My notes on Hidden Markov Models and the three fundamental algorithms

Let's assume we have a hidden Markov model: that is

1. A markov chain M
2. An initial probability distribution π, where π_S is the probability that $M_0 = S$.
3. A set of observations $\mathcal{O}$, and a set of observation probabilities $P(O|M)$, where $P(O|S)$ is the probability of observing O given that the markov chain is in state S. Furthermore, we assume $PO(O_t|M, O_0, \ldots, O_{t-1}) = P(O|S)$ (O_t only depends on the current state S).

Basically, we want algorithms that let us answer any question about this model given some observations.

First, let's code a discrete markov chain model.

```
import numpy as np
# a HMM is a Markov chain, a set of observations, and a set of observation probabilities
class HMM:
    def __init__(self, transition_matrix, observation_matrix):
        self.transition_matrix = transition_matrix  # P(M_t | M_{t-1})
        self.observation_matrix = observation_matrix  # P(O_t | M_t)
    def p_obs(self, observation, state):
        return self.observation_matrix[state][observation]  # P(O_t | M_t)
    def p_trans(self, state_in, state_out):
        return self.transition_matrix[state_in][state_out]  # P(M_t | M_{t-1})
```

The likelihood of a sequence of observations

Problem

Given a hidden markov model and a sequence of observations $\mathcal{O}_0, \dots, \mathcal{O}_t$, what is $P(\mathcal{O}_0, \dots, \mathcal{O}_t)$?

Forward algorithm

The idea of the forward algorithm is simple.

Consider the slightly modified question of computing

$$P(S_t; O_0, \dots, O_t)$$

We can compute this inductively,

$$P(S_t; O_0, \dots, O_t) = \sum P(S_{t-1}; O_0, \dots, O_{t-1}) P(O_t | S_t) P(S_t | S_{t-1})$$

Now we can sum over all possible S_t to get our desired probability:

$$P(O_0, \dots, O_t) = \sum_{S_t} P(S_t; O_0, \dots, O_t)$$

In code,

```
def forward_update_probs(obs,prev_probs,hmm):
    m = hmm.transition_matrix
    o = hmm.observation_matrix
    return m @ prev_probs * o[:,obs]

def forward(obs_seq, hmm, initial_probs):
    probs = initial_probs * hmm.observation_matrix[:, obs_seq[0]]
    for obs in obs_seq[1:]:
        probs = forward_update_probs(obs, probs, hmm)
    return np.sum(probs)
```

So for example,

```python
m = np.array([[0.9, 0.1], [0.2, 0.8]])  # transition matrix
o = np.array([[0.8, 0.2], [0.1, 0.9]])  # observation matrix
hmm = HMM(m, o)
initial_probs = np.array([0.5, 0.5])  # initial probabilities
obs_seq = [0, 1, 0]  # sequence of observations
likelihood = forward(obs_seq, hmm, initial_probs)
print(f"Likelihood of the sequence {obs_seq} is {likelihood}")

# check likelihoods sum to one
# find most likely sequence
from itertools import product
total_likelihood = 0
most_likely_seq = None
highest_likelihood = 0
for obs_seq in product([0,1],repeat=3):
    likelihood = forward(obs_seq, hmm, initial_probs)
    if likelihood > highest_likelihood:
        most_likely_seq = obs_seq
        highest_likelihood = likelihood
    total_likelihood += likelihood
print(f"Total likelihood of all sequences is {total_likelihood}, should be 1.0")
print(f"Most likely sequence is {most_likely_seq} with likelihood {highest_likelihood}")
```

```
Likelihood of the sequence [0, 1, 0] is 0.07130000000000002
Total likelihood of all sequences is 1.0000000000000002, should be 1.0
Most likely sequence is (1, 1, 1) with likelihood 0.2628000000000001
```

Most likely hidden state sequence

Problem

Given a hidden markov model and a sequence of observations $\mathcal{O}_0, \dots, \mathcal{O}_t$, what is the most likely sequence of hidden states $S_0, \dots, S_t$?

The algorithm is very similar to the forward algorithm above. We compute

$$\sup_{S_0,\dots,S_{t-1}} P(S_0, \dots, S_{t-1}, S_t; O_0, \dots, O_t)$$

which has an inductive formula:

$$\begin{aligned} &\sup_{S_0,\dots,S_{t-1}} P(S_0,\dots,S_{t-1},S_t;O_0,\dots,O_t) \\ &= \sup_{S_{t-1}} \sup_{S_0,\dots,S_{t-2}} P(S_0,\dots,S_{t-2},S_{t-1};O_0,\dots,O_{t-1})P(S_t|S_{t-1})P(O_t|S_t) \end{aligned}$$

