Report_final_project_Stoch

Amir Pourmand Final Report Stochastic Processes
Student No: 99210259

Q1-3

There are three main problems in hidden markov models, namely, likelihood, decoding and learning. Here, we want to explain each one of them and also implement them completely in pure python!

You can get a sense of markov model a its properties here.

Q1 - Baum-Weltch Algorithm

We can now calculate the temporary variables, according to Bayes' theorem:

$$\gamma_i(t) = P(X_t = i \mid Y, \theta) = \frac{P(X_t = i, Y \mid \theta)}{P(Y \mid \theta)} = \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N} \alpha_j(t)\beta_j(t)}$$

which is the probability of being in state i at time t given the observed sequence Y and the parameters θ

$$\xi_{ij}(t) = P(X_t = i, X_{t+1} = j \mid Y, \theta) = \frac{P(X_t = i, X_{t+1} = j, Y \mid \theta)}{P(Y \mid \theta)} = \frac{\alpha_i(t)a_{ij}\beta_j(t+1)b_j(y_{t+1})}{\sum_{k=1}^{N}\sum_{w=1}^{N} \alpha_k(t)a_{kw}\beta_w(t+1)b_w(y_{t+1})}$$

which is the probability of being in state i and j at times t and $t+1$ respectively given the observed sequence Y and parameters θ.
The denominators of $\gamma_i(t)$ and $\xi_{ij}(t)$ are the same ; they represent the probability of making the observation Y given the parameters θ.
The parameters of the hidden Markov model θ can now be updated:

$$- \pi_i^* = \gamma_i(1),$$

which is the expected frequency spent in state i at time 1 .

$$- a_{ij}^* = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)},$$

which is the expected number of transitions from state i to state j compared to the expected total number of transitions away from state i. To clarify, the number of transitions away from state i does not mean transitions to a different state j, but to any state including itself. This is equivalent to the number of times state i is observed in the sequence from $t = 1$ to $t = T$ - 1 .

$$- b_i^*(v_k) = \frac{\sum_{t=1}^{T} 1_{y_t = v_k} \gamma_i(t)}{\sum_{t=1}^{T} \gamma_i(t)},$$

where

$$1_{y_t = v_k} = \begin{cases} 1 & \text{if } y_t = v_k \\ 0 & \text{otherwise} \end{cases}$$

The function to implement baum weltch is defined as:

```
    def baum_welch(self, seq, iterations=100):
```

and for each sequence, it would call

```
    def _baum_welch_iteration(self, seq, cons=100):
```

The gist of the algorithm is that γ and ζ are calculated in each step as follows:

```
gamma = [dict([(s, 0) for s in self.transition_probs]) for t in xrange(len(seq))]
        for t in xrange(len(seq)):
            norm = sum(
                [np.exp(forward_probabilities[t][k] + backward_probabilities[t][k] + 1200)
for k in
                 self.transition_probs])
            for j in self.transition_probs:
                gamma[t][j] = np.exp(forward_probabilities[t][j] +
backward_probabilities[t][j] + 1200) / norm
```

This is for gamma which just get a logarithm of formula we said.

```
zeta = [dict([(s, dict([(s, 0) for s in self.transition_probs])) for s in
self.transition_probs]) for i in
                xrange(len(seq))]
        for t in xrange(len(seq) - 1):
            norm = sum([sum([
                np.exp(forward_probabilities[t][i] + self._log_transition_probs[i][j]
                       + backward_probabilities[t + 1][j] + self._log_emission_probs[j]
[seq[t + 1]] + 1200)
                for j in self.transition_probs[i]]) for i in self.transition_probs])

            for i in self.transition_probs:
                for j in self.transition_probs:
                    if j in self._log_transition_probs[i]:

                        zeta[t][i][j] = np.exp(forward_probabilities[t][i] +
self._log_transition_probs[i][j]
                                               + backward_probabilities[t + 1][j] +
self._log_emission_probs[j][
                                                   seq[t + 1]] + 1200) / norm
                    else:
                        zeta[t][i][j] = 0
```

and this is for updating the zeta.

> After roughly 20 states, the state probabilities will converge. The result of the algorithm is in 3 files, named `emission_probs.json`, `initial_probs.json`, `transition_probs.json`.

Q2- Forward-backward algorithm (or just forward algorithm)

We want to answer the question: What is the probability that a model generated a sequence of observations? In other words, we want to calculate the probabilty of our observations given that we know our hidden markov model parameters completely. This is formulated by this matematical notation.

$$P(O|\lambda) = ?$$

Since observations depend on state sequences, we should bring state sequences into equation

$$P(O|\lambda) = \sum_Q P(O,Q|\lambda) = \sum_Q P(O|Q,\lambda)P(Q|\lambda)$$

We can easily get $P(O|Q,\lambda) = \prod_i^N P(O_i|Q_i,\lambda)$ and $P(Q|\lambda) = \prod_i^N a_{i-1,i}$.

