B20AI013_HMM_assignment

1. Part 1 - Implementation of Baum Welch

Complete the code to read a given database of observations on the suspect in order to prepare a Hidden Markov Model that represents the behaviour of the suspect.

In this section, I implement the `LearnModel()` function in `hmm.py`.

The function takes in the database of all 1000 observation sequences in the form of a list of lists, and trains the parameters of the HMM model. This is done by **initializing the parameters** $\lambda = (A, B)$ of our HMM model to some initial values. The choice of these values is very important, the reason will be discussed shortly afterwards in this part.

After the initialization, the parameters are further trained in an unsupervised manner on the data of the observation sequences given using the **Baum-Welch algorithm.** It works by iteratively finding a maximum-likelihood estimate of the observation sequences. Essentially, we use an HMM to guess state labels and then use those state labels to fit a new HMM, over and over again, iteratively, until the HMM stabilizes.

Finding the maximum-likelihood estimate of the parameters is a non-convex problem, which means the solution obtained can be locally and not globally optimal. Where the parameters converge is highly dependent on the initial values. They ensure that there are patterns present in the HMM, and subsequent iterations determine which of those patterns are real, while replacing the ones that are not.

> 💡 Note:
> The code shown below for this question and the code in the submission .py file are slightly different. This is due to experiments with the initialization of the HMM showing that the model converges to a better global optimum that satisfies the assumptions given in the question, when the initial estimates for the parameters are unform / random, rather than carefully chosen. This is possible because the final convergence of the HMM is highly sensitive to the initial values and there may be some human bias in setting the initial values by hand as I have done in the code snippet below.
> When I tried initializing with equal values obeying the constraints, the HMM performed much better, converged much faster (1 iteration), and passed all the tests.
>
> However, one problem with this is that the resulting states will have no meaningful patterns in them so they wont have meaning in the same sense as the names assigned to them. This is because of the training being unsupervised, and the only supervision is in the initial conditions.
>
> Thus, I finally tried to use Softmaxing in order to smoothen the probabilities provided in the initial conditions while keeping the patterns. This seemed to work as the HMM converged very quickly (2 iterations).

The implementation of this is done as follows:

function FORWARD-BACKWARD(*observations* of len T, *output vocabulary* V, *hidden state set* Q) **returns** *HMM=(A,B)*

initialize A and B
iterate until convergence
E-step

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{\alpha_T(q_F)} \quad \forall\, t \text{ and } j$$

$$\xi_t(i,j) = \frac{\alpha_t(i)\, a_{ij} b_j(o_{t+1})\beta_{t+1}(j)}{\alpha_T(q_F)} \quad \forall\, t,\ i, \text{ and } j$$

M-step

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1}\sum_{k=1}^{N} \xi_t(i,k)}$$

$$\hat{b}_j(v_k) = \frac{\sum_{t=1 s.t.\ O_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

return A, B

Figure A.14 The forward-backward algorithm.

Source - HMM-book uploaded in classroom.

