EQ2341 Pattern Recognition and Machine Learning Assignment 4

Hang Qin
hangq@kth.se

Chenting Zhang
chzha@kth.se

May 22, 2023

1 Introduction

In this assignment, we are going to implement the Backward Algorithm and verify our implementation. In the forward algorithm implementation, we assumed that the hidden Markov model λ was already available. However, in practice, a HMM should be trained from the pre-recorded observed sequences. Therefore, backward variable $\beta_{i,t}$ and scaled backward variable $\hat{\beta}_{i,t}$ are introduced in addition to the previous calculated scaled forward variable $\hat{\alpha}_{i,t}$ and the forward scaled factor c_t to estimate probabilities of the hidden states $P[S_t = j|x_1, \ldots, x_t, \ldots, x_T, \lambda]$.

2 Backward Algorithm Implementation

Here we implement the backward algorithm following the steps presented in the textbook. Figure 1 shows the corresponding code. The first argument is the forward scaled factor c_t produced by the forward algorithm function. The second argument of our backward algorithm pX is the probability $b_j(x_t)$, which is the probability of the observed data at time step t are produced by the j^{th} state.

The backward algorithm contains three parts, initialization, forward step, and termination. And the iterative calculation procedure is mainly about the backward variable $\beta_{i,t}$ and its scaled version $\hat{\beta}_{i,t}$. The comments in the code illustrate that clearly. The comments also indicate which formula from the textbook the code implements by labeling it with the formula number. In the initialization part, the last state of the scaled backward variable $\hat{\beta}_{i,T}$ is calculated both for infinite-duration HMM and for finite-duration HMM. Then in the iterative part, $\hat{\beta}_{i,t}$ is derived from $\hat{\beta}_{i,t+1}$. Thus, $\hat{\beta}_{i,t}$ could be generated from $t = T - 1$ to $t = 1$ recursively. The forward scaled factor c_t plays the role as temporary variable in this implementation.

```python
def backward(self, c, p_x):
    beta_hat = []

    """initialization"""
    if self.is_finite:
        beta_hat0 = [beta/(c[-1]*c[-2]) for beta in self.A[:, -1]]  # eq 5.65
    else:
        beta_hat0 = [1/c[-1] for i in range(self.A.shape[0])]  # eq 5.64
    beta_hat.insert(0, beta_hat0)

    """backward step"""
    c = c[:-2] if self.is_finite else c[:-1]
    c.reverse()
    for t in range(len(c)):
        beta_hat_t = []
        for i in range(self.A.shape[0]):
            beta_i_t = sum([p_x[j, -t-1]*self.A[i, j]*beta_hat[0][j] for j in range(self.A.shape[0])])
            beta_hat_i_t = beta_i_t/c[t]
            beta_hat_t.append(beta_hat_i_t)  # eq 5.70
        beta_hat.insert(0, beta_hat_t)

    return beta_hat
```

Figure 1: The backward algorithm implementation

3 Backward Algorithm Verification

To verify our backward algorithm, we test it using the setup given by the textbook. The model parameters and observed data sequence are defined as follows.

$$q = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad A = \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0 & 0.9 & 0.1 \end{pmatrix} \quad x = \begin{pmatrix} -0.2 & 2.6 & 1.3 \end{pmatrix}$$

In addition to the scaled factors $c = \begin{pmatrix} 1 & 0.1625 & 0.8266 & 0.0581 \end{pmatrix}$ given by the forward algorithm. Feeding these results into the Backward algorithm, further computations in figure 2 show the final scaled backward variable $\hat{\beta}_{j,t}, t = 1 \ldots 3$. It is consistent with the results shown in the textbook, showing that our function works correctly and successfully implements the backward algorithm.

```
x = [-0.2  2.6  1.3]

q = [1 0]

A =
 [[0.9 0.1 0. ]
 [0.  0.9 0.1]]

alpha_hat =
 [[1.     0.3847 0.4189]
 [0.     0.6153 0.5811]]

c = [1.     0.1625 0.8266 0.0581]

beta_hat =
 [[1.     1.0389 0.    ]
 [8.4154 9.3504 2.0818]]

P(X = x|λ)= -9.187726979475208
```

Figure 2: The verify results