derivative with respect to \( \alpha \) and letting \( A := \sum_{t \neq \text{root}} \frac{x_t x_{p(t)} + 1}{2}, B := \sum_{t \neq \text{root}} \frac{1 - x_t x_{p(t)}}{2}, \) we get that \( \hat{\alpha} = \frac{A}{A + B} \).

(c) Using the initialization:

\[
\begin{align*}
\text{Alpha: } & 0.5 \\
\text{Mu: } & [-2.0, 2.0] \\
\text{sigma^2: } & [2.0, 2.0]
\end{align*}
\]

we obtain, after 10 iterations:

\[
\begin{align*}
\text{Alpha: } & 0.14618968397080356 \\
\text{Mu: } & [-1.1265550498403933, 1.0268861488781378] \\
\text{sigma^2: } & [0.8706558345196702, 0.9556590261995657]
\end{align*}
\]

while the true parameters were \( \alpha = 0.1, \mu = (-1.0, 1.0) \) and \( \sigma^2 = (1.0, 1.0) \). The code is included in Appendix B.

(d) Replace sums by maxima in the bottom up pass (Equation 1) and keep track of the child states that produce a maximizing configuration in the bottom-up pass. This is essentially just the max-product algorithm.

(e) The true hidden state values are presented in Figure 3. The max configuration obtained using the parameter obtained from EM are presented in Figure 4. Only 25 (10\%) of the hidden states were misclassified, mostly in the last level.
Figure 3: True hidden states.

Figure 4: Maximum configuration.
A Code for problem 2

from numpy import *
import math
import copy as cp

class Poisson:
    def __init__(self, param):
        self.param = param

    def P(self, k):
        def logfac(n):
            return sum(log(range(1, n+1)))
        log_p = k * log(self.param) - self.param - logfac(k)
        return exp(log_p)

    def ML(self, data, weights):
        self.param = sum(data * weights) / sum(weights)

def DeepCopy(arr):
    elts = map(cp.deepcopy, arr.flat)
    return array(elts).reshape(arr.shape)

class H2MM:
    """A general class for representing HMMs whose observation distributions are themselves mixtures of exponential families."""
    def __init__(self, initial, mc_transition, mix_transition, families):
        """initial is a vector specifying the initial distribution for the Markov chain."

        mc_transition is a matrix specifying the transition probabilities for the Markov chain.

        mix_transition is a matrix whose ijth entry gives the probability of being in mixture component j given that the Markov chain is in state i.

        families is a matrix of exponential families with a row for each MC state and a column for each mixture component."

        self.initial = copy(initial)
        self.mc_transition = copy(mc_transition)
        self.mix_transition = copy(mix_transition)
        self.families = DeepCopy(families)

        n_mc_states = initial.shape[0]
        n_mix_states = mix_transition.shape[1]

        self.mc_states = range(n_mc_states)
        self.mix_states = range(n_mix_states)
```python
assert(n_mc_states == self.mc_transition.shape[0])
assert(n_mc_states == self.mc_transition.shape[1])
assert(n_mc_states == self.mix_transition.shape[0])
assert(n_mc_states == self.families.shape[0])
assert(n_mix_states == self.families.shape[1])

def ObservationProbs(self, datum):
    
    """Returns an array of conditional probabilities P(y | q = i) for all states i in the Markov chain."""
    full_probs = self.FullObservationProbs(datum)
    ret = zeros(len(self.mc_states))
    for i in self.mc_states:
        ret[i] = dot(full_probs[i, :], self.mix_transition[i, :])
    return ret

def FullObservationProbs(self, datum):
    
    """Returns a 2d array of conditional probabilities P(y | q = i, p = j) for all states i in the Markov chain and mixture components j."""
    ret = [i.P(datum) for i in self.families.flat]
    return array(ret).reshape((len(self.mc_states), len(self.mix_states)))

def Alpha(self, data):
    """Performs a (normalized) alpha-recursion given the data.

    Returns a ndata x nstates matrix, whose (t,i) element is (up to a constant depending on the data) p(y_1, \dots, y_{t-1}, q_t).
    """
    alpha = zeros((len(data), len(self.mc_states)))
    alpha[0, :] = self.initial
    log_normalization = log(sum(alpha[0, :]))
    alpha[0, :] /= sum(alpha[0, :])
    for t in range(1, len(data)):
        data_prob = self.ObservationProbs(data[t - 1])
        alpha[t, :] = dot(self.mc_transition.T, alpha[t - 1, :] * data_prob)
        log_normalization += log(sum(alpha[t, :]))
        alpha[t, :] /= sum(alpha[t, :])
        # p(y) = \sum_i p(y, q_t = i) = \sum_i alpha_t(i) p(y_t | q_t)
        # Note that this is different from the formula in the book since we've modified alpha to be p(y_1, \dots, y_{t-1}, q_t) instead of p(y_1, \dots, y_t, q_t). Also, our alpha is normalized.
        data_prob = self.ObservationProbs(data[-1])
        self.log_likelihood = log_normalization + log(sum(alpha[t, :] * data_prob))
    return alpha

def Beta(self, data):
```
"""Performs a (normalized) beta-recursion given the data.

