HMM

Saly Al Masri

Question 1: Matrix Formulation of the HMM Problem

Problem Explanation

The problem involves formulating the Hidden Markov Model (HMM) problem using the initial probability vector π, the transition probability matrix A, and the observation probability matrix B. The scenario describes a system with:

- **Two hidden states**:
  - c_1: A biased coin that shows heads with a probability of 0.9 and tails with 0.1.
  - c_2: A fair coin that shows heads and tails with equal probabilities of 0.5.
- **Observations (O_t)**: At each time step t, an observation is made (head or tail), but the hidden state that generated the observation is unknown.
- **Initial conditions**: Each coin has an equal probability of 0.5 of being selected initially.

Formulation in Matrix Form

To represent this problem in matrix form, we define the following:

Initial Probability Vector (π)

The initial probability vector π represents the probability distribution over the hidden states at the beginning ($t = 1$). From the problem description:

Text origin:

"Let us assume that the probability of selecting any coin is 0.5, both initially and at every next time step."

Thus, the initial probability vector is:

$$\pi = \begin{bmatrix} P(X_1 = c_1) \\ P(X_1 = c_2) \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}.$$

This indicates that both coins are equally likely to be selected initially.

Transition Probability Matrix (A)

The transition probability matrix A specifies the probabilities of transitioning from one hidden state to another. From the problem description:

Text origin:

> *"Let us assume that the probability of selecting any coin is 0.5, both initially and at every next time step."*

This means that, at any time step, there is an equal probability of transitioning from one state to any other state or staying in the same state. Specifically:

$$\begin{aligned} P(X_{t+1} = c_1 | X_t = c_1) &= 0.5, \quad P(X_{t+1} = c_2 | X_t = c_1) = 0.5, \\ P(X_{t+1} = c_1 | X_t = c_2) &= 0.5, \quad P(X_{t+1} = c_2 | X_t = c_2) = 0.5. \end{aligned}$$

Thus, the transition probability matrix is:

$$A = \begin{bmatrix} P(X_{t+1} = c_1 | X_t = c_1) & P(X_{t+1} = c_2 | X_t = c_1) \\ P(X_{t+1} = c_1 | X_t = c_2) & P(X_{t+1} = c_2 | X_t = c_2) \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}.$$

Observation Probability Matrix (B)

The observation probability matrix B describes the probability of observing a specific outcome (head or tail) given a hidden state. From the problem description:

Text origin:

> *"We know that coin c_1 is biased and will show heads with probability* 0.9 *and tails with probability* 0.1*, while coin c_2 shows heads and tails each with a probability of* 0.5*."*

Thus:

$$\begin{aligned} P(O_t = \mathrm{H} | X_t = c_1) &= 0.9, \quad P(O_t = \mathrm{T} | X_t = c_1) = 0.1, \\ P(O_t = \mathrm{H} | X_t = c_2) &= 0.5, \quad P(O_t = \mathrm{T} | X_t = c_2) = 0.5. \end{aligned}$$

The observation probability matrix is:

$$B = \begin{bmatrix} P(O_t = \mathrm{H} | X_t = c_1) & P(O_t = \mathrm{T} | X_t = c_1) \\ P(O_t = \mathrm{H} | X_t = c_2) & P(O_t = \mathrm{T} | X_t = c_2) \end{bmatrix} = \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix}.$$

Final Representation

Using the matrices derived from the problem description, the HMM model can be represented as:

$$\pi = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}, \quad A = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}, \quad B = \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix}.$$

Each component (π, A, B) is directly linked to a specific part of the text, ensuring the formulation adheres to the problem requirements.

HMM0: Next Observation Distribution

Code Analysis and Fulfillment of Requirements

Parsing Matrices

The function `parse_matrix` reads and parses a matrix from the input provided in the format specified by the requirements. This function ensures:

- Each matrix is represented as a list of lists in Python.
- The dimensions of the matrix (rows and columns) are extracted and validated.

This directly addresses the need for handling the input format provided by Kattis and ensures that the transition matrix, observation matrix, and initial state distribution are correctly represented for further operations.