From there, the actual program structure is more or less the same.

```python
def viterbi_update_probs(obs, prev_probs, hmm):
    m = hmm.transition_matrix
    o = hmm.observation_matrix
    new_probs = np.zeros_like(prev_probs)
    new_states = np.zeros_like(prev_probs, dtype=int)
    for i in range(len(prev_probs)):
        choices = o[i, obs] * m[: , i] * prev_probs
        newstate = np.argmax(choices)
        new_probs[i] = choices[newstate]
        new_states[i] = newstate
    return new_probs, new_states

    # find most likely state
def viterbi(obs_seq, hmm, initial_probs):
    prob_list = [initial_probs]
    state_list = []
    for obs in obs_seq:
        probs, state_indices = viterbi_update_probs(obs, prob_list[-1], hmm)
        prob_list.append(probs)
        state_list.append(state_indices)
    print(state_list)
    # backtrack to find most likely sequence
    state_sequence = []
    final_state = np.argmax(prob_list[-1])
    p = prob_list[-1][final_state]
    state_sequence.append(final_state)
    for states in reversed(state_list[:-1]):
        final_state = states[final_state]
        state_sequence.append(final_state)
    most_likely_states = list(reversed(state_sequence))
    return most_likely_states, p
```

Again, let's do a quick example:

```python
m = np.array([[0.9, 0.1], [0.2, 0.8]])  # transition matrix
o = np.array([[0.8, 0.2], [0.1, 0.9]])  # observation matrix
hmm = HMM(m, o)
initial_probs = np.array([0.5, 0.5])  # initial probabilities
obs_seq = [0,1,1]

first_step_probs = viterbi_update_probs(obs_seq[0], initial_probs, hmm)
print(f"First step probabilities: {first_step_probs[0]}")
print(f"These should be of the same scale as  P(0):{np.sum(initial_probs*o[:,0])} ")

seq, p= viterbi(obs_seq,hmm,initial_probs)
# normalise the probability correctly
p = p/forward(obs_seq,hmm,initial_probs)
print(f"Most likely sequence: {seq}")
print(f"Has probabiliy {p}")
```

```
First step probabilities: [0.36 0.04]
These should be of the same scale as  P(0):0.45
[array([0, 1]), array([0, 0]), array([0, 1])]
Most likely sequence: [np.int64(0), np.int64(0), np.int64(1)]
Has probabiliy 0.2196610169491525
```

Learning an HMM

Problem

Given only the state space, can we learn both the transition matrix $P(M_t|M_{t-1})$ and the observation matrix $P(O_t|M_t)$?

The algorithm is a pretty simple, iterative one. First, we choose a prior model θ. Then we update the model as follows.

1. First we calculate $P(O_0, \dots, O_{t-1}, M_t|\theta)$ and $P(O_t, \dots, O_t|M_t, \theta)$.
2. Using these two ‘forward’ and ‘backward’ probabilities, we can compute new estimates for $P(M_t|M_{t-1})$ and $P(O_t|M_t)$. Exact formulas for $P(M_t|M_{t-1})$ and $P(O_t|M_t)$ are easy to write down, but a little wordy; I just write them out in code below.
3. Repeat until convergence.

```python
def forward_estimates(hmm, initial_dist,obs):
    probs = [initial_dist * hmm.observation_matrix[:, obs[0]]]
    for i in range(1, len(obs)):
        new_prob = hmm.observation_matrix[:, obs[i]] * hmm.transition_matrix @ probs[-1]
        probs.append(new_prob)
    # scale to prevent accumulation of numerical error
    probs = [p / np.sum(p) for p in probs]
    return np.array(probs)

def backward_estimates(hmm, obs):
    obs = list(reversed(obs))
    probs = [ [1.0]*len(hmm.observation_matrix) ]
    for o in obs[1:]:
        new_probs = hmm.observation_matrix[:, o] * hmm.transition_matrix @ probs[-1]
        probs.append(new_probs)
    probs = [p / np.sum(p) for p in reversed(probs)]
    # scale to prevent accumulation of numerical error
    return np.array(probs)
def gammaVec(fwd, bkwd):
    gamma = [fwd[i] * bkwd[i] / np.sum(fwd[i] * bkwd[i]) for i in range(len(fwd))]
    return np.array(gamma)
def xiMat(hmm, fwd,bkwd):
    # output is a rank three tensor
    # one leg = time-1, two legs = hidden states
    xi = np.zeros((len(fwd)-1, len(fwd[0]), len(fwd[0])))
    for t in range(len(fwd)-1):
        for i in range(len(fwd[0])):
            for j in range(len(fwd[0])):
                xi[t,i,j] = fwd[t][i] * bkwd[t+1][j] * hmm.transition_matrix[i,j] * hmm.obse
        xi[t,:,:] /= np.sum(xi[t,:,:])
    return xi

def updateHMM(hmm, xiM, gammaV):
    new_dist = gammaV[0]
    new_tstn_matrix = np.zeros_like(hmm.transition_matrix)
    for i in range(len(new_tstn_matrix)):
        for j in range(len(new_tstn_matrix)):
            new_tstn_matrix[i,j] = np.sum(xiM[:,i,j]) / np.sum(gammaV[:-1,i])
    new_obs_matrix = np.zeros_like(hmm.observation_matrix)
    for i in range(len(new_obs_matrix)):
        for j in range(len(new_obs_matrix[0])):
            delta = [obs[k] == j for k in range(len(obs))]
            new_obs_matrix[i,j] = np.sum(gammaV[:,i] * delta) / np.sum(gammaV[:,i])
    return HMM(new_tstn_matrix, new_obs_matrix), new_dist

def train_step_HMM(hmm, obs, initial_dist):
    fwd = forward_estimates(hmm, initial_dist, obs)
    bkwd = backward_estimates(hmm, obs)
    gammaV = gammaVec(fwd, bkwd)
    xiM = xiMat(hmm, fwd, bkwd)
    return updateHMM(hmm, xiM, gammaV)

def train_HMM(hmm, obs, initial_dist, n_steps=10):
    for _ in range(n_steps):
        hmm, initial_dist = train_step_HMM(hmm, obs, initial_dist)
    return hmm, initial_dist

def random_HMM(n_states, n_obs):
    transition_matrix = np.random.rand(n_states, n_states)
    transition_matrix /= transition_matrix.sum(axis=1, keepdims=True)
    observation_matrix = np.random.rand(n_states, n_obs)
    observation_matrix /= observation_matrix.sum(axis=1, keepdims=True)
    return HMM(transition_matrix, observation_matrix)
```