Returns a ndata x nstates matrix, whose (t,i) element is (up to a constant depending on the data) p(y_{t+1}, \cdots, y_T | q_t = i).
"""

beta = zeros((len(data), len(self.mc_states)))
beta[-1, :] = ones((1, len(self.mc_states))) / len(self.mc_states)

for t in range(len(data)-2, -1, -1):
    data_prob = self.ObservationProbs(data[t+1])
    beta[t, :] = dot(self.mc_transition, (beta[t+1, :] * data_prob))
    beta[t, :] /= sum(beta[t, :])

return beta

def Inference(self, data):
    """Finds the probabilities \gamma_t(i, j) = P(q_t = i, p_t = j | y) \xi_{t,t+1}(i, j) = P(q_{t+1} = j, q_t = i, y)"
    alpha = self.Alpha(data)
    beta = self.Beta(data)

    gamma = zeros((len(data), len(self.mc_states), len(self.mix_states)))
    xi = zeros((len(data)-1, len(self.mc_states), len(self.mc_states)))

    for t in range(len(data)):
        data_prob = self.ObservationProbs(data[t])
        full_data_prob = self.FullObservationProbs(data[t])
        gamma[t, :] = (array(self.mix_transition) * full_data_prob
                        * array(alpha[t, :] * beta[t, :], ndmin = 2).T)
        gamma[t, :] /= sum(gamma[t, :])

        if t < len(data) - 1:
            next_data_prob = self.ObservationProbs(data[t+1])
            xi[t, :] = outer(alpha[t, :] * data_prob, beta[t+1, :] *
                            next_data_prob) * array(self.mc_transition)
            xi[t, :] /= sum(xi[t, :])

    return (gamma, xi)

def UpdateParameters(self, data):
    """Given some data, updates the current parameters by taking one step under the EM algorithm.""
    (gamma, xi) = self.Inference(data)
    # We were asked not to estimate the initial distribution.
# self.initial = sum(gamma[0, :], axis=1)
self.mc_transition = sum(xi, axis=0)
self.mix_transition = sum(gamma, axis=0)
for i in self.mc_states:
    self.mc_transition[i, :] /= sum(self.mc_transition[i, :])
for j in self.mix_states:
    self.families[i,j].ML(data, gamma[:, i, j])

def LogLikelihood(self, data):
    self.Alpha(data)
    return self.log_likelihood

class MixtureModel:
    def __init__(self, families, mixture):
        self.families = DeepCopy(families)
        self.mixture = copy(mixture)
        assert(len(families) == len(mixture))

def LogLikelihood(self, data):
    def P(k):
        return sum([f.P(k) * self.mixture[i]
                    for (i, f) in enumerate(self.families)])
    return sum([log(P(k)) for k in data])

def HiddenProbs(self, data):
    """Returns an array of dimension t x k, whose i,j entry is p(q = j | data[i])."""
    ret = zeros((len(data), len(self.mixture)))
    for (i, k) in enumerate(data):
        ret[i, :] = array([f.P(k) for f in self.families]) * self.mixture
        ret[i, :] /= sum(ret[i, :])
    return ret