Matrix Multiplication

The function `multiply_matrices` performs matrix multiplication, a core operation for both steps of HMM0. Specifically:

- It validates the dimensions of the input matrices to ensure they are compatible for multiplication, raising an error if they are not.
- It calculates the product of two matrices using nested loops, ensuring that the mathematical definition of matrix multiplication is respected.

This implementation is essential for fulfilling the first requirement of multiplying the initial state distribution with the transition matrix and the second requirement of multiplying the resulting distribution with the observation matrix.

Answer to Question 2: Result of the First Operation

To answer this question, we multiply the initial state distribution vector π with the transition matrix A.

Given Inputs

The initial state distribution is:

$$\pi = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}$$

The transition matrix A is:

$$A = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$$

Matrix Multiplication

The resulting state distribution after the transition is given by:

$$\text{Next State Distribution} = \pi \times A$$

The calculation is performed as follows:

$$\text{Row 1:} \quad (0.5 \cdot 0.5) + (0.5 \cdot 0.5) = 0.25 + 0.25 = 0.5$$
$$\text{Row 2:} \quad (0.5 \cdot 0.5) + (0.5 \cdot 0.5) = 0.25 + 0.25 = 0.5$$

Result

The resulting next state distribution is:

$$\text{Next State Distribution} = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}$$

—

Answer to Question 3: Result of the Second Operation

To answer this question, we take the result from the first operation (the next state distribution vector) and multiply it with the observation matrix B.

Given Inputs

The next state distribution from Question 2 is:

$$\text{Next State Distribution} = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}$$

The observation matrix B is:

$$B = \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix}$$

Matrix Multiplication

The resulting observation distribution is given by:

$$\text{Next Observation Distribution} = \text{Next State Distribution} \times B$$

The calculation is performed as follows:

$$\text{Observation 1 (H):} \quad (0.5 \cdot 0.9) + (0.5 \cdot 0.5) = 0.45 + 0.25 = 0.7$$

$$\text{Observation 2 (T):} \quad (0.5 \cdot 0.1) + (0.5 \cdot 0.5) = 0.05 + 0.25 = 0.3$$

Result

The resulting next observation distribution is:

$$\text{Next Observation Distribution} = \begin{bmatrix} 0.7 & 0.3 \end{bmatrix}$$

Interpretation

This result represents the probabilities of observing heads and tails at the next time step, based on the current state distribution and the observation matrix.

HMM1 Problem

Parsing the Matrices

Code (Lines 1–6):

```
def parse_matrix(input_line):
    values = list(map(float, input_line.split()))
    rows, cols = int(values[0]), int(values[1])
    matrix = [values[2 + i * cols: 2 + (i + 1) * cols] for i in range(rows)]
    return matrix, rows, cols
```

This section parses matrices from input lines, ensuring they match the format (row size, column size, and elements).

Text Correspondence:

- *"You might want to store the α values in matrix form. Be careful to index all matrices and vectors correctly and to maintain the temporal order of summation and product as in the formulas."*

Forward Algorithm Initialization

Code (Lines 12–19, Step 1):

```
# Initialize alpha at time t=1
for i in range(num_states):
    alpha[0][i] = initial_state_dist[0][i] * emission_matrix[i][emissions[0]]
```

This initializes the α values for $t = 1$, using the equation:

$$\alpha_1(i) = \pi_i b_i(O_1), \tag{1}$$

where:

- π_i: Initial probability for state i.
- $b_i(O_1)$: Observation probability for the first observation O_1.

Text Correspondence:

- Equations (2.6), (2.7), (2.8):

$$\alpha_1(i) = \pi_i b_i(O_1).$$

- *"We start off by computing the probability of having observed the first observation o_1 and having been in any of the hidden states."*

Forward Algorithm Recursion

Code (Lines 21–27, Step 2):

```
for t in range(1, num_emissions):
    for i in range(num_states):
        alpha[t][i] = sum(alpha[t - 1][j] * transition_matrix[j][i] for j in range(nu
                      * emission_matrix[i][emissions[t]]
```

This computes $\alpha_t(i)$ recursively for $t > 1$ using the equation:

$$\alpha_t(i) = \left[\sum_{j=1}^{N} \alpha_{t-1}(j) a_{ji}\right] b_i(O_t), \tag{2}$$

where:

- a_{ji}: Transition probability from state j to state i.
- $b_i(O_t)$: Observation probability of O_t given state i.
- $\alpha_{t-1}(j)$: Previous α values.

