Project 5: HMMs

In [1]:
```
# !pip install hmmlearn
```

In [2]:
```
# imports
import numpy as np
import matplotlib.pyplot as plt
from itertools import product
from functools import reduce
import pandas as pd
from collections import Counter
from hmmlearn import hmm

np.random.seed(200314659)
```

After importing the required libraries, we generate the state transition matrix and emission probability matrix by leveraging the random library. I also found a library in python that creates a model with the probability matrices, so that has been initialized as well

In [3]:
```
# Generating P_ij

P = np.random.random((4, 4))
for i in range(4):
    P[i] = P[i]/sum(P[i])
B = np.random.random((4, 3))
for i in range(4):
    B[i] = B[i]/sum(B[i])
pi = [1, 0, 0, 0]
model = hmm.MultinomialHMM(n_components=4, algorithm='viterbi', random_state=200
model.startprob_ = np.array(pi)
model.transmat_ = np.array(P)
model.emissionprob_ = np.array(B)
```

Task 1

The data set generation has been performed as explained in the document. The same has been verified for correctness by running the matrices in the hmmlearn library

In [4]:
```
t = 1
q = 1
states = [1]

r = np.random.choice([1,2,3,4], p=P[q-1])
O = [np.random.choice([1,2,3], p=B[r-1])]

for i in range(999):
    q = np.random.choice([1,2,3,4])
    r = np.random.choice([1,2,3,4], p=P[q-1])
    O.append(np.random.choice([1,2,3], p=B[r-1]))
    states.append(q)

print(Counter(states))
print(Counter(O))
```

```
Counter({1: 263, 4: 258, 3: 241, 2: 238})
Counter({3: 473, 1: 329, 2: 198})
```

Above implementation uses no major libraries. However, there were libraries to perform the same - this has been implemented in the block below

In [5]:
```
model_obs, model_states = model.sample(1000)
print(Counter(model_states.flatten()))
print(Counter(model_obs.flatten()))
```

```
Counter({2: 370, 1: 325, 3: 219, 0: 86})
Counter({2: 458, 0: 323, 1: 219})
```

Task 2

Given that you know the parameters of the HMM, calculate the probability $p(O|\lambda)$ that the sequence of observations

O = 123312331233 came from the HMM.

In [28]:
```
def mul(x, y):
    return x * y

P_mle = {}
B_mle = {}
for i, j in enumerate(map(list, zip(*P))):
    P_mle[i+1] = j
for i, j in enumerate(map(list, zip(*B))):
    B_mle[i+1] = j

pi = pd.DataFrame(data={1: [1], 2: [0], 3: [0], 4: [0]})

observations = [1,2,3,3]
pi_states = [1,2,3,4]

score = 0
all_hmms = list(product(*(pi_states,) * len(observations)))

for chain in all_hmms:
    e_chain = list(zip(chain, [1] + list(chain)))
    p_hs = list(map(lambda x: pd.DataFrame(data=P_mle, index=pd.Index([1,2,3,4],
    p_hs[0] = pi[chain[0]]

    e_obser = list(zip(observations, chain))
    p_obs = list(map(lambda x: pd.DataFrame(data=B_mle, index=pd.Index([1,2,3,4]

    score += reduce(mul, p_obs) * reduce(mul, p_hs)

print(score[0])
```

```
0.016158508588046218
```

In [8]:
```
observations = [1,2,3,3]
oo = [i-1 for i in observations]
obs_sequence = np.array([oo]).T
p = np.exp(model.score(obs_sequence))
print("p({}) = {} ".format([observations[i] for i in obs_sequence.T[0]], p))
```

```
p([1, 2, 3, 3]) = 0.016158508588046214
```

For the given sequence, when I tried running it on the code that followed the logic for Lambda calculation it took a lot of time. Thus, I used a smaller sample to check for correctness and ran the same observations on the model defined previously. The following output was obtained

In [9]:
```
model.score(obs_sequence)
```

Out[9]:
```
-4.1253085204918785
```

In [11]:
```
observations = [1,2,3,3,1,2,3,3,1,2,3,3]
oo = [i-1 for i in observations]
obs_sequence = np.array([oo]).T
score = np.exp(model.score(obs_sequence))
print(score)
```

```
2.2425228886660963e-06
```

In order to calculate this, we marginalize all possible chains of the hidden variables and determmine the MLE value or the score that the model determines by computing the product of all probabilities related to the observables and the the product of all probabilities of transitioning from x at t to x at t + 1.