The corresponding code for the following is given below:

```
# initialize the model
    model = HMM()
    debug = False
    original_transitions = model._transitions
    original_emissions = model._emissions
    epsilon = 1
    gap_norm = np.inf
    iteration = 0
    while gap_norm < epsilon:
        iteration += 1
        print(f'iteration#{iteration}: {gap_norm:.3f}')
        previous_emissions = model._emissions.copy()
        previous_transitions = model._transitions.copy()
        A = model._transitions
        B = model.B
        N = model._N
        Pi = model.Pi
        obs_stats = {}
        ######################################################################## E -step
        for obs_index, obs_list in enumerate(dataset):
            T = len(obs_list)
            alpha = {}
            beta = {}
            eta = {}
            gamma = {}
            obs_stats[obs_index] = {
                'obs_list': obs_list,
                'alpha'   : alpha,
                'beta'    : beta,
                'eta'     : eta,
                'gamma'   : gamma,
                'T'       : T,
            }
            alpha[0] = np.array([[Pi(SuspectState(j+1)) * B(SuspectState(j+1), obs_list[0])] for j in range(N)])
            for t in range(1, T):
```

```
            # print('alpha', alpha[0].shape)
            alpha[t] = alpha[t-1] * np.array([[B(SuspectState(j+1), obs_list[t])] for j in range(N)])
            alpha[t] = A.T @ alpha[t]

        beta[T-1] = np.array([[1] for j in range(N)])
        # print(f'beta;{beta[T-1].shape}')
        for t in reversed(range(T-1)):
            beta[t] = A.T @ (beta[t+1] * np.array([[B(SuspectState(j+1), obs_list[t+1])] for j in range(N)]))
        # print(beta[0].shape)

        for t in range(T-1):
            eta[t] = {
                'numerator'  : alpha[t] * A * np.array([B(SuspectState(j+1), obs_list[t+1]) for j in range(N)]),
                'denominator': sum([alpha[t][j][0] * beta[t][j][0] for j in range(N)]),
            }
            # print(eta[t]['denominator'])
        for t in range(T):
            gamma[t] = {
                'numerator'  : (alpha[t] * beta[t]).flatten(),
                'denominator': np.sum([alpha[t][j] * beta[t][j] for j in range(N)]),
            }
            # print(gamma[t]['numerator'] / gamma[t]['denominator'])

    ########################################################################## M -step
    for i in range(N):
        for j in range(N):
            numerator = 0
            denominator = 0
            for obs_index in range(len(dataset)):
                obs_list = obs_stats[obs_index]['obs_list']
                alpha    = obs_stats[obs_index]['alpha']
                beta     = obs_stats[obs_index]['beta']
                gamma    = obs_stats[obs_index]['gamma']
                eta      = obs_stats[obs_index]['eta']
                T        = obs_stats[obs_index]['T']
                numerator += sum([eta[t]['numerator'][i][j] / eta[t]['denominator'] for t in range(T-1)])
                denominator += sum([sum([eta[t]['numerator'][i][k] / eta[t]['denominator'] for k in range(N)]) for t in range(T-1)])
            A[i][j] = numerator / denominator
            # A[i][j]  = sum([eta[t]['numerator'][i][j] / eta[t]['denominator'] for t in range(T-1)])
            # A[i][j] /= sum([sum([eta[t]['numerator'][i][k] / eta[t]['denominator'] for k in range(N)]) for t in range(T-1)])
        for j in range(N):
            for daytime in range(3): # + 6
                for action in range(4): # + 9
                    numerator = 0
                    denominator = 0
                    vk = Observation(Daytime(daytime+6), Action(action+9))
                    for obs_index in range(len(dataset)):
                        obs_list = obs_stats[obs_index]['obs_list']
                        alpha    = obs_stats[obs_index]['alpha']
                        beta     = obs_stats[obs_index]['beta']
                        gamma    = obs_stats[obs_index]['gamma']
                        eta      = obs_stats[obs_index]['eta']
                        T        = obs_stats[obs_index]['T']
                        # print(len([gamma[t]['numerator'][j]/gamma[t]['denominator'] for t in range(T) if obs_list[t] == vk]))
                        numerator += sum([gamma[t]['numerator'][j]/gamma[t]['denominator'] for t in range(T) if (obs_list[t].daytime == vk.daytime and obs_list[t].action == vk.action)])
                        denominator += sum([gamma[t]['numerator'][j]/gamma[t]['denominator'] for t in range(T)])
                    # print(denominator, vk.action, vk.daytime, obs_list[T-1].action, obs_list[T-1].daytime, numerator)
                    model._emissions[j-1][daytime][action] = numerator / denominator

        gap_norm = np.linalg.norm(model._emissions - previous_emissions) + np.linalg.norm(model._transitions - previous_transitions)
    if debug:
        print(model._transitions)
        print(model._emissions)


    return model
```

2. Part 2 - Implementation of Forward algorithm

This part deals with computing, given an HMM $\lambda = (A, B)$, the probability of observing a sequence of observations O. We use the Forward Algorithm in order to make the computation below much faster than an exhaustive search over all possible Q by utilizing simplifications afforded by the Markov Assumption and Output Independence.

$$\Sigma_{Q=q_1 q_2 ... q_T}\ P(o_1, o_2, ..., o_T | q_1, q_2, ..., q_T, \lambda)$$

This can be solved by decomposing the pattern as follows:

$$\begin{aligned}
&\Sigma_{Q=q_1 q_2 ... q_t}\ P(q_1, q_2, ..., q_{t+1} = j, o_1, o_2, ..., o_{t+1}, \lambda) \\
&= \Sigma_i\ \Sigma_{Q=q_1 q_2 ... q_t}\ P(q_1, q_2, ..., q_t = i, q_{t+1} = j, o_1, o_2, ..., o_t, \lambda) \\
&= \Sigma_i\ \Sigma_{Q=q_1 q_2 ... q_{t-1}}\ P(q_1, q_2, ..., q_t = i, o_1, o_2, ..., o_{t-1}, \lambda) * A_{ij} * b_j(o_t)
\end{aligned}$$

Thus, for the purposes of our computation, we can define:

$$\begin{aligned}
\alpha_1(j) &= \pi_j b_j(o_t), \quad 1 \leq j \leq N \\
\alpha_t(j) &= \Sigma_i \alpha_{t-1}(i) * A_{ij} * b_j(o_t), \quad 1 < t \leq T, 1 \leq j \leq N
\end{aligned}$$

and the solution to finding the probability of the sequence O, we compute:

$$P(O|\lambda) = \Sigma_i \alpha_T(i)$$

Code for the following is given below-

```
def Liklihood(model: HMM, obs_list: list) -> float:
    A = model._transitions
    B = model.B
    N = model._N
    Pi = model.Pi
    alpha = np.array([[Pi(SuspectState(i+1))] for i in range(N)])
    T = len(obs_list)
    for t in range(T):
        alpha  =  alpha * np.array([[B(SuspectState(j+1), obs_list[t])] for j in range(len(alpha))])
        if t == T-1:
            break
        alpha  =  A.T @ alpha
    return sum([alpha[i][0] for i in range(N)])
```