Text Correspondence:

- Equations (2.9–2.13):

$$\alpha_t(i) = b_i(O_t)\left[\sum_{j=1}^{N} a_{ji}\alpha_{t-1}(j)\right].$$

- *"We need to marginalize over the probability of having been in any other state at $t-1$... multiply this estimate with the matching observation probability."*

Final Probability Computation

Code (Lines 29–30, Step 3):

```
final_probability = sum(alpha[num_emissions - 1][i] for i in range(num_states))
```

This computes the final probability of the entire observation sequence:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i). \tag{3}$$

Text Correspondence:

- Final equation:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i).$$

- *"Sum up final α values to get the probability of the sequence."*

Key Points from Code and Text

1. **Matrix Parsing (Lines 1–6):** Matches the input structure and prepares for computation.
    - Text: *"Index all matrices and vectors correctly."*
2. **Initialization of $\alpha_1(i)$ (Lines 12–19):** Direct implementation of Equation (2.8).
3. **Recursive Computation (Lines 21–27):** Matches Equations (2.9–2.13).
4. **Final Probability (Lines 29–30):** Implements the final summation step.

Question 4: Why is it valid to substitute $O_{1:t} = o_{1:t}$ with $O_t = o_t$ when we condition on the state $X_t = x_i$?

Explanation

The substitution $O_{1:t} = o_{1:t}$ with $O_t = o_t$ is valid due to the Markov property of Hidden Markov Models (HMMs). The Markov property ensures that:

1. The current state X_t depends only on the previous state X_{t-1}, not on the entire history of states:

$$P(X_t|X_{t-1}, X_{t-2}, \ldots, X_1) = P(X_t|X_{t-1}).$$

2. The current observation O_t depends only on the current state X_t, making it independent of all past observations and states:

$$P(O_t|X_t, X_{t-1}, \ldots, X_1) = P(O_t|X_t).$$

In the Forward Algorithm

Base Case: At $t = 1$,

$$\alpha_1(i) = P(O_1 = o_1, X_1 = x_i|\lambda) = P(O_1 = o_1|X_1 = x_i)P(X_1 = x_i).$$

$$\alpha_1(i) = \pi_i b_i(O_1),$$

This uses the initial state distribution π_i and the emission probabilities $b_i(o_1)$.

Inductive Step: For $t > 1$,

$$\alpha_t(i) = P(O_{1:t} = o_{1:t}, X_t = x_i|\lambda).$$

Using the Markov property:

$$P(O_{1:t}|X_t) = P(O_t|X_t)P(O_{1:t-1}|X_t),$$

we compute:

$$\alpha_t(i) = P(O_t = o_t|X_t = x_i)\sum_{j=1}^{N} \alpha_{t-1}(j)P(X_t = x_i|X_{t-1} = x_j),$$

$$\alpha_t(i) = \left[\sum_{j=1}^{N} \alpha_{t-1}(j)a_{ji}\right] b_i(O_t)$$

where:

- $P(O_t = o_t | X_t = x_i) = b_i(o_t)$ (emission probability),
- $P(X_t = x_i | X_{t-1} = x_j) = a_{ji}$ (transition probability).

This computes the final probability of the entire observation sequence:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i). \tag{4}$$

Conclusion

By conditioning on $X_t = x_i$, the observation $O_t = o_t$ becomes independent of all previous observations and states. This allows us to compute $\alpha_t(i)$ recursively using only the current observation O_t and the probabilities from the previous step $\alpha_{t-1}(j)$. The time complexity of the forward algorithm is $O(N^2T)$, where N is the number of states, and T is the number of time steps.

HMM2

Step 1: Initialization

Text Reference:
The initialization is specified in equations:

$$\delta_1(i) = \pi_i b_i(o_1) \quad \text{(equations 2.16 and 2.17).}$$

This step computes the initial probabilities for all states based on the initial state distribution (π_i) and the emission probability of the first observation ($b_i(o_1)$).

