hw1

February 2, 2021

1 HW 1

Max Schrader 1/23/2021

1.1 HW 1a. Part 1

[1]:
```
import os
import sys
import csv
import random
PATH = !pwd  # a bit of Jupyter Magic to get the working path
PATH = os.path.join(*os.path.split(PATH[0])[:-1])
if PATH not in sys.path:
    print(f"adding {PATH} to path")
    sys.path.append(PATH)
```

```
adding /home/max/Documents/Homework/homework/ME-691 to path
```

1.1.1 Enter the Givens

[2]:
```
NUM_STATES = 3
INITIAL_PROB = [0.5, 0.2, 0.3]
TRANSITION_MATRIX = [[0.4, 0.3, 0.3],
                     [0.2, 0.6, 0.2],
                     [0.1, 0.1, 0.8]]
SEQUENCE_NUM = 100
STATES = 1000
FILE_PATH = os.path.join(PATH, 'HW1', 'states.csv')
```

1.1.2 Generating 100 sequences of 1000 states

Creating the functions & local variables

[3]:
```
def random_weighted(weights: list) -> list:
    return [num for num, weight in enumerate(weights) for _ in range(int(weight␣
 ↪* 100))]

transition_matrix = [random_weighted(weights) for weights in TRANSITION_MATRIX]
initial_prob = random_weighted(INITIAL_PROB)
```

```
def prob_generator(choice_list: list, data: list=[], count: int=0) -> list:
    if count < STATES:
        data.append(random.choice(choice_list))
        return prob_generator(choice_list=transition_matrix[data[-1]],␣
 ↪data=data, count=count+1)
    return data
```

Generating the states

[4]:
```
states = [prob_generator(choice_list=initial_prob, data=[], count=0) for _ in␣
 ↪range(SEQUENCE_NUM)]
```

1.1.3 Save to CSV

[5]:
```
with open(FILE_PATH, 'w') as f:
    writer = csv.writer(f, )
    writer.writerows(states)
```

1.2 HW 1a. Part 2

1.2.1 Read in the states

[6]:
```
# could also skip the i/o and just use "states" variable directly
states_2 = []
with open(FILE_PATH, 'r') as f:
    reader = csv.reader(f, )
    for row in reader:
        states_2.append([int(item) for item in row])
```

1.2.2 Initializing the Parameters

[7]:
```
A = [[0] * NUM_STATES for _ in range(NUM_STATES)]
count_A = [0] * NUM_STATES
PI = [0] * NUM_STATES
```

1.2.3 Calculating the Actual Probabilities

[8]:
```
for sequence in states_2:
    PI[sequence[0]] += 1
    for state, state_2 in zip(sequence[:-1], sequence[1:]):
        A[state][state_2] += 1
        count_A[state] += 1
final_A = list(zip(*[[round(value/divisor, 3) for value, divisor in zip(row,␣
 ↪count_A)] for row in A]))
final_pi = [round(pi/SEQUENCE_NUM, 3) for pi in PI]
```

[9]:
```
print("Calculated A: ", final_A)
print("Calculated PI: ", final_pi)
```

```
Calculated A:  [(0.394, 0.305, 0.3), (0.199, 0.605, 0.196), (0.1, 0.1, 0.801)]
Calculated PI:  [0.51, 0.14, 0.35]
```

1.2.4 Calculating the Error

[10]:
```
error_A = [[round(calc - actual, 3) for calc, actual in zip(row_c, row_a)] for␣
 ↪row_c, row_a in zip(final_A, TRANSITION_MATRIX)]
error_pi = [round(calc - actual, 3) for calc, actual in zip(final_pi,␣
 ↪INITIAL_PROB)]

print("Error A: ", error_A)
print("Error PI: ", error_pi)
```

```
Error A:  [[-0.006, 0.005, 0.0], [-0.001, 0.005, -0.004], [0.0, 0.0, 0.001]]
Error PI:  [0.01, -0.06, 0.05]
```

1.3 HW 1b. Part 1

[11]:
```
STATE_NUM = range(3)
OBSERVATIONS = 2
A = [[0.4, 0.3, 0.3],
     [0.2, 0.6, 0.2],
     [0.1, 0.1, 0.8]]
B = [[0.1, 0.9],
     [0.5, 0.5],
     [0.8, 0.2]]
PI = [0.5, 0.2, 0.3]

O = [0, 0, 1, 0, 1]
```

[12]:
```
def forward_algo(o: list, a: list, count: int) -> list:
    if count < len(o):
        a_ = [PI[i] * B[i][o[count]] for i in STATE_NUM] if count < 1 else␣
 ↪[sum([a[i] * A[i][j] for i in STATE_NUM]) * B[j][o[count]] for j in␣
 ↪STATE_NUM]
        return forward_algo(o=o, a=a_, count=count+1)
    return sum(a)

def backward_algo(o: list, b: list, count: int) -> list:
    if count < len(o) - 1:
        b_ = [1] * len(STATE_NUM) if count < 1 else [sum([A[i][j] * B[j][o[-1 *␣
 ↪count]] * b[j] for j in STATE_NUM]) for i in STATE_NUM]
        return backward_algo(o=o, b=b_, count=count+1)
    return b
```

```
def viterbi_algo(o: list, delta: list, psi: list, count: int) -> tuple:
    if count < len(o):
        if count < 1:
            delta.append([PI[i] * B[i][o[count]] for i in STATE_NUM])
            psi.append([0] * len(STATE_NUM))
        else:
            delta.append([max([delta[-1][i] * A[i][j] * B[j][o[count]] for i in␣
↪STATE_NUM]) for j in STATE_NUM])
            psi_max = [max([delta[-1][i] * A[i][j] for i in STATE_NUM]) for j␣
↪in STATE_NUM]
            psi.append([[delta[-1][i] * A[i][j] for i in STATE_NUM].
↪index(psi_max[j]) for j in STATE_NUM])
        return viterbi_algo(o, delta=delta, psi=psi, count = count + 1)
    return delta, psi, [[psi[-1 * t][i] for i, val in enumerate(delta[-1* t])␣
↪if val == max(delta[-1 * t])] for t in range(len(psi))]
```

[13]:
```
P_forward = forward_algo(o=O, a=[], count=0)
P_backward = backward_algo(o=O, b=[], count=0)

print(f"Probability Forward: {P_forward}")
print(f"Probability Backward: {P_backward}")
```

```
Probability Forward: 0.017151982400000005
Probability Backward: [0.10794000000000002, 0.10467200000000002,
0.06211400000000002]
```

[14]:
```
delta, psi, path = viterbi_algo(o=O, delta=[], psi=[], count=0)

print(f"The most probable path is: {[p[0] + 1 for p in path]}")
```

```
The most probable path is: [1, 3, 3, 3, 3]
```