But this is infeasible. That's because enumerating all state space sequences would be impossible. There are lots of them! So, we will use dynamic programming approach. We need to define parameter α. $\alpha_t(i)$ defines the probability of being in state i in time t and oberving t first observations. For example, $\alpha_2(3) = P(O_1, O_2, q_2 = S_3|\lambda)$.

$$\alpha_t(i) = P(O_1, O_2, O_3, ..., O_t, q_t = S_i|\lambda)$$

We need to define it with recursively (this is helpful in dynamic programming):

$$\alpha_{t+1}(i) = P(O_1, O_2, O_3, ..., O_t, O_{t+1}, q_{t+1} = S_i|\lambda)$$
$$= \sum_{k=1}^{S} P(O_1, O_2, O_3, ..., O_t, O_{t+1}, q_t = S_k, q_{t+1} = S_i|\lambda)$$
$$= \sum_{k=1}^{S} P(O_1, ..., O_t, q_t = S_k|\lambda) P(O_{t+1}, q_{t+1} = S_i|q_t = S_k, \lambda)$$
$$= \sum_{k=1}^{N} \alpha_t(k) P(q_{t+1} = S_i|q_t = S_k) P(O_{t+1}|q_{t+1} = S_i)$$
$$= [\sum_{k=1}^{N} \alpha_t(k) a_{k,i}] b_{q_{t+1}}(O_{t+1})$$

We also set $a_{0,1} = \pi$.

Finally, in order to calculate $P(O|\lambda)$

$$P(O|\lambda) = P(O_1, O_2, O_3, ..., O_T|\lambda)$$
$$= \sum_{i=1}^{N} P(O_1, O_2, ..., O_T, q_t = S_i|\lambda)$$
$$= \sum_{i=1}^{N} a_T(i)$$

This algorithm is simply run with forward backward function in my code. The results of the algorithm is in file `forward_backward_output.txt`.

Q3 - Viterbi Algorihm

We need to define a paramater called β that will help us solve the problem easier. It says that given that you know your state, what is probability of observing the future?

$$\beta_t(i) = P\left(O_{t+1}, O_{t+2}, \cdots O_T \mid q_t = S_i, \lambda\right)$$

Obviously, $\beta_T(i) = 1$.

$$\beta_t(i) = P\left(O_{t+1}, O_{t+2}, \cdots, O_T \mid q_t = S_i, \lambda\right)$$
$$= \sum_{k=1}^{N} P\left(O_{t+1}, O_{t+2}, \cdots, O_T, q_{t+1} = S_k \mid q_t = S_i, \lambda\right)$$
$$= \sum_{k=1}^{N} P\left(O_{t+2}, \cdots, O_T \mid O_{t+1}, q_{t+1} = S_k, q_t = S_i, \lambda\right) P(O_{t+1}, q_{t+1} = S_k|q_t = S_i, \lambda)$$
$$= \sum_{k=1}^{N} \beta_{t+1}(i) a_{i,k} b_{t+1}(k)$$

What sequence of states best explains observations? What does best means?
First, we define $\gamma_t(i)$ as probability of being in state i at time t.

$$\gamma_t(i) = P\left(q_t = S_i \mid O, \lambda\right)$$

and we can calculate that easily using α and β parameters.

$$\begin{gathered}\gamma_t(i) = P\left(q_t = S_i \mid O, \lambda\right) \\ = \frac{P(O_1, O_2, ..., O_T, q_t = S_i|\lambda)}{P(O|\lambda)} \\ = \frac{P\left(O_1 \ldots O_t, q_t = S_i \mid \lambda\right) P\left(O_{t+1} \ldots O_T \mid q_t = S_i, \lambda\right)}{P(O \mid \lambda)} \\ = \frac{\alpha_t(i)\beta_t(i)}{P(O \mid \lambda)} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^N \alpha_t(j)\beta_t(j)}\end{gathered}$$

we can define our goal as:

$$q_t = \underset{1 \le i \le N}{\operatorname{argmax}} \left[\gamma_t(i)\right]$$

The problem is that the result might not make sense. It might choose states that are not connected!
So Instead, we define the goal like this, i.e., find sequence of states which maximze the probability of observations.

$$\underset{Q}{\operatorname{argmax}} P(Q \mid O, \lambda) = \underset{Q}{\operatorname{argmax}} \frac{P(Q, O|\lambda)}{P(O|\lambda)} = \underset{Q}{\operatorname{argmax}} P(Q, O|\lambda)$$

Last equation is written since the denominator is independent of Q.

$$\delta_j(t) = \max_{q_1, q_2, \cdots, q_{t-1}} P\left(q_1 q_2 \ldots q_t = j, O_1 O_2 \ldots O_t \mid \lambda\right)$$

So,

$$\delta_{j+1}(t) = \max_{q_1, q_2, \cdots, q_t} P\left(q_1 q_2 \ldots q_{t+1} = j, O_1 O_2 \ldots O_{t+1} \mid \lambda\right)$$

This algorithm is run through viterbi function of my algorithm. The results are also saved in file `viterbi_output.txt`.