3. Part 3 - Implementation of Viterbi Algorithm

This part deals with computing, given an HMM $\lambda = (A, B)$, the Maximum Likelihood estimate for the state sequence that is most likely to produce a sequence of observations O. We use the Viterbi Algorithm in order to make the computation below much faster than an exhaustive search over all possible Q by utilizing simplifications afforded by the Markov Assumption and Output Independence.

$$\text{argmax}_{Q=q_1q_2...q_T}\ P(q_1, q_2, ..., q_T|o_1, o_2, ..., o_T, \lambda)$$

This can be solved by decomposing the pattern as follows:

$$\begin{gathered}\max_{Q=q_1q_2...q_t}\ P(q_1, q_2, ..., q_{t+1} = j, o_1, o_2, ..., o_{t+1}, \lambda) \\ = \max_i \max_{Q=q_1q_2...q_t}\ P(q_1, q_2, ..., q_t = i, q_{t+1} = j, o_1, o_2, ..., o_t, \lambda) \\ = \max_i \max_{Q=q_1q_2...q_{t-1}}\ P(q_1, q_2, ..., q_t = i, o_1, o_2, ..., o_{t-1}, \lambda) * A_{ij} * b_j(o_t)\end{gathered}$$

thus, for the purposes of our computation, we can define:

$$\begin{gathered}v_1(j) = \pi_j b_j(o_1), \quad \forall 1 \leq j \leq N \\ v_t(j) = \max_i\ v_{t-1}(i) * A_{ij} * b_j(o_t), \quad \forall 1 < t \leq T, 1 \leq j \leq N\end{gathered}$$

However, the original purpose was not to compute the maximum joint probability of state sequences, but to compute the exact state sequence for which the score is maximised. Hence, we also compute a backtrace:

$$\begin{gathered}bt_1(j) = 0, \quad \forall 1 \leq j \leq N \\ bt_t(j) = \text{argmax}_i\ v_{t-1}(i) * A_{ij} * b_j(o_t), \quad \forall 1 < t \leq T, 1 \leq j \leq N\end{gathered}$$

Finally, the Joint Probability of the Most Likely State Sequence, and the state sequence, can be computed using the following:

$$\begin{gathered}\text{score} = \max_i\ v_T(i) \\ q_T = \text{argmax}_i\ v_T(i)\end{gathered}$$

Code for the following is given below-

```
def GetHiddenStates(model: HMM, obs_list: list) -> list:
    N = model._N
    A = model._transitions
    B = model.B
    Pi = model.Pi
    viterbi = np.array([[Pi(SuspectState(i+1)) * B(SuspectState(i+1), obs_list[0])] for i in range(N)])
    backtraces = []
    state_sequence = []
    T = len(obs_list)
    for t in range(T):
        if t < T-1:
            backtrace  = np.argmax(viterbi.flatten() * A.T * np.array([[B(SuspectState(j+1), obs_list[t])] for j in range(N)]), axis=1, keepdims=True)
            viterbi      = np.max (viterbi.flatten() * A.T * np.array([[B(SuspectState(j+1), obs_list[t])] for j in range(N)]), axis=1, keepdims=True)
        else:
            backtrace  = np.argmax(viterbi.flatten())
            score = np.max(viterbi.flatten())
        backtraces.append(backtrace)

    state_list = []
    while len(backtraces):
        bt = backtraces.pop()
        state_list.append(SuspectState(bt+1))
        if len(backtraces) == 0:
            break
        backtraces[-1] = backtraces[-1][bt]
    state_list = state_list[::-1]
    return state_list
```

4. Part 4 - New HMM

Here, I have augmented the old HMM model with 1 new state, `Misc2`. This state is used to model the distinction between the miscellaneous tasks that precede a burglary (malicious miscellanous state) and miscellanous tasks that do not precede a burglary (benign miscellanous tasks).

- New Assumption 1 → After migrating, the burglar proceeds to enter the `Planning` state or the `Misc` state.
- New Assumption 2 → Right before the burglary, the burglar does some miscellaneous tasks related to the burglary in the `Misc2` state.

The choice of the state was based on observation of the initial assumptions and the observation sequences that were given in the database. The results upon adding this new state were that the new HMM model was not a very good fit on the data as well as the previous HMM model was. This can be seen in the decreased likelihood scores.

```
old likelihood-> 1.7213728110167658e-09
new likelihood-> 4.946263803852736e-10
```

A possible reason for this is that there are too many distributions being fit on the data.

Additionally, I came up with a measure for the sparsity of the model parameter matrices based on their L2 norm. The sparsity of both models came out roughly equal with this.

```
sparsity measure of the old transition matrix: 0.34847103310352157
sparsity measure of the new transition matrix: 0.3372701419110978
```

This means that both models are equally confident in their transitions from one state to another. However, it is expected that the new model is significantly trickier to converge due to the higher number of states which increases the complexity of the model.

Based on the experiments, it seems wise to keep the number of states as is or reduce them to 4 somehow in order to allow for a better, lesser confident modelling of uncertainty in states, so that the predicted states for a given sequence are more dependent on the sequence itself rather than the transition values.

All the code for the new HMM is given in the `new_hmm.py` file itself and will not be included here as it is too long for that.