Indian Institute of Information Technology Surat

Lab Report on
Natural Language Processing (CS 601) Practical

Submitted by

[RAHUL KUMAR SINGH] (UI21CS44)

Course Faculty

Mrs. Nidhi Desai

Department of Computer Science and Engineering
Indian Institute of Information Technology Surat
Gujarat-394190, India

Jan-2024

Lab No: 6

Aim:

Hidden Markov Model. Generate transmission & emission matrix
Implement Hidden Markov Model on various time-series data to compare the efficiency and accuracy of the model. Also generate transmission and emission matrix while using plot visualization tools

Description:

- HMM infers hidden states based on observed data, where states influence the observable symbols.
- **Transmission Probabilities:** Likelihood of transitioning between hidden states, updated during training to maximize observation sequence likelihood.
- **Emission Probabilities:** Likelihood of an observation being generated by a hidden state, refined during training.
- **Expectation-Maximization (EM) Algorithm:** Iterative process with expectation (compute expected state probabilities) and maximization (update model parameters) to improve model fit.
- **Forward Algorithms:** It calculates observation probability up to a point,
- **Backward Algorithms:** It calculates probability for future observations, aiding in state probability estimation.
- Likelihood Calculation is computed using the forward algorithm after each iteration to evaluate model fit, higher likelihood means better fit.
- **State Probabilities (gamma):** Gamma represents state probability at each time step.
- **Transition Probabilities (psi):** Psi represents transition probability between states at each time step.

