```python
# A class for performing hidden markov models

import copy
import numpy as np

class HMM():

    def __init__(self, transmission_prob, emission_prob, obs=None):
        '''
        Note that this implementation assumes that n, m, and T are small
        enough not to require underflow mitigation.

        Required Inputs:
        - transmission_prob: an (n+2) x (n+2) numpy array, initial, where n is
        the number of hidden states
        - emission_prob: an (m x n) 2-D numpy array, where m is the number of
        possible observations

        Optional Input:
        - obs: a list of observation labels, in the same order as their
        occurence within the emission probability matrix; otherwise, will assume
        that the emission probabilities are in alpha-numerical order.
        '''
        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):
        '''
        If observation labels are not given, will assume that the emission
        probabilities are in alpha-numerical order.
        '''
        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):
        '''
        Trains the model parameters according to the observation sequence.

        Input:
        - observations: 1-D string array of T observations
        '''
        self.observations = observations
        self.obs = self.assume_obs()
        self.psi = [[[0.0] * (len(self.observations)-1) for i in range(self.n)] for i in range(self.n)]
        self.gamma = [[0.0] * (len(self.observations)) for i in range(self.n)]
        for i in range(iterations):
            old_transmission = self.transmission_prob.copy()
            old_emission = self.emission_prob.copy()
            if verbose:
                print("Iteration: {}".format(i + 1))
            self.expectation()
            self.maximization()

    def expectation(self):
        '''
        Executes expectation step.
        '''
        self.forward = self.forward_recurse(len(self.observations))
        self.backward = self.backward_recurse(0)
        self.get_gamma()
        self.get_psi()

    def get_gamma(self):
        '''
        Calculates the gamma matrix.
        '''
```

```
        '''
        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):
        '''
        Runs the psi calculation.
        '''
        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):
        '''
        Calculates the psi for a transition from i->j for t > 0.
        '''
        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):
        '''
        Executes maximization step.
        '''
        self.get_state_probs()
        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)
            for obs in range(self.m):
                self.emission_prob[obs][i] = self.estimate_emission(i, obs)

    def get_state_probs(self):
        '''
        Calculates total probability of a given state.
        '''
        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):
        '''
        Estimates transmission probabilities from i to j.
        '''
        return sum(self.psi[i][j]) / self.state_probs[i]

    def estimate_emission(self, j, observation):
        '''
        Estimate emission probability for an observation from state j.
        '''
        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):
        '''
        Runs the backward recursion.
        '''
        # Initialization at T
        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
        # Recursion for T --> 0
        else:
```

```python
            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):
        '''
        Initialization of backward probabilities.
        '''
        return self.transmission_prob[self.n + 1][state + 1]

    def backward_probability(self, index, backward, state, final=False):
        '''
        Calculates the backward probability at index = t.
        '''
        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):
        '''
        Executes forward recursion.
        '''
        # Initialization
        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
        # Recursion
        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:
                    # Termination
                    self.forward_final[state] = self.forward_probability(index, forward, state, final=True)
            return forward

    def forward_initial(self, observation, state):
        '''
        Calculates initial forward probabilities.
        '''
        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):
        '''
        Calculates the alpha for index = t.
        '''
        p = [0] * self.n
        for prev_state in range(self.n):
            if not final:
                # Recursion
                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
            else:
                # Termination
                p[prev_state] = forward[prev_state][index-1] * self.transmission_prob[self.n][prev_state + 1]
        return sum(p)

    def likelihood(self, new_observations):
        '''
        Returns the probability of a observation sequence based on current model
        parameters.