References

- Hidden Markov Model youtube
- A tutorial on hidden markov model
- Chapter 9, Hidden Markov Models
- Lecture 16, Hidden Markov Models

Q4

Here, I want to explain what is metropolist-hastings algorithm. Also, I want to implement the algorithm completely in pure python. So let's go.

Stationary Distribution

Suppose that we have a markov chain model. The transition matrix (or transition kernel) is known. Therefore, we can easily get stationary distribution, i.e., the probabiliy of staying in each state in the long run.

For example, suppose that we have a transition matrix P and we want to understand stationary distribution of P:

$$P = \begin{bmatrix} 0.1 & 0.2 & 0.7 \\ 0.3 & 0.6 & 0.1 \\ 0.3 & 0.4 & 0.3 \end{bmatrix}$$

```
import numpy as np
transition = np.array([[0.1,0.2,0.7],
                    [0.3,0.6,0.1],
                    [0.3,0.4,0.3] ])
```

We have two ways, we can calculate some large powers of transition matrix like P^{10000}:

```
np.linalg.matrix_power(transition,10000)
```

Which gives us

```
array([[0.25  , 0.4375, 0.3125],
       [0.25  , 0.4375, 0.3125],
       [0.25  , 0.4375, 0.3125]])
```

or we can use eigendecomposition to decompose matrix into a set a eigenvectors and eigenvalues, this is a common approach in taking high powers of matrices, especially big ones. Here I use the code from this stackoverflow post.

```
def get_stationary_distribution(Q):
    evals, evecs = np.linalg.eig(Q.T)
    evec1 = evecs[:,np.isclose(evals, 1)]
    evec1 = evec1[:,0]
    stationary = evec1 / evec1.sum()
    stationary = stationary.real
    stationary = stationary / stationary.sum()
    return stationary

stationary_dist=get_stationary_distribution(transition)
stationary_dist
```

This will also result in

```
array([0.25  , 0.4375, 0.3125])
```

It means that in the long run, we would be in first state 0.25% of time.

Metropolis-Hastings Algorithm

Now, suppose that we want to sample from this markov chain. There are two algorithms. One is called metropolis which works only on symmetric transition matrices in which $p_{ij} = p_{ji}$. But this is not always possible. Hasting, hence the name, extended the algorithm to be used with non-symmetric matrices.

We start from random state x. Then we choose one of its neighbours based on the transition probabiliy. We either accept it or reject it and stay where we are. This is the gist of algorithm.

First you need to start from a random state, let's call it X_0.

1- Sample a candidate point Y from our transition matrix given previous X.
2- Compute the ration $r = \frac{\pi(y)p(y,x)}{\pi(x)p(x,y)}$
3- Compute $\alpha = min(r, 1)$
4- Accept Y with probabiliy α or reject it with probabiliy $1 - \alpha$

Do this process iteratively and you will have a sample from markov chain.

```
def metropolis_hastings(sample_items, transition, stationary_dist):
    sample = np.zeros(sample_items,dtype=np.int32)
    for i in range(sample_items-1):
            x = sample[i]
            y = w_choice(transition[x])

            r_nominator = stationary_dist[y]*transition[y,x]
            r_denominator = stationary_dist[x]*transition[x,y]

            a = 0
            if r_denominator == 0:
                a = 1
            else:
                a = min(r_nominator / r_denominator,1)

            if random.uniform(0,1) <= a:
                sample[i+1] = y
            else:
                sample[i+1] = x
    return sample
print(metropolis_hastings(10,transition,stationary_dist))
```

```
[0 1 1 2 0 1 1 1 1 1]
```

> Since `r_denominator` could be zero, I added an if clause. It is also one of the cases that if `r_denominator` is zero. alpha should be one.

