Attached files of the report:

test.py
train.py
hmm.py
ex02.ipynb
In the output folder: result1.txt, result2.txt
In the models folder: model_01.txt, model_02.txt, model_03.txt, model_04.txt, model_05.txt

From the model_init file, the information of the matrix A and B and the probability vector pi is read. These values are given as the initial value to each of the 5 models. Reading these parameters from the file is performed by the read_model function, which is implemented in both test.py and train.py. This function takes a string as an address in the input and then reads its contents and outputs in the form of matrices and vector A, B, pi. Using this information read from model_init, an initial HMM model is defined.
In train.py, the address of a training data file such as seq_model_0x.txt is given. The lines of this file are produced as a list of sequences, each of which is like a training data line. This list is given to the train_model function of the HMM class to train the model. It takes time for the entire training data to be read and training to be performed. This time is shown at the end.
After the model is ready, using the write_models_to_files function, it is written and saved in the folder whose address is given as an argument with the same name mentioned in the format of the parameters that were written in model_init.
Models are built and trained using the HMM class. This class is fully implemented in the hmm.py file as follows.
In this class, in addition to the 3 parameters mentioned, for the convenience of calculations, each model has a dic list of letters used in the models (1 to 6 uppercase letters of the English alphabet) and a number N, which is equal to the number of states. In this class, 4 functions are implemented.
The forward and backward functions, each of which work for the given input sequence and calculate the value of p(observation|model) with the model from which they are called. The forward function moves forward by calculating the alpha matrix and finally returns the alpha matrix itself and the total value of its last row as the p(observation|model) value. The backward function moves forward by calculating the beta matrix. And finally, it returns the sum of the product of the first line of beta in pi along with the beta matrix. Both of these functions use dynamic programming to calculate the probability of an input observation sequence for a given model.

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

formula

$$\beta[T][i] = 1$$

$$\beta[t][i] = \sum_{j=1}^{N} A[i][j]\, B[O_{t+1}][j]\, \beta[t+1][j]$$

implementation

$$P(O|\lambda) = \sum_{i=1}^{N} \pi_i \beta_0(i) = \sum_{i=1}^{N} P_i[i]\, \beta[0][i]$$

Probability estimation

$$\alpha_0(i) = \pi_i$$

$$\alpha_{t+1}(j) = \left(\sum_{i=1}^{N} \alpha_t(i)\, a_{ij} \right) b_j(O_{t+1})$$

formula

$$\alpha[0][i] = P_i[i]$$

$$\alpha[t+1][j] = \left(\sum_{i=1}^{N} \alpha[t][i] A[i][j] \right) \times B[O_{t+1}][j]$$

implementation

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i) = \sum_{i=1}^{N} \alpha[T][i]$$

Probability estimation

Another function implemented in HMM is the train_model function, which takes the content of the training file or a part of it as an input list and performs the training of the model on the model. In each step of the training, a list element equivalent to a line of the data file seq_model_0x.txt is read. Using the baum_welch function, the obtained values for the model parameters are calculated. The two returned matrices of values are in the form of pairs that, if divided, give the new values of A and B. But these values are stored in the same binary matrix so that the last line of training data is also read. All the binaries obtained from all the lines are added together and finally by dividing the first element by the second of these binaries in each row and column, the same number as that row and column in the NxN A and B matrices is

obtained. In order to get the error vector pi, it is enough to add all the errors obtained in each output of baum_welch together and finally divide by the number of loop steps (average). Finally, the new_A, new_B and new_pi values replace the A, B, pi values in the model, and the training on the input data ends. For the number of iterations it reads from the system input as the first argument, it goes over the training data and updates the model parameters.
The implemented baum_welch function implements the Baum-Welch algorithm for a given input line. Dynamic programming is used and with the help of alpha and beta matrices which are taken from forward and backward algorithms for the same training input and calculates the values of gamma and xi matrices with the current parameters of the model. For the given input, the first row of Gamma results as the new pi and for the two new matrices A and B, two NxN matrices, each row of which is a double, as I explained above. For the new A, for all rows of xi and gamma, the values should be added together and placed in the first and second bins of the binaries, respectively. For the new matrix B, the gamma values for all the rows and columns must be added to obtain the second matrix. If the desired row is an observation in the input that has a specific value such as row B, then the sum of their similar gammas is calculated as the first binary root of matrix B.

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

formula

$$\gamma[t][i] = \frac{\alpha[t][i]\,\beta[t][i]}{\sum_{j=1}^{N} \alpha[t][j]\,\beta[t][j]}$$

implementation

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

Formula

$$\xi_i[t][i][j] = \frac{\alpha[t][i]A[i][j]B[o_{t+1}][j]\beta[t+1][j]}{\sum_{t=1}^{N}\alpha[t][k]\beta[t][k]}$$

implementation

$$\pi_i^* = \gamma_1$$

$$A[i][j]^* = \frac{\sum_{t=1}^{T}\xi_i[t][i][j]}{\sum_{t=1}^{T}\gamma[t][i]} = \frac{\text{[illegible]}}{\text{[illegible]}}$$

$$B[i][j]^* = \frac{\sum_{t=1}^{T}\gamma[t][j] : \text{if } o_t = v_i}{\sum_{t=1}^{T}\gamma[t][j]}$$

Parameter estimation

In the test.py file, the model whose address is given is read from the text file with the read_model function. The test data file is read line by line from the given address and prepared as a list of sequences. The value of p(o|model) is calculated for each line for every 5 models. This calculation is done by measure_p_o_lambda. In write_results_to_file, the best result, i.e. the highest calculated probability, is selected, and the model that obtained that result is written to a file with the specified name and address along with the calculated probability size.
The files obtained by running these files are in the output folder. These files are generated with the following commands. The results obtained with 5 iterations:

```
python train.py 5 ./hmm_data/model_init.txt ./hmm_data/seq_model_01.txt ./models/model_01.txt
python train.py 5 ./hmm_data/model_init.txt ./hmm_data/seq_model_02.txt ./models/model_02.txt
python train.py 5 ./hmm_data/model_init.txt ./hmm_data/seq_model_03.txt ./models/model_03.txt
python train.py 5 ./hmm_data/model_init.txt ./hmm_data/seq_model_04.txt ./models/model_04.txt
python train.py 5 ./hmm_data/model_init.txt ./hmm_data/seq_model_05.txt ./models/model_05.txt
```

and to generate results and evaluate the model on two test sets:

```
python test.py ./hmm_data/modellist.txt ./hmm_data/testing_data1.txt ./output/result1.txt
```

```
python test.py ./hmm_data/modellist.txt ./hmm_data/testing_data2.txt ./output/result2.txt
```

The results of these executions are files in the models and output folders, respectively.
Before these files were prepared, all these codes for all 5 models and their construction and evaluation were written in ex02.ipynb. In this file, 5 models were available in the form of a dictionary named models. All the functions described in the HMM class are available on this lens. The train.py and test.py file functions exist here with a slight change, and the change is related to receiving the addresses of the files from the system input, in addition to training only one model in each run.
In this notebook, there is a function named measure_precision_on_testing_set1 that counts the final precision of result1.txt according to the answers of the testing_answer.txt file. I trained all 5 models with 5 iterations and calculated the accuracy. This accuracy is equal to 0.756