Homework 9

Anand Kamble
Department of Scientific Computing
Florida State University

Problem 1

In this problem, we analyze a sequence of dice rolls generated by the Dishonest Casino model. The dataset `hmm_pb1.csv` represents these dice rolls, where each die can be in one of two states: 'Fair' or 'Loaded'. We aim to implement the Viterbi, forward, and backward algorithms to analyze this sequence and report results accordingly.

Part (a): Viterbi Algorithm Implementation

We implemented the Viterbi algorithm to identify the most probable sequence of states, given the observations. Using log probabilities for transition and emission probabilities helped mitigate potential floating-point underflow issues.

HMM Parameters

- **States**: $0 = \text{Fair}$, $1 = \text{Loaded}$
- **Transition Probabilities**:
$$a = \begin{bmatrix} 0.95 & 0.05 \\ 0.10 & 0.90 \end{bmatrix}$$
- **Emission Probabilities**: Fair die follows uniform distribution, while Loaded die is biased towards a roll of 6.
$$b = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{1}{10} & \frac{1}{10} & \frac{1}{10} & \frac{1}{10} & \frac{1}{10} & \frac{1}{2} \end{bmatrix}$$
- **Initial State Probabilities**: $\pi = [0.5, 0.5]$

Results

Using the Viterbi algorithm, we determined the most likely sequence of states y for the observed dice rolls:

```
Most likely sequence y of states (1=Fair, 2=Loaded):
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1]
```

Part (b): Forward and Backward Algorithms

The forward and backward algorithms were implemented with scaling to avoid underflow. We calculated $\alpha_{1,139}/\alpha_{2,139}$ and $\beta_{1,139}/\beta_{2,139}$ as required.

Results

The calculated ratios for $t = 139$ are as follows:

$$\frac{\alpha_{1,139}}{\alpha_{2,139}} = 13.079570428492065$$
$$\frac{\beta_{1,139}}{\beta_{2,139}} = 3.1584763971655754$$

Problem 2: Parameter Estimation using Baum-Welch Algorithm

In this problem, we are provided with a sequence of 20,000 dice rolls (dataset `hmm_pb2.csv`) and tasked with estimating the HMM parameters using the Baum-Welch algorithm.

Initialization

We initialized the initial state probabilities π, transition probabilities a, and emission probabilities b with random values, ensuring they sum to 1.

Baum-Welch Algorithm Implementation

The Baum-Welch algorithm iteratively updated π, a, and b through expectation and maximization steps, converging to estimates for the model parameters.

Results

After 500 iterations, the estimated parameters are:

- **Initial State Probabilities** (π):

$$\pi = [1.00000000\text{e}{+}000\ 9.61890076\text{e-}207]$$

- **State Transition Probabilities** (a):

$$a = \begin{bmatrix} 0.8711917 & 0.1288083 \\ 0.21299416 & 0.78700584 \end{bmatrix}$$

- **Observation Probabilities** (b):

$$b = \begin{bmatrix} 0.21237817 & 0.2137017 & 0.20424693 & 0.18968089 & 0.0900276 & 0.08996471 \\ 0.10482205 & 0.09082342 & 0.11362572 & 0.20883941 & 0.18721835 & 0.29467105 \end{bmatrix}$$

References

[1] Risto Hinno. *Baum-Welch Algorithm.* Accessed: 2024-10-31. 2023. URL: `https://ristohinno.medium.com/baum-welch-algorithm-4d4514cf9dbe`.

[2] Abhisek Jana. *Derivation and Implementation of Baum Welch Algorithm for Hidden Markov Model.* Accessed: 2024-10-31. 2019. URL: `https://adeveloperdiary.com/data-science/machine-learning/derivation-and-implementation-of-baum-welch-algorithm-for-hidden-markov-model/`.

[3] Pierian Training. *Viterbi Algorithm Implementation in Python: A Practical Guide.* Accessed: 2024-10-31. 2023. URL: `https://pieriantraining.com/viterbi-algorithm-implementation-in-python-a-practical-guide/`.