Code Reference:

```
for i in range(num_states):
    delta[0][i] = initial_state_dist[0][i] * emission_matrix[i][emissions[0]]
    delta_idx[0][i] = 0  # No previous state for t=0
```

- The code correctly calculates $\delta_1(i)$ for all i, matching the formula $\pi_i b_i(o_1)$. - The statement `delta_idx[0][i] = 0` ensures no predecessor is assigned for the first time step, as required.

—

Step 2: Recursion

Text Reference:
The recursion is defined in equations:

$$\delta_t(i) = \max_j \left[\delta_{t-1}(j) a_{ji}\right] b_i(o_t) \quad \text{(equations 2.18 and 2.19).}$$

It updates $\delta_t(i)$ by considering all possible paths leading to state i at time t, selecting the maximum probability path.

Code Reference:

```
for t in range(1, num_emissions):
    for i in range(num_states):
        max_prob = float('-inf')
        max_state = 0
        for j in range(num_states):
            prob = delta[t - 1][j] * transition_matrix[j][i]
            if prob > max_prob:
                max_prob = prob
                max_state = j
        delta[t][i] = max_prob * emission_matrix[i][emissions[t]]
        delta_idx[t][i] = max_state
```

- The code calculates $\delta_t(i)$ by iterating over all states j (the previous states) and selecting the one that maximizes $\delta_{t-1}(j)a_{ji}$. - The term $b_i(o_t)$ is correctly multiplied after finding the maximum. - The equation $\delta_t^{\text{id}}(i) = \text{argmax}_j[\delta_{t-1}(j)a_{ji}]$ (equation 2.20) is implemented as `delta_idx[t][i] = max_state`.

—

Step 3: Backtracking

Text Reference:
The backtracking process is described in equations 2.22 and 2.23. It reconstructs the most likely sequence of states by tracing back through the δ^{id} matrix.

Code Reference:

```
most_likely_sequence = [0] * num_emissions
most_likely_sequence[-1] = max(range(num_states), key=lambda i: delta[num_emissions -

for t in range(num_emissions - 2, -1, -1):
    most_likely_sequence[t] = delta_idx[t + 1][most_likely_sequence[t + 1]]
```

- The final state is selected as $\arg\max_j \delta_T(j)$ (equation 2.21). - The most likely sequence is reconstructed using δ^{id}, as specified in the backtracking formulas.

—

Input Parsing and Validation

- **Parsing Matrices:** The function `parse_matrix` extracts the dimensions and contents of the matrices, matching the input structure typically provided for HMM problems. - **Emission Sequence Validation:** The code ensures the emission sequence length matches the provided count, preventing runtime errors.

—

Code and Text Alignment

Below is the mapping of the key formulas in the text to the corresponding sections in the code:

Text Equation	Code Section
Equation 2.16	`delta[0][i] = initial_state_dist[0][i] * emission_matrix[i][emissions[0]]`
Equation 2.17	Initialization loop for $\delta_1(i)$.
Equation 2.18	`prob = delta[t - 1][j] * transition_matrix[j][i]`
Equation 2.19	`delta[t][i] = max_prob * emission_matrix[i][emissions[t]]`
Equation 2.20	`delta_idx[t][i] = max_state`
Equation 2.21	`most_likely_sequence[-1] = max(range(num_states), ... )`
Equations 2.22-2.23	`most_likely_sequence[t] = delta_idx[t + 1][most_likely_sequence[t + 1]]`

Question 5

The question asks:

- How many values are stored in the matrices δ and δ^{id}, and what is their role?
- These matrices are computed using the **Viterbi algorithm** to find the most likely sequence of states for a given observation sequence.

—

Objective

The objective is to:

1. Understand the role of δ and δ^{id} matrices in the Viterbi algorithm.
2. Calculate the number of values stored in each matrix.
3. Link this calculation to the parameters of the problem, such as the number of time steps (T) and the number of states (N).

—

Role of δ and δ^{id} Matrices

$\delta_t(i)$

- **Definition**: Represents the highest probability of any path that accounts for the first t observations and ends at state i.
- **Calculation**:
  - At $t = 1$:
  $$\delta_1(i) = \pi_i b_i(O_1),$$
  where:

    - π_i is the initial state probability.
    - $b_i(O_1)$ is the probability of observing O_1 given state i.
  - For $t > 1$:

$$\delta_t(i) = \max_j \left[\delta_{t-1}(j)a_{ji}\right] b_i(O_t),$$

  where:
    - $\delta_{t-1}(j)$ is the value from the previous time step.
    - a_{ji} is the transition probability from state j to state i.
    - $b_i(O_t)$ is the emission probability of observing O_t given state i.