Source Code:

```
import copy
import numpy as np
import logging

for handler in logging.root.handlers[:]:
    logging.root.removeHandler(handler)
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
class HMM():
    def __init__(self, transmission_prob, emission_prob, obs=None):
        self.transmission_prob = transmission_prob
        self.emission_prob = emission_prob
        self.n = self.emission_prob.shape[1]
        self.m = self.emission_prob.shape[0]
        self.observations = None
        self.forward = []
        self.backward = []
        self.psi = []
        self.obs = obs
        self.emiss_ref = {}
        self.forward_final = [0 , 0]
        self.backward_final = [0 , 0]
```

```
        self.state_probs = []
        if obs is None and self.observations is not None:
            self.obs = self.assume_obs()
    def assume_obs(self):
        obs = list(set(list(self.observations)))
        obs.sort()
        for i in range(len(obs)):
            self.emiss_ref[obs[i]] = i
        return obs
    def train(self, observations, iterations=10, verbose=True):
        self.observations = observations
        self.obs = self.assume_obs()
        self.psi = [[[0.0] * (len(self.observations)-1) for _ in range(self.n)] for _ in
range(self.n)]
        self.gamma = [[0.0] * len(self.observations) for _ in range(self.n)]
        for i in range(iterations):
            logging.info("Starting iteration {}".format(i + 1))
            old_transmission = self.transmission_prob.copy()
            old_emission = self.emission_prob.copy()
            self.expectation()
            logging.info("Expectation step completed")
            self.maximization()
            logging.info("Maximization step completed")
            likelihood = self.likelihood(self.observations)
            logging.info("Likelihood after iteration {}: {}".format(i + 1, likelihood))
            if verbose:
                print("Iteration {}: Likelihood = {}".format(i + 1, likelihood))
    def expectation(self):
        logging.info("Calculating forward probabilities")
        self.forward = self.forward_recurse(len(self.observations))
        logging.info("Forward probabilities calculated")
        logging.info("Calculating backward probabilities")
        self.backward = self.backward_recurse(0)
        logging.info("Backward probabilities calculated")
        logging.info("Calculating gamma values")
        self.get_gamma()
        logging.info("Gamma values calculated")
        logging.info("Calculating psi values")
        self.get_psi()
        logging.info("Psi values calculated")
    def get_gamma(self):
        self.gamma = [[0, 0] for i in range(len(self.observations))]
        for i in range(len(self.observations)):
            self.gamma[i][0] = (float(self.forward[0][i] * self.backward[0][i]) /
                                float(self.forward[0][i] * self.backward[0][i] +
                                self.forward[1][i] * self.backward[1][i]))
            self.gamma[i][1] = (float(self.forward[1][i] * self.backward[1][i]) /
                                float(self.forward[0][i] * self.backward[0][i] +
                                self.forward[1][i] * self.backward[1][i]))
    def get_psi(self):
        for t in range(1, len(self.observations)):
            for j in range(self.n):
                for i in range(self.n):
```

```
                    self.psi[i][j][t-1] = self.calculate_psi(t, i, j)
    def calculate_psi(self, t, i, j):
        alpha_tminus1_i = self.forward[i][t-1]
        a_i_j = self.transmission_prob[j+1][i+1]
        beta_t_j = self.backward[j][t]
        observation = self.observations[t]
        b_j = self.emission_prob[self.emiss_ref[observation]][j]
        denom = float(self.forward[0][i] * self.backward[0][i] + self.forward[1][i] *
self.backward[1][i])
        return (alpha_tminus1_i * a_i_j * beta_t_j * b_j) / denom
    def maximization(self):
        logging.info("Calculating state probabilities")
        self.get_state_probs()
        logging.info("State probabilities calculated")
        logging.info("Updating transmission probabilities")
        for i in range(self.n):
            self.transmission_prob[i+1][0] = self.gamma[0][i]
            self.transmission_prob[-1][i+1] = self.gamma[-1][i] / self.state_probs[i]
            for j in range(self.n):
                self.transmission_prob[j+1][i+1] = self.estimate_transmission(i, j)
        logging.info("Transmission probabilities updated")
        logging.info("Updating emission probabilities")
        for obs in range(self.m):
            for i in range(self.n):
                self.emission_prob[obs][i] = self.estimate_emission(i, obs)
        logging.info("Emission probabilities updated")
    def get_state_probs(self):
        self.state_probs = [0] * self.n
        for state in range(self.n):
            summ = 0
            for row in self.gamma:
                summ += row[state]
            self.state_probs[state] = summ
    def estimate_transmission(self, i, j):
        return sum(self.psi[i][j]) / self.state_probs[i]
    def estimate_emission(self, j, observation):
        observation = self.obs[observation]
        ts = [i for i in range(len(self.observations)) if self.observations[i] == observation]
        for i in range(len(ts)):
            ts[i] = self.gamma[ts[i]][j]
        return sum(ts) / self.state_probs[j]
    def backward_recurse(self, index):
        if index == (len(self.observations) - 1):
            backward = [[0.0] * (len(self.observations)) for i in range(self.n)]
            for state in range(self.n):
                backward[state][index] = self.backward_initial(state)
            return backward
        else:
            backward = self.backward_recurse(index+1)
            for state in range(self.n):
                if index >= 0:
                    backward[state][index] = self.backward_probability(index, backward, state)
                if index == 0:
```

```
                    self.backward_final[state] = self.backward_probability(index, backward, 0,
final=True)
            return backward
    def backward_initial(self, state):
        return self.transmission_prob[self.n + 1][state + 1]
    def backward_probability(self, index, backward, state, final=False):
        p = [0] * self.n
        for j in range(self.n):
            observation = self.observations[index + 1]
            if not final:
                a = self.transmission_prob[j + 1][state + 1]
            else:
                a = self.transmission_prob[j + 1][0]
            b = self.emission_prob[self.emiss_ref[observation]][j]
            beta = backward[j][index + 1]
            p[j] = a * b * beta
        return sum(p)
    def forward_recurse(self, index):
        if index == 0:
            forward = [[0.0] * (len(self.observations)) for i in range(self.n)]
            for state in range(self.n):
                forward[state][index] = self.forward_initial(self.observations[index], state)
            return forward
        else:
            forward = self.forward_recurse(index-1)
            for state in range(self.n):
                if index != len(self.observations):
                    forward[state][index] = self.forward_probability(index, forward, state)
                else:
                    self.forward_final[state] = self.forward_probability(index, forward, state,
final=True)
            return forward
    def forward_initial(self, observation, state):
        self.transmission_prob[state + 1][0]
        self.emission_prob[self.emiss_ref[observation]][state]
        return self.transmission_prob[state + 1][0] *
self.emission_prob[self.emiss_ref[observation]][state]
    def forward_probability(self, index, forward, state, final=False):
        p = [0] * self.n
        for prev_state in range(self.n):
            if not final:
                obs_index = self.emiss_ref[self.observations[index]]
                p[prev_state] = forward[prev_state][index-1] * self.transmission_prob[state +
1][prev_state + 1] * self.emission_prob[obs_index][state]
            else:
                p[prev_state] = forward[prev_state][index-1] *
self.transmission_prob[self.n][prev_state + 1]
        return sum(p)
    def likelihood(self, new_observations):
        new_hmm = HMM(self.transmission_prob, self.emission_prob)
        new_hmm.observations = new_observations
        new_hmm.obs = new_hmm.assume_obs()
        forward = new_hmm.forward_recurse(len(new_observations))
```

```
        return sum(new_hmm.forward_final)
if __name__ == '__main__':
    state = hmmdata.copy()
    emission = np.array([[0.7, 0], [0.2, 0.3], [0.1, 0.7]])
    transmission = np.array([ [0, 0, 0, 0], [0.5, 0.8, 0.2, 0], [0.5, 0.1, 0.7, 0], [0, 0.1,
0.1, 0]])
    observations = ['1','1','1','2','3','3','2','3','2','2']
    model = HMM(transmission, emission)
    model.train(observations)
    print("Model transmission probabilities:\n{}".format(model.transmission_prob))
    print("Model emission probabilities:\n{}".format(model.emission_prob))
    new_seq = ['1','1','1']
    print("Finding likelihood for {}".format(new_seq))
    likelihood = model.likelihood(new_seq)
    print("Likelihood: {}".format(likelihood))
```

