## 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)**

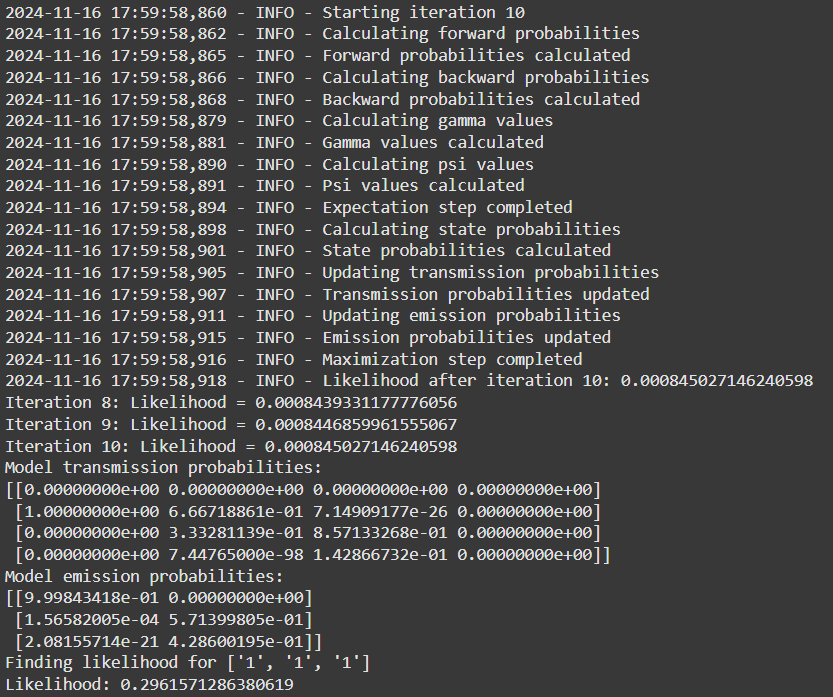
**print("Likelihood: {}".format(likelihood))**

## Input:

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

new\_seq = ['1','1','1']

## Output:

****

## 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.