Example of the HMM learning algorithm

OK, let's try to generate a random HMM and some observations.

```
def generate_observations(hmm, initial_probs, samples):
    obs = []
    num_states = hmm.transition_matrix.shape[0]
    state = np.random.choice(num_states, p=initial_probs)
    for _ in range(samples):
        observation = np.random.choice(len(hmm.observation_matrix[state]), p=hmm.observation_
        obs.append(observation)
        state = np.random.choice(num_states, p=hmm.transition_matrix[state])
    return obs

n_states = 3
n_obs = 5
true_hmm = random_HMM(n_states, n_obs)
true_initial_probs = np.random.rand(n_states)
true_initial_probs /= true_initial_probs.sum()

n_samples = 300
obs = generate_observations(true_hmm, true_initial_probs, n_samples)
print(f"First twenty observations: {np.array(obs)[:20]}")
```

```
First twenty observations: [1 1 1 3 1 1 2 1 4 3 1 3 2 2 3 1 4 1 3 2]
```

Now, let's train an HMM on our set of observations.

```
hmm = random_HMM(n_states, n_obs)
n_steps = 100
hmm, initial_probs = train_HMM(hmm, obs, true_initial_probs, n_steps)
print(f"Training with #steps = {n_steps}, #states = {n_states},  #observables = {n_obs}, #obs
print(f"True transition matrix:\n{true_hmm.transition_matrix}")
print(f"Estimated transition matrix:\n{hmm.transition_matrix}")
print(f"Mean squared error:{np.mean((true_hmm.transition_matrix - hmm.transition_matrix)**2)
print(f"True observation matrix:\n{true_hmm.observation_matrix}")

print(f"Estimated observation matrix:\n{hmm.observation_matrix}")
print(f"Mean squared error:{np.mean((true_hmm.observation_matrix - hmm.observation_matrix)**
```

```
Training with #steps = 100, #states = 3,  #observables = 5, #observations = 300
True transition matrix:
[[0.35258663 0.38438572 0.26302765]
 [0.32964946 0.33422157 0.33612896]
 [0.507411   0.39651089 0.09607811]]
Estimated transition matrix:
[[0.53921208 0.11250364 0.34825309]
 [0.37217434 0.22127698 0.40657767]
 [0.15779829 0.53436863 0.30783563]]
Mean squared error:0.035735024569017025
True observation matrix:
[[0.41076738 0.13472827 0.28866819 0.15949621 0.00633995]
 [0.08826143 0.28016781 0.17977608 0.2808124  0.17098228]
 [0.13285838 0.28260535 0.27006007 0.09114706 0.22332914]]
Estimated observation matrix:
[[0.25620575 0.28405608 0.20672549 0.16105219 0.09196049]
 [0.26088947 0.27082387 0.21074811 0.16388481 0.09365374]
 [0.26295331 0.26498291 0.21256831 0.16509215 0.09440332]]
Mean squared error:0.01022433772731991
```