Input:

observations = ['1','1','1','2','3','3','2','3','2','2']
new_seq = ['1','1','1']

Output:

```
2024-11-16 17:59:58,860 - INFO - Starting iteration 10
2024-11-16 17:59:58,862 - INFO - Calculating forward probabilities
2024-11-16 17:59:58,865 - INFO - Forward probabilities calculated
2024-11-16 17:59:58,866 - INFO - Calculating backward probabilities
2024-11-16 17:59:58,868 - INFO - Backward probabilities calculated
2024-11-16 17:59:58,879 - INFO - Calculating gamma values
2024-11-16 17:59:58,881 - INFO - Gamma values calculated
2024-11-16 17:59:58,890 - INFO - Calculating psi values
2024-11-16 17:59:58,891 - INFO - Psi values calculated
2024-11-16 17:59:58,894 - INFO - Expectation step completed
2024-11-16 17:59:58,898 - INFO - Calculating state probabilities
2024-11-16 17:59:58,901 - INFO - State probabilities calculated
2024-11-16 17:59:58,905 - INFO - Updating transmission probabilities
2024-11-16 17:59:58,907 - INFO - Transmission probabilities updated
2024-11-16 17:59:58,911 - INFO - Updating emission probabilities
2024-11-16 17:59:58,915 - INFO - Emission probabilities updated
2024-11-16 17:59:58,916 - INFO - Maximization step completed
2024-11-16 17:59:58,918 - INFO - Likelihood after iteration 10: 0.000845027146240598
Iteration 8: Likelihood = 0.0008439331177776056
Iteration 9: Likelihood = 0.0008446859961555067
Iteration 10: Likelihood = 0.000845027146240598
Model transmission probabilities:
[[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.00000000e+00 6.66718861e-01 7.14909177e-26 0.00000000e+00]
 [0.00000000e+00 3.33281139e-01 8.57133268e-01 0.00000000e+00]
 [0.00000000e+00 7.44765000e-98 1.42866732e-01 0.00000000e+00]]
Model emission probabilities:
[[9.99843418e-01 0.00000000e+00]
 [1.56582005e-04 5.71399805e-01]
 [2.08155714e-21 4.28600195e-01]]
Finding likelihood for ['1', '1', '1']
Likelihood: 0.2961571286380619
```

Conclusion:

- The model uses EM to optimize transmission and emission probabilities.
- It handles observation sequences through efficient probability calculations.
- Computes the likelihood of sequences, useful for sequence prediction tasks.
- Encapsulates HMM functionality for potential extensions, like decoding.