$\delta_t^{\mathbf{id}}(i)$

- **Definition**: Stores the state j that maximized $\delta_{t-1}(j)a_{ji}$ for each i.
- **Purpose**: Helps in backtracking the most likely sequence of states once δ is fully computed.

—

Number of Values Stored

δ Matrix

- δ has one value for each combination of t (time step) and i (state).
- Number of time steps: T.
- Number of states: N.
- **Total values**: $T \times N$.

$\delta^{\mathbf{id}}$ Matrix

- δ^{id} stores one value for each combination of t (except the first time step) and i (state).
- Number of time steps: $T - 1$ (because it starts at $t = 2$).
- Number of states: N.
- **Total values**: $(T - 1) \times N$.

—

Detailed Steps to Solve

1. **Initialization**:
$$\delta_1(i) = \pi_i b_i(O_1), \quad \forall i.$$
Store $\delta_1(i)$ in the δ matrix.

2. **Inductive Step**: For each t from 2 to T, compute:
$$\delta_t(i) = \max_j \left[\delta_{t-1}(j) a_{ji}\right] b_i(O_t),$$
$$\delta_t^{\text{id}}(i) = \arg\max_j \left[\delta_{t-1}(j) a_{ji}\right].$$
Store the values in δ and δ^{id}.

3. **Backtracking**: Start from $t = T$ and trace back using δ^{id} to find the most likely sequence of states.

—

Final Answer

- **Number of values in δ:** $T \times N$.
- **Number of values in δ^{id}:** $(T-1) \times N$.

—

Textual Mapping to Images

1. **Image: Viterbi Algorithm Formulas**:
   - The formula for $\delta_t(i)$ (inductive step) is shown.
   - The role of $\delta_t^{\text{id}}(i)$ is explained as storing the index j that maximized $\delta_{t-1}(j)a_{ji}$.
2. **Image: Initialization, Inductive Step, Termination**:
   - The initialization formula $\delta_1(i) = \pi_i b_i(O_1)$ matches the first step in the algorithm.
   - The inductive step for $\delta_t(i)$ and $\delta_t^{\text{id}}(i)$ is clearly defined.
   - The backtracking process using δ^{id} is shown in termination.

1 HMM3

1. Forward Algorithm (α)

Theory

The forward algorithm computes the **probability of observing the partial observation sequence $O_{1:t}$ up to time t, given the current state $X_t = x_i$.** This is used in conjunction with the backward algorithm for parameter re-estimation and log-likelihood calculations.

Formula:

- **Initialization ($t = 0$):**
$$\alpha_0(i) = \pi_i \cdot b_i(O_1)$$
- **Recursion ($t > 0$):**
$$\alpha_t(i) = \left[\sum_{j=1}^{N} \alpha_{t-1}(j) \cdot a_{ji} \right] \cdot b_i(O_t)$$
- **Scaling for Numerical Stability:** Use scaling factors c_t to prevent underflow:
$$c_t = \frac{1}{\sum_{i=1}^{N} \alpha_t(i)}, \quad \alpha_t(i) \leftarrow \alpha_t(i) \cdot c_t$$

Code Implementation

```
def forward_algorithm(A, B, pi, emissions):
    N = len(A)
    T = len(emissions)
    alpha = [[0.0] * N for _ in range(T)]
    scale = [0.0] * T  # Scaling factors for each time step

    # Initialize alpha for t=0
    for i in range(N):
        alpha[0][i] = pi[0][i] * B[i][emissions[0]]
        scale[0] += alpha[0][i]

    # Scale alpha at t=0
    scale[0] = 1.0 / scale[0]
    for i in range(N):
        alpha[0][i] *= scale[0]

    # Compute alpha for t > 0
    for t in range(1, T):
        for i in range(N):
            for j in range(N):
                alpha[t][i] += alpha[t - 1][j] * A[j][i]
            alpha[t][i] *= B[i][emissions[t]]
            scale[t] += alpha[t][i]

        # Scale alpha at time t
        scale[t] = 1.0 / scale[t]
        for i in range(N):
            alpha[t][i] *= scale[t]

    return alpha, scale
```