Appendix: Code

The Python code used for the implementation of the Viterbi, forward, backward, and Baum-Welch algorithms is attached below:

```
#%%
from typing import Any
import numpy as np
import pandas as pd

# Problem 1
# =========

# Read the dataset hmm_pb1.csv
data_pb1: pd.DataFrame = pd.read_csv('hmm_pb1.csv', header=None)
x_pb1:np.ndarray = data_pb1.values.flatten()  # Convert to 1D numpy array

# Define the HMM parameters as per the Dishonest casino model
# States: 0='Fair', 1='Loaded'
# Observations: dice rolls (1-6)

# Transition probabilities
a:np.ndarray = np.array([[0.95, 0.05],
              [0.10, 0.90]])

# Emission probabilities
b:np.ndarray = np.zeros((2, 6))
# Fair die probabilities
b[0, :] = 1/6
# Loaded die probabilities
b[1, :] = [1/10, 1/10, 1/10, 1/10, 1/10, 1/2]

# Initial state probabilities
pi = np.array([0.5, 0.5])

# Convert probabilities to log probabilities to avoid underflow
log_a:np.ndarray = np.log(a)
log_b:np.ndarray = np.log(b)
log_pi:np.ndarray = np.log(pi)

# Implement the Viterbi algorithm using log probabilities
def viterbi_log(x, log_pi, log_a, log_b) -> np.ndarray[Any,
↪   np.dtype[np.unsignedinteger[Any]]]:
    N: int = len(log_pi)  # Number of states
    T: int = len(x)       # Length of the observation sequence
    delta: np.ndarray[Any, np.dtype[np.float64]] = np.zeros((T, N))
    psi: np.ndarray[Any, np.dtype[Any]] = np.zeros((T, N), dtype=int)

    # Initialization
    delta[0, :] = log_pi + log_b[:, x[0]-1]
    psi[0, :] = 0

    # Recursion
    for t in range(1, T):
        for j in range(N):
            probabilities = delta[t-1, :] + log_a[:, j]
            psi[t, j] = np.argmax(probabilities)
            delta[t, j] = np.max(probabilities) + log_b[j, x[t]-1]
```

```
    # Termination
    path = np.zeros(T, dtype=int)
    path[T-1] = np.argmax(delta[T-1, :])

    # Path backtracking
    for t in range(T-2, -1, -1):
        path[t] = psi[t+1, path[t+1]]

    # Adjust path to match state labels (1='Fair', 2='Loaded')
    path += 1
    return path

# Run the Viterbi algorithm
y_viterbi = viterbi_log(x_pb1, log_pi, log_a, log_b)

# Output the obtained sequence y for verification
print("Most likely sequence y of states (1='Fair', 2='Loaded'):")
print(y_viterbi)

# Implement the forward algorithm with scaling
def forward_algorithm_scaled(x, pi, a, b) -> tuple[np.ndarray[Any,
↪   np.dtype[np.float64]], np.ndarray[Any, np.dtype[np.float64]]]:
    N: int = len(pi)
    T: int = len(x)
    alpha: np.ndarray[Any, np.dtype[np.float64]] = np.zeros((T, N))
    c: np.ndarray[Any, np.dtype[np.float64]] = np.zeros(T)

    # Initialization
    alpha[0, :] = pi * b[:, x[0]-1]
    c[0] = np.sum(alpha[0, :])
    alpha[0, :] /= c[0]

    # Induction
    for t in range(1, T):
        for j in range(N):
            alpha[t, j] = np.sum(alpha[t-1, :] * a[:, j]) * b[j, x[t]-1]
        c[t] = np.sum(alpha[t, :])
        alpha[t, :] /= c[t]
    return alpha, c

# Implement the backward algorithm with scaling
def backward_algorithm_scaled(x, a, b, c) -> np.ndarray[Any, np.dtype[np.float64]]:
    N = a.shape[0]
    T: int = len(x)
    beta: np.ndarray[Any, np.dtype[np.float64]] = np.zeros((T, N))

    # Initialization
    beta[T-1, :] = 1 / c[T-1]

    # Induction
    for t in range(T-2, -1, -1):
        for i in range(N):
            beta[t, i] = np.sum(a[i, :] * b[:, x[t+1]-1] * beta[t+1, :])
        beta[t, :] /= c[t]
    return beta

# Run the forward and backward algorithms
alpha, c = forward_algorithm_scaled(x_pb1, pi, a, b)
```

```python
beta = backward_algorithm_scaled(x_pb1, a, b, c)

# Report alpha_1_139/alpha_2_139 and beta_1_139/beta_2_139
t_index = 139  # t=139
alpha_ratio = alpha[t_index, 0] / alpha[t_index, 1]
beta_ratio = beta[t_index, 0] / beta[t_index, 1]

print("\nalpha_1_139 / alpha_2_139 =", alpha_ratio)
print("beta_1_139 / beta_2_139 =", beta_ratio)

# Problem 2
# =========

# Read the dataset hmm_pb2.csv
data_pb2 = pd.read_csv('hmm_pb2.csv', header=None)
x_pb2 = data_pb2.values.flatten()  # Convert to 1D numpy array

# Initialize the HMM parameters randomly
def initialize_random_parameters(N, M):
    pi = np.random.rand(N)
    pi /= np.sum(pi)
    a = np.random.rand(N, N)
    a /= np.sum(a, axis=1, keepdims=True)
    b = np.random.rand(N, M)
    b /= np.sum(b, axis=1, keepdims=True)
    return pi, a, b

# Implement the Baum-Welch algorithm
def baum_welch(x, N, M, max_iter=100):
    T = len(x)
    pi, a, b = initialize_random_parameters(N, M)
    for iteration in range(max_iter):
        # E-step
        alpha, c = forward_algorithm_scaled(x, pi, a, b)
        beta = backward_algorithm_scaled(x, a, b, c)

        log_likelihood = -np.sum(np.log(c))
        print(f"Iteration {iteration+1}, Log-Likelihood: {log_likelihood}")


        gamma = (alpha * beta) / np.sum(alpha * beta, axis=1, keepdims=True)
        xi = np.zeros((T-1, N, N))
        for t in range(T-1):
            denom = np.dot(np.dot(alpha[t, :], a) * b[:, x[t+1]-1], beta[t+1, :])
            for i in range(N):
                numerator = alpha[t, i] * a[i, :] * b[:, x[t+1]-1] * beta[t+1, :]
                xi[t, i, :] = numerator / denom

        # M-step
        pi = gamma[0, :]
        a = np.sum(xi, axis=0) / np.sum(gamma[:-1, :], axis=0)[:, None]
        b_num = np.zeros((N, M))
        for k in range(M):
            indices = np.where(x == k+1)[0]
            if len(indices) > 0:
                b_num[:, k] = np.sum(gamma[indices, :], axis=0)
        b = b_num / np.sum(gamma, axis=0)[:, None]
    return pi, a, b
```

```python
# Run the Baum-Welch algorithm on the observed data
N = 2  # Number of states
M = 6  # Number of possible observations (dice rolls 1-6)
pi_estimated, a_estimated, b_estimated = baum_welch(x_pb2, N, M, max_iter=500)

# Report the estimated HMM parameters
print("\nEstimated initial state probabilities (pi):")
print(pi_estimated)

print("\nEstimated state transition probabilities (a):")
print(a_estimated)

print("\nEstimated observation probabilities (b):")
print(b_estimated)
```