Also, `w_choice` function chooses one value from transition matrix. I got it from this stackoverflow post.

```
from random import uniform
import random
def w_choice(seq):
    """
    choose an item from list of items whose probability is given
    example: input [0.3,0.4,0.3]
    output: it may choose from [0,1,2] with given probabilty
    """
    total_prob = sum([item for item in seq])
    chosen = random.uniform(0,total_prob)
    cumulative = 0
    for i in range(len(seq)):
        cumulative += seq[i]
        if cumulative > chosen:
            return i
```

If I sample 1000 times from this markov chain and get the probablity of staying in each state. I should exactly get my stationary distribution. Let's test it.

```
import pandas as pd
number_of_iterations = 100000
samples_per_iteration = 100
samples=np.zeros((number_of_iterations,samples_per_iteration))
```

```
for i in range(number_of_iterations):
    samples[i]= metropolis_hastings(samples_per_iteration
                                    ,transition,stationary_dist)

samples=samples.reshape(-1)
imperical_stationary=pd.Series(samples).value_counts()
        .sort_index()/len(samples)
imperical_stationary
```

And here's the output. Pretty close. Huh?

```
0.0    0.260744
1.0    0.427493
2.0    0.311763
dtype: float64
```

For our example first I used the probabilities which I got from Q1.

```
model = HMM(initial_converged,transition_converged,emission_converged)
```

I've also written a function to normalize a matrix

```
def normalize_matrix(matrix):
    """
    input: two dimensional matrix
    returns a matrix which its rows sum to 1
    """
    return matrix / np.sum(matrix,1)[:,None]
```

Then I removed the edges and got the primary stationary matrix.

```
import copy
transition_matrix = convert_dictionary_to_matrix(model.transition_probs)
transition2 =copy.deepcopy(model.transition_probs)

remove_edge(transition2,8,12)
remove_edge(transition2,12,13)
remove_edge(transition2,18,19)
remove_edge(transition2,16,17)
remove_edge(transition2,6,12)

transition_matrix2 = convert_dictionary_to_matrix(transition2)
transition_matrix2 = normalize_matrix(transition_matrix2)

stationary_distribution=get_stationary_distribution(transition_matrix)
stationary_distribution
```

Then I calculated alpha values for all x and y's

```
state_count = len(transition_matrix)
a_array = np.zeros((state_count,state_count))
for i in range(state_count):
    for j in range(state_count):
        r_nominator = stationary_distribution[j]*transition_matrix2[j,i]
        r_denominator = stationary_distribution[i]*transition_matrix2[i,j]

        a = 0
        if r_denominator == 0:
            a = 1
        else:
            a = min(r_nominator / r_denominator,1)

        a_array[i,j] = a
```

and using alpha values from previous section, I dervied new stationary matrix which is pretty close to the former one.

```
new_transition_matrix = a_array * transition_matrix2
stationary2= get_stationary_distribution(normalize_matrix(new_transition_matrix))
### check if they are close
(stationary2-stationary_distribution) < 0.01
```

and this outputs

```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True])
```

I also sampled from the distribution which again confirms that stationary distribution and transition matrix which is derived is correct.

```
### sampling
iterations = 1000
sample_items = 100
x_array = np.zeros((iterations,sample_items))
for e in range(iterations):
    x_array[e,0] = int(random.uniform(0,state_count))
    for i in range(sample_items-1):
        x = int(x_array[e,i])
        y = w_choice(transition_matrix2[x])

        if random.uniform(0,1) <= a_array[x,y]:
            x_array[e,i+1] = y
        else:
            x_array[e,i+1] = x
```

This is evaluates the result of the sampling processs:

```
x_array=x_array.reshape(-1)
imperical_stationary=pd.Series(x_array).value_counts().sort_index()/len(x_array)
np.array(imperical_stationary)-stationary_distribution < 0.01
```

By this we see that this is pretty close to our former stationary distribution.

```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True])
```

So in general it shows that the algorithm is able to generate highly close results to what we need. It also shows that empirical distribution gets near the real stationary distribution.

References

- [On the Random Walk Metropolis Algorithm]
- [11. Sampling Methods: Markov Chain Monte Carlo]

Q5

I simulated to see what happens if we listen to the chief officer. He specifically says that probablity of two edges should be equal to min value of them. This would, of course, change the stationary distribution which I want we wanted.

So, let's see how it differs.

```
min_matrix=np.where(transition_matrix<transition_matrix.T,transition_matrix,transition_matr
ix.T)
proposal_stationary=get_stationary_distribution(normalize_matrix(min_matrix))
proposal_stationary
```

```
array([0.03623193, 0.05314085, 0.04338589, 0.039074  , 0.04035088,
       0.05016627, 0.0429694 , 0.02380878, 0.04097378, 0.04377247,
       0.0249732 , 0.03266443, 0.05314085, 0.02670919, 0.03966445,
       0.03096373, 0.04686786, 0.04565732, 0.03981049, 0.04184281,
       0.04970957, 0.04558968, 0.05314085, 0.03842504, 0.01696628])
```

so proposal distribution is differnet from our distribution which makes us believe that officer is right. Since it is different from what the thief may think, it confuses the thief.

But If the town is big, it might not be possible. This is becuase some probablities would be so small that they never converge to a stationary distribution. Also, it may introduce some sections of the city which are separated from other parts and this is problematic in this context.