2. Backward Algorithm (β)

Theory

The backward algorithm computes the **probability of observing all future observations $O_{t+1:T}$ given the current state $X_t = x_i$.** The formula is:

$$\beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(j) \cdot b_j(O_{t+1}) \cdot a_{ij}$$

The initialization step:

$$\beta_T(i) = 1$$

Code Implementation

```
def backward_algorithm(A, B, emissions, scale):
    N = len(A)
    T = len(emissions)
    beta = [[0.0] * N for _ in range(T)]

    # Initialize beta at time T-1
    for i in range(N):
        beta[T - 1][i] = scale[T - 1]

    # Compute beta for t < T-1
    for t in range(T - 2, -1, -1):
        for i in range(N):
            for j in range(N):
                beta[t][i] += A[i][j] * B[j][emissions[t + 1]] * beta[t + 1][j]
            beta[t][i] *= scale[t]

    return beta
```

3. Gamma and Di-Gamma Calculation

Theory

Gamma ($\gamma_t(i)$) is the **probability of being in state i at time t,** given all observations:

$$\gamma_t(i) = \frac{\alpha_t(i) \cdot \beta_t(i)}{\sum_{k=1}^{N} \alpha_t(k) \cdot \beta_t(k)}$$

Di-Gamma ($\xi_t(i,j)$) is the **joint probability of being in state i at time t and state j at time $t+1$:**

$$\xi_t(i,j) = \frac{\alpha_t(i) \cdot a_{ij} \cdot b_j(O_{t+1}) \cdot \beta_{t+1}(j)}{\sum_{k=1}^{N} \alpha_t(k) \cdot \beta_t(k)}$$

Code Implementation

```
gamma = [[0.0] * N for _ in range(T)]
di_gamma = [[[0.0] * N for _ in range(N)] for _ in range(T - 1)]

for t in range(T - 1):
    for i in range(N):
        gamma[t][i] = sum(
            alpha[t][i] * A[i][j] * B[j][emissions[t + 1]] * beta[t + 1][j]
            for j in range(N)
        )
        for j in range(N):
            di_gamma[t][i][j] = (
                alpha[t][i] * A[i][j] * B[j][emissions[t + 1]] * beta[t + 1][j]
            )

for i in range(N):
    gamma[T - 1][i] = alpha[T - 1][i]
```

4. Re-Estimation of Parameters (A, B, π)

Theory

Parameters are **iteratively improved to maximize the likelihood of the observation sequence**, making the model more accurate **Transition Matrix** (A):

$$a_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

Emission Matrix (B):

$$b_j(k) = \frac{\sum_{t=1}^{T} \gamma_t(j) \cdot 1(O_t = k)}{\sum_{t=1}^{T} \gamma_t(j)}$$

Initial Probabilities (π):

$$\pi_i = \gamma_1(i)$$

Code Implementation

```
# Re-estimate A, B, and pi
for i in range(N):
    # Update pi
    pi[0][i] = gamma[0][i]

    # Update A
    denom = sum(gamma[t][i] for t in range(T - 1))
    for j in range(N):
        numer = sum(di_gamma[t][i][j] for t in range(T - 1))
        A[i][j] = numer / denom if denom != 0 else 0.0
    # Update B
    denom = sum(gamma[t][i] for t in range(T))
    for k in range(M):
        numer = sum(gamma[t][i] for t in range(T) if emissions[t] == k)
        B[i][k] = numer / denom if denom != 0 else 0.0
```

5. Log-Likelihood for Convergence

Theory

The log-likelihood of the observation sequence, **Measures how well the model explains the observation sequence. Iterations stop when the log-likelihood converges**, preventing unnecessary computations:

$$\log P(O|\lambda) = -\sum_{t=1}^{T} \log(c_t)$$

Code Implementation

```
def log_prob(scale):
    return -sum(math.log(s) for s in scale)
```

6. Overall Baum-Welch Algorithm

Theory

The Baum-Welch algorithm iteratively refines the model parameters:

1. Initialize $\lambda = (A, B, \pi)$.
2. Compute all $\alpha_t(i)$, $\beta_t(i)$, $\gamma_t(i,j)$, $\gamma_t(i)$ values.
3. Re-estimate $\lambda = (A, B, \pi)$.
4. Repeat until convergence.