From some further analysis, I also noticed that the score values for all possible combinations of an input observation adds up to 1 verifying the value obtained above to be correct.

Task 3

Estimation of most probable sequence

In [14]:
```
observations = [1,2,3,3,1,2,3,3,1,2,3]
oo = [i-1 for i in observations]

_, sequences = model.decode(np.array([oo]).T)
sequence = []

for s in sequences:
    sequence.append(states[int(s)])
```

In [15]:
```
print( "For the observations: ", observations, "we get sequence: ", sequence)
```

```
For the observations:  [1, 2, 3, 3, 1, 2, 3, 3, 1, 2, 3] we get sequence:  [1, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2]
```

The hmmlearn model allows us to decode the observations and identify the hiddenstates to determine the corresponding values. The observations and their corresponding sequences have been identified and displayed.

It was understood that the Viterbi algorithm was commonly used for the estimation of hidden state values using dynamic programming. The model.decode() function performs the same and it uses the alpha (highest joint probability) values for calculation.

Task 4

Training the hmm model

In [26]:
```
new_model = hmm.MultinomialHMM(n_components=4, algorithm='viterbi', random_state
new_model = new_model.fit(model_obs)
```

In [27]:
```
print("old_P = ",P)
print("new_P = ", new_model.transmat_)
print("old_B = ", B)
```

```
print("new_B = ", new_model.emissionprob_)
print("old_pi = ", model.startprob_)
print("new_pi = ", new_model.startprob_)
```

```
old_P =  [[0.35736501 0.1057952  0.2281218  0.30871799]
 [0.05958057 0.35752997 0.01098403 0.57190544]
 [0.05418169 0.19766124 0.7012921  0.04686497]
 [0.04710583 0.54091908 0.32629607 0.08567901]]
new_P =  [[0.32627748 0.30914319 0.17984501 0.18473432]
 [0.3795852  0.33625037 0.1399961  0.14416833]
 [0.3133156  0.29876962 0.19315349 0.19476129]
 [0.37708467 0.33225828 0.14437024 0.14628681]]
old_B =  [[0.4857143  0.10391807 0.41036764]
 [0.53486341 0.15204137 0.31309522]
 [0.20448086 0.37904062 0.41647852]
 [0.15550001 0.09513801 0.74936198]]
new_B =  [[0.31341797 0.05070164 0.63588039]
 [0.50957404 0.01426692 0.47615904]
 [0.02230523 0.6630869  0.31460787]
 [0.27853961 0.53070303 0.19075736]]
old_pi =  [1 0 0 0]
new_pi =  [1.06782351e-02 9.87957065e-01 1.66165708e-14 1.36469980e-03]
```

The previously generated sequence is used to create the emission and transmission probability values along with pi. We notice that there is a high unbalance in the values generated for P and B but the pi value is completely differnt in the new model. We also notice that it implements the Baum-Welch algorithm to do the same after calling the fit() function with the previously generated observations.

Conclusions

We can concude that the hidden states converges to a stationary state distribution and doesnt store the current observation.

Ref:

http://www.davidsbatista.net/blog/2017/11/11/HHM_and_Naive_Bayes/ (http://www.davidsbatista.net/blog/2017/11/11/HHM_and_Naive_Bayes/)
https://brilliant.org/wiki/stationary-distributions/ (https://brilliant.org/wiki/stationary-distributions/)
https://towardsdatascience.com/hidden-markov-model-implemented-from-scratch-72865bda430e (https://towardsdatascience.com/hidden-markov-model-implemented-from-scratch-72865bda430e)
https://hmmlearn.readthedocs.io/en/latest/tutorial.html (https://hmmlearn.readthedocs.io/en/latest/tutorial.html)