def UpdateParameters(self, data):
    weights = self.HiddenProbs(data)
    self.mixture = sum(weights, axis = 0)
    self.mixture /= sum(self.mixture)
    for (i, f) in enumerate(self.families):
                f.ML(data, weights[:, i])

def Main():
    # Fitting and testing modified HMMs
    f = open("data/hw4-2.data", "r")
    data = [int(x) for x in f.readlines()]
    f = open("data/hw4-2.test", "r")
    test_data = [int(x) for x in f.readlines()]
initial = ones(3) / 3
mc_transition = ones((3, 3)) / 3
mix_transition = ones((3, 2)) / 2
families = array([ [Poisson(1), Poisson(5)],
                  [Poisson(50), Poisson(100)],
                  [Poisson(200), Poisson(300)]])
h = H2MM(initial, mc_transition, mix_transition, families)

for t in range(10):
    print("Training likelihood: %f" % h.LogLikelihood(data))
    print("Test likelihood: %f" % h.LogLikelihood(test_data))
h.UpdateParameters(data)

# Fitting and testing mixture model.
m = MixtureModel(families.flatten(), ones(6) / 6)
for t in range(10):
    print("Training likelihood: %f" % m.LogLikelihood(data))
    print("Test likelihood: %f" % m.LogLikelihood(test_data))
m.UpdateParameters(data)

Main()

**B Code for problem 3**

from numpy import *
import math
import copy as cp
def DeepCopy(arr):
    elts = map(cp.deepcopy, arr.flat)
    return array(elts).reshape(arr.shape)

class Gaussian:
    def __init__(self, mean, var):
        self.mean = mean
        self.var = var

    def P(self, x):
        return exp(-(x - self.mean)**2 / (2*self.var)) / (sqrt(2 * pi * self.var))

    def ML(self, data, weights):
        self.mean = sum(data * weights) / sum(weights)
        self.var = sum(weights * ((data - self.mean)**2)) / sum(weights)

class HiddenTree:
    def __init__(self, transition, observations):
        self.transition = transition
        self.observations = DeepCopy(observations)
def Parent(self, i):
    return (i-1)/2

def Brother(self, i):
    return i + 2*(i % 2) - 1

def LeftChild(self, i):
    return i*2 + 1

def RightChild(self, i):
    return i*2 + 2

def Transition(self, i, j):
    if i == j:
        return self.transition
    return 1-self.transition

def Observation(self, i, y):
    return self.observations[i].P(y)

def Beta(self, data):
    """Do a backwards recursion to compute p(q_t, descendents).""
    beta = zeros((len(data), 2))
    log_normalization = log(2)
    for i in range(len(data)/2, len(data)):
        beta[i,:] = ones(2) / 2
    for i in range(len(data)/2 - 1, -1, -1):
        for q in [0, 1]:
            beta[i,q] = sum([self.Transition(q,r) * self.Transition(q,s) *
                             self.Observation(r,data[self.LeftChild(i)]) *
                             self.Observation(s,data[self.RightChild(i)]) *
                             beta[self.LeftChild(i),r] * beta[self.RightChild(i),s]
                          for r in [0, 1] for s in [0, 1]])
            log_normalization += log(sum(beta[i,:]))
        beta[i,:] /= sum(beta[i,:])
    self.log_likelihood = log_normalization + log(beta[0,0] * self.Observation(0, data[0])
                                                  + beta[0,1] * self.Observation(1, data[0]))
    return beta

def Alpha(self, data, beta):
    """Do a forwards recursion to compute p(q_t, non-descendents).""
    alpha = zeros((len(data), 2))
    alpha[0,:] = ones(2) / 2
    for i in range(1, len(data)):
for q in [0, 1]:
    alpha[i,q] = sum([self.Transition(p,q) * self.Transition(p,r) * 
                      self.Observation(p, data[self.Parent(i)]) * 
                      self.Observation(r, data[self.Brother(i)]) * 
                      alpha[self.Parent(i),p] * beta[self.Brother(i),r] 
                      for p in [0, 1] for r in [0, 1]])