        '''
        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))
```

```
        forward = new_hmm.forward_recurse(len(new_observations))
        return sum(new_hmm.forward_final)

if __name__ == '__main__':
    # Example inputs from Jason Eisner's Ice Cream and Baltimore Summer example
    # http://www.cs.jhu.edu/~jason/papers/#eisner-2002-tnlp
    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 = ['2','3','3','2','3','2','3','2','2','3','1','3','3','1','1',
                    '1','2','1','1','1','3','1','2','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))
    # Probability of a new sequence
    new_seq = ['1', '2', '3']
    print("Finding likelihood for {}".format(new_seq))
    likelihood = model.likelihood(new_seq)
    print("Likelihood: {}".format(likelihood))
```

```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
Iteration: 6
Iteration: 7
Iteration: 8
Iteration: 9
Iteration: 10
Model transmission probabilities:
[[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.44069481e-13 9.33776929e-01 7.18678407e-02 0.00000000e+00]
 [1.00000000e+00 6.62230707e-02 8.64943107e-01 0.00000000e+00]
 [0.00000000e+00 3.10801009e-14 6.31890522e-02 0.00000000e+00]]
Model emission probabilities:
[[0.64048542 0.        ]
 [0.14806851 0.5343899 ]
 [0.21144608 0.4656101 ]]
Finding likelihood for ['1', '2', '3']
Likelihood: 3.2956914388507033e-15
```

```
pip install hmmlearn
```

```
Collecting hmmlearn
  Downloading hmmlearn-0.3.3-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (3.0 kB)
Requirement already satisfied: numpy>=1.10 in /usr/local/lib/python3.11/dist-packages (from hmmlearn) (2.0.2)
Requirement already satisfied: scikit-learn!=0.22.0,>=0.16 in /usr/local/lib/python3.11/dist-packages (from hmmlearn) (1.6.1)
Requirement already satisfied: scipy>=0.19 in /usr/local/lib/python3.11/dist-packages (from hmmlearn) (1.14.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn!=0.22.0,>=0.16->hmmlearn
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn!=0.22.0,>=0.16->hr
Downloading hmmlearn-0.3.3-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (165 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 165.9/165.9 kB 5.7 MB/s eta 0:00:00
Installing collected packages: hmmlearn
Successfully installed hmmlearn-0.3.3
```

```
import numpy as np
from hmmlearn import hmm

# Define the number of hidden states and observation symbols
n_states = 3  # number of hidden states
n_observations = 4  # number of possible observation symbols

# Example observations (could be any discrete data, for example: [0, 1, 2, 3] mapping to different observations)
observations = np.array([[0], [1], [2], [3], [0], [1]])

# Transition matrix (state transitions)
# The rows represent the current state, and the columns represent the next state
transition_probabilities = np.array([
    [0.7, 0.2, 0.1],
    [0.3, 0.4, 0.3],
    [0.4, 0.3, 0.3]
])

# Emission matrix (probabilities of observing a symbol given a state)
# The rows represent the states, and the columns represent the observation symbols
emission_probabilities = np.array([
    [0.6, 0.1, 0.2, 0.1],
    [0.1, 0.4, 0.4, 0.1],
    [0.3, 0.3, 0.2, 0.2]
])
```

```
# Initial state distribution (probability of starting in each state)
initial_probabilities = np.array([0.5, 0.3, 0.2])

# Create and fit the model
model = hmm.MultinomialHMM(n_components=n_states, n_iter=1000)
model.startprob_ = initial_probabilities
model.transmat_ = transition_probabilities
model.emissionprob_ = emission_probabilities

# Fit the model with the observation sequence
model.fit(observations)

# Predict the hidden states given the observations
hidden_states = model.predict(observations)

print("Observations:", observations.T)
print("Predicted Hidden States:", hidden_states)
```

```
WARNING:hmmlearn.hmm:MultinomialHMM has undergone major changes. The previous version was implementing a CategoricalHMM (a special c
https://github.com/hmmlearn/hmmlearn/issues/335
https://github.com/hmmlearn/hmmlearn/issues/340
WARNING:hmmlearn.base:Even though the 'startprob_' attribute is set, it will be overwritten during initialization because 'init_para
WARNING:hmmlearn.base:Even though the 'transmat_' attribute is set, it will be overwritten during initialization because 'init_paran
WARNING:hmmlearn.base:Fitting a model with 8 free scalar parameters with only 6 data points will result in a degenerate solution.
Observations: [[0 1 2 3 0 1]]
Predicted Hidden States: [1 2 1 2 1 2]
```

Start coding or generate with AI.