Code Implementation

```
def baum_welch(A, B, pi, emissions, max_iterations=100, threshold=1e-6):
    old_log_prob = -float("inf")

    for iteration in range(max_iterations):
        # Forward and backward passes
        alpha, scale = forward_algorithm(A, B, pi, emissions)
        beta = backward_algorithm(A, B, emissions, scale)

        # Compute gamma and di-gamma
        gamma = [[0.0] * N for _ in range(T)]
        di_gamma = [[[0.0] * N for _ in range(N)] for _ in range(T - 1)]
        # (Gamma and Di-Gamma code here)

        # Re-estimate A, B, and pi
        # (Re-estimation code here)

        # Check for convergence
        log_prob_curr = log_prob(scale)
        if log_prob_curr - old_log_prob < threshold:
            break
        old_log_prob = log_prob_curr

    return A, B
```

Answer to Question 6

To address why we need to divide the di-gamma function $\gamma_t(i, j)$ by the sum of the final α values, we provide a detailed explanation as follows:

—

Step 1: Definition of the Di-Gamma and Gamma Functions

Di-Gamma Function $\gamma_t(i, j)$: The di-gamma function $\gamma_t(i, j)$ represents the joint probability of being in state i at time t and transitioning to state j at time $t + 1$, given the observation sequence and model parameters $\lambda = (A, B, \pi)$. It is defined as:

$$\gamma_t(i, j) = P(q_t = S_i, q_{t+1} = S_j \mid O, \lambda).$$

Expanding this:

$$\gamma_t(i, j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{P(O \mid \lambda)},$$

where:

- $\alpha_t(i)$: Probability of observing $O_1, O_2, \ldots, O_t$ and being in state i at time t.
- $\beta_{t+1}(j)$: Probability of observing $O_{t+1}, O_{t+2}, \ldots, O_T$ given that state j is active at time $t + 1$.
- a_{ij}: Transition probability from state i to state j.
- $b_j(O_{t+1})$: Emission probability of observing O_{t+1} given state j.
- $P(O \mid \lambda)$: Total probability of the observation sequence O.

Gamma Function $\gamma_t(i)$: The gamma function $\gamma_t(i)$ is the marginal probability of being in state i at time t, given the observation sequence. It is calculated as:

$$\gamma_t(i) = \sum_{j=1}^{N} \gamma_t(i, j).$$

—

Step 2: The Role of the Sum of Final α Values

The sum of the final α values, $\sum_{i=1}^{N} \alpha_T(i)$, is equal to $P(O \mid \lambda)$, the total probability of observing the sequence O given the model λ. Dividing by this ensures that the computed probabilities are valid and normalized.

Reason for Normalization:

- The di-gamma function $\gamma_t(i,j)$ computes unnormalized probabilities for states i and j.
- Without dividing by $P(O \mid \lambda)$, these values would not sum to 1 and would only be proportional to the true probabilities.
- By dividing by $P(O \mid \lambda)$, we ensure $\gamma_t(i,j)$ represents valid probabilities that sum to 1.

—

Step 3: Connection to the Baum-Welch Algorithm

The Baum-Welch algorithm, an Expectation-Maximization (EM) method, relies on $\gamma_t(i,j)$ and $\gamma_t(i)$ to estimate:

- The expected number of transitions between states i and j, calculated as $\sum_{t=1}^{T-1} \gamma_t(i,j)$.
- The expected number of visits to state i, calculated as $\sum_{t=1}^{T} \gamma_t(i)$.

By normalizing $\gamma_t(i,j)$ with $P(O \mid \lambda)$, we ensure the re-estimated model parameters (e.g., transition probabilities a_{ij}) are valid and sum to 1.