    alpha[i,:] /= sum(alpha[i,:])
return alpha

def LogLikelihood(self, data):
    self.Beta(data)
    return self.log_likelihood

def Inference(self, data):
    """Finds
    gamma_t(q) = p(q_t = q | y)
    for each t and
    xi_t(q,r) = p(q_p = q, q_t = r | y)
    for each non-root t (where p is the parent of t).""
    beta = self.Beta(data)
    alpha = self.Alpha(data, beta)
    gamma = zeros((len(data), 2))
    xi = zeros((len(data)-1, 2, 2))
    for t in range(0, len(data)):
        data_probs = array([self.Observation(0, data[t]), self.Observation(1, data[t])])
        gamma[t,:] = alpha[t,:] * beta[t,:] * data_probs
        gamma[t,:] /= sum(gamma[t,:])
        if t > 0:
            p = self.Parent(t)
            b = self.Brother(t)
            for qp in [0,1]:
                # p(y_brother's subtree | q_p = p)
                brother_subtree = sum([self.Observation(qb, data[b]) * 
                                        self.Transition(qp, qb) * beta[b,qb] 
                                        for qb in [0, 1]])
                for qt in [0,1]:
                    xi[t-1,qp, qt] = (self.Observation(qt, data[t]) * self.Observation(qp, data[t] * 
                                          self.Transition(qp, qt) * 
                                          beta[t,qt] * alpha[p,qp] * brother_subtree)
                xi[t-1,:] /= sum(xi[t-1,:])
    return (gamma, xi)
def MAP(self, data):
    """Finds the maximum-probability assignment to the hidden states.""

    # Modified backwards recursion (with max instead of sum):
    beta = zeros((len(data), 2))
    max_child_config = zeros((len(data)/2, 2, 2))

    for i in range(len(data)/2, len(data)):
        beta[i,:] = ones(2) / 2
    for i in range(len(data)/2 - 1, -1, -1):
        for q in [0, 1]:
            child_probs = [self.Transition(q,r) * self.Transition(q,s) *
                            self.Observation(r,data[self.LeftChild(i)]) *
                            self.Observation(s,data[self.RightChild(i)]) *
                            beta[self.LeftChild(i),r] * beta[self.RightChild(i),s]
                            for r in [0, 1] for s in [0, 1]]
            beta[i,q] = max(child_probs)
            max_child_config[i,q,:] = unravel_index(argmax(child_probs), (2,2))
            beta[i,:] /= sum(beta[i,:])

    map = zeros(len(data))
    def FindMAP(root, state):
        map[root] = state
        if root < len(data)/2:
            child_states = max_child_config[root,state,:]
            FindMAP(self.LeftChild(root), child_states[0])
            FindMAP(self.RightChild(root), child_states[1])
    FindMAP(0, argmax(beta[0,:]))
    return map

def UpdateParameters(self, data):
    """Performs one step of the EM algorithm.""
    gamma, xi = self.Inference(data)
    self.observations[0].ML(data, gamma[:,0])
    self.observations[1].ML(data, gamma[:,1])
    transitions = sum(xi, axis=0) / xi.shape[0]
    self.transition = transitions[0,0] + transitions[1,1]

def Main():
    f = open("data/hw4-3.data", "r")
    data = [double(x) for x in f.readlines()]
    gs = array([Gaussian(-2, 2), Gaussian(2, 2)])
    t = HiddenTree(0.5, gs)
    print("Likelihood %f:" % t.LogLikelihood(data))
for i in range(10):
    t.UpdateParameters(data)
    print("Likelihood %f: " % t.LogLikelihood(data))
    print("alpha: %f" % (1 - t.transition))
    print("mu_1: %f, sigma_1^2" % t.observations[0].mean, t.observations[0].var)
    print("mu_2: %f, sigma_2^2" % t.observations[1].mean, t.observations[1].var)
    print(t.MAP(data))

Main()