—

Step 4: Conclusion

We divide the di-gamma function $\gamma_t(i,j)$ by the sum of the final α values, $P(O \mid \lambda)$, because:

1. $P(O \mid \lambda)$ acts as a normalization constant for the computed probabilities.
2. This step ensures $\gamma_t(i,j)$ represents valid conditional probabilities given the observation sequence O.
3. Normalized probabilities are essential for updating the model parameters during the Baum-Welch algorithm, ensuring they remain valid probabilities.

Without this normalization, the resulting probabilities would not sum to 1, leading to incorrect parameter re-estimation.

Question 5: Matrix Dimensions and Stored Values in δ and δ^{idx}

Understanding the Dimensions

- **Variables:**
  - N: Number of hidden states in the HMM.
  - T: Length of the observation sequence.
- **δ Matrix:**
  - Represents the highest probability along a single path, up to time t, that ends in state i.
  - Dimensions: $T \times N$.
- **δ^{idx} Matrix:**
  - Stores the index of the state that maximized $\delta_t(i)$ at each time t for each state i.
  - Dimensions: $(T - 1) \times N$.

Calculating the Number of Stored Values

- **For the δ Matrix:**
  - Number of time steps: T.
  - Number of states: N.
  - **Total values stored:** $T \times N$.
- **For the δ^{idx} Matrix:**
  - Number of time steps: $T - 1$ (since we start from $t = 2$).
  - Number of states: N.
  - **Total values stored:** $(T - 1) \times N$.

Example Illustration

- If $N = 3$ states and an observation sequence of length $T = 5$:
  - δ **Matrix:** $5 \times 3 = 15$ values.
  - δ^{idx} **Matrix:** $(5 - 1) \times 3 = 12$ values.

Conclusion

- δ Matrix: Stores $T \times N$ values.
- δ^{idx} Matrix: Stores $(T - 1) \times N$ values.

—

Question 6: Importance of Dividing by the Sum Over Final α Values in the Di-Gamma Function

Understanding the Di-Gamma Function

The di-gamma function $\gamma_t(i, j)$ represents the probability of being in state i at time t and transitioning to state j at time $t + 1$, given the entire observation sequence $O_{1:T}$ and the model parameters λ:

$$\gamma_t(i, j) = P(X_t = i, X_{t+1} = j | O_{1:T}, \lambda)$$

Formula for $\gamma_t(i, j)$:

$$\gamma_t(i, j) = \frac{\alpha_t(i) a_{i,j} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{k=1}^{N} \alpha_T(k)}$$

- **Numerator:** Joint probability of being in state i at time t, transitioning to state j at $t + 1$, and observing the entire sequence $O_{1:T}$.
- **Denominator:** Total probability of the observation sequence $P(O_{1:T}|\lambda)$, computed as $\sum_{k=1}^{N} \alpha_T(k)$.

Reason for Dividing by the Sum Over Final α Values

- **Normalization:**
  - Dividing by $\sum_{k=1}^{N} \alpha_T(k)$ normalizes $\gamma_t(i, j)$ so that the probabilities sum to 1 over all possible state transitions.
- **Computing Conditional Probabilities:**
  - The numerator gives the joint probability $P(X_t = i, X_{t+1} = j, O_{1:T}|\lambda)$.
  - Dividing by $P(O_{1:T}|\lambda)$ converts this to the conditional probability $P(X_t = i, X_{t+1} = j|O_{1:T}, \lambda)$.
- **Parameter Estimation:**
  - Accurate estimation of the model parameters A, B, and π requires normalized probabilities.

Mathematical Justification

- **Total Probability of Observations:**

$$P(O_{1:T}|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

- **Conditional Probability Calculation:**

$$\gamma_t(i, j) = \frac{P(X_t = i, X_{t+1} = j, O_{1:T}|\lambda)}{P(O_{1:T}|\lambda)} = P(X_t = i, X_{t+1} = j|O_{1:T}, \lambda)$$

- **Ensuring Sum to One:**

$$\sum_{i=1}^{N} \sum_{j=1}^{N} \gamma_t(i, j) = 1$$

Conclusion

- Dividing by the sum over the final α values ensures that $\gamma_t(i,j)$ correctly represents the conditional probability of transitioning from state i to state j at time t, given all observations.
- This normalization is essential for accurate parameter estimation in the Baum-Welch algorithm, enabling the model to converge and represent the data effectively.