9.Implementing a Hidden Markov Model (HMM) for a Simple Application.

A statistical model called a Hidden Markov Model (HMM) is used to describe systems with changing unobservable states over time. It is predicated on the idea that there is an underlying process with concealed states, each of which has a known result. Probabilities for switching between concealed states and emitting observable symbols are defined by the model.

Because of their superior ability to capture uncertainty and temporal dependencies, HMMs are used in a wide range of industries, including finance, bioinformatics, and speech recognition. HMMs are useful for modelling dynamic systems and forecasting future states based on sequences that have been seen because of their flexibility.

**Hidden Markov Model in Machine Learning**

**The hidden Markov Model** (HMM) is a [statistical model](https://www.geeksforgeeks.org/difference-between-statistical-model-and-machine-learning/) that is used to describe the probabilistic relationship between a sequence of observations and a sequence of hidden states. It is often used in situations where the underlying system or process that generates the observations is unknown or hidden, hence it has the name “Hidden Markov Model.”

It is used to predict future observations or classify sequences, based on the underlying hidden process that generates the data.

An HMM consists of two types of variables: hidden states and observations.

* The **hidden states** are the underlying variables that generate the observed data, but they are not directly observable.
* The **observations** are the variables that are measured and observed.

The relationship between the hidden states and the observations is modeled using a probability distribution. The Hidden Markov Model (HMM) is the relationship between the hidden states and the observations using two sets of probabilities: the transition probabilities and the emission probabilities.

* The **transition probabilities**describe the probability of transitioning from one hidden state to another.
* The **emission probabilities** describe the probability of observing an output given a hidden state.

**Hidden Markov Model  Algorithm**

The Hidden Markov Model (HMM) algorithm can be implemented using the following steps:

**Step 1: Define the state space and observation space**

* The state space is the set of all possible hidden states, and the observation space is the set of all possible observations.

**Step 2: Define the initial state distribution**

* This is the probability distribution over the initial state.

**Step 3: Define the state transition probabilities**

* These are the probabilities of transitioning from one state to another. This forms the transition matrix, which describes the probability of moving from one state to another.

**Step 4: Define the observation likelihoods:**

* These are the probabilities of generating each observation from each state. This forms the emission matrix, which describes the probability of generating each observation from each state.

**Step 5: Train the model**

* The parameters of the state transition probabilities and the observation likelihoods are estimated using the Baum-Welch algorithm, or the forward-backward algorithm. This is done by iteratively updating the parameters until convergence.

**Step 6: Decode the most likely sequence of hidden states**

* Given the observed data, the Viterbi algorithm is used to compute the most likely sequence of hidden states. This can be used to predict future observations, classify sequences, or detect patterns in sequential data.

**Step 7: Evaluate the model**

* The performance of the HMM can be evaluated using various metrics, such as accuracy, precision, recall, or F1 score.

To summarise, the HMM algorithm involves defining the state space, observation space, and the parameters of the state transition probabilities and observation likelihoods, training the model using the Baum-Welch algorithm or the forward-backward algorithm, decoding the most likely sequence of hidden states using the Viterbi algorithm, and evaluating the performance of the model.

**Implementation in Python**

Key steps in the Python implementation of a simple [Hidden Markov Model](https://www.geeksforgeeks.org/markov-chains-in-nlp/) (HMM) using the hmmlearn library.

**Example 1. Predicting the weather**

**Problem statement**: Given the historical data on weather conditions, the task is to predict the weather for the next day based on the current day’s weather.

pip install hmmlearn

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from hmmlearn import hmm

# Define the state space

states = ["Sunny", "Rainy"]

n\_states = len(states)

print('Number of hidden states :',n\_states)

# Define the observation space

observations = ["Dry", "Wet"]

n\_observations = len(observations)

print('Number of observations :',n\_observations)

# Define the initial state distribution

state\_probability = np.array([0.6, 0.4])

print("State probability: ", state\_probability)

# Define the state transition probabilities

transition\_probability = np.array([[0.7, 0.3] ,[0.3, 0.7]])

print("\nTransition probability:\n", transition\_probability)

# Define the observation likelihoods

emission\_probability= np.array([[0.9, 0.1],[0.2, 0.8]])

print("\nEmission probability:\n", emission\_probability)

model = hmm.CategoricalHMM(n\_components=n\_states)

model.startprob\_ = state\_probability

model.transmat\_ = transition\_probability

model.emissionprob\_ = emission\_probability

# Define the sequence of observations

observations\_sequence = np.array([0, 1, 0, 1, 0, 0]).reshape(-1, 1)

observations\_sequence

# Predict the most likely sequence of hidden states

hidden\_states = model.predict(observations\_sequence)

print("Most likely hidden states:", hidden\_states)

log\_probability, hidden\_states = model.decode(observations\_sequence, lengths = len(observations\_sequence),

algorithm ='viterbi' )

print('Log Probability :',log\_probability)

print("Most likely hidden states:", hidden\_states)

**10.Implement a MDP(Markov Decision Process) to run value and policy iteration in any environment.**

To implement a Markov Decision Process (MDP) solver that uses value iteration and policy iteration, we first need to understand the components of an MDP. An MDP is defined by:

1. States (S): A finite set of states.

2. Actions (A): A finite set of actions.

3. Transition Model (P): The probability \(P(s' | s, a)\) of transitioning to state \(s'\) from state \(s\) by taking action \(a\).

4. Reward Function (R): The immediate reward received after transitioning from state \(s\) to state \(s'\) by taking action \(a\).

5. Discount Factor (\(\gamma\)): A factor \(\gamma \in [0, 1]\) that represents the present value of future rewards.

Value Iteration

Value iteration computes the optimal state value function by iteratively updating the value of each state based on the Bellman equation until convergence.

Policy Iteration

Policy iteration alternates between policy evaluation (computing the value of the current policy) and policy improvement (updating the policy based on the computed values) until the policy stabilizes.

**Implementation**

Here’s a Python implementation of value iteration and policy iteration for a generic MDP environment.

python

importnumpy as np

class MDP:

def \_\_init\_\_(self, states, actions, transition\_prob, rewards, gamma):

self.states = states

self.actions = actions

self.transition\_prob = transition\_prob # P(s' | s, a)

self.rewards = rewards # R(s, a, s')

self.gamma = gamma

defvalue\_iteration(self, epsilon=1e-6):

V = np.zeros(len(self.states))

while True:

delta = 0

for s in self.states:

v = V[s]

V[s] = max(sum(self.transition\_prob[s][a][s\_next] \* (self.rewards[s][a][s\_next] + self.gamma \* V[s\_next])

fors\_next in self.states)

for a in self.actions)

delta = max(delta, abs(v - V[s]))

if delta < epsilon:

break

return V

defpolicy\_iteration(self):

policy = np.zeros(len(self.states), dtype=int)

V = np.zeros(len(self.states))

defone\_step\_lookahead(s, V):

A = np.zeros(len(self.actions))

for a in self.actions:

A[a] = sum(self.transition\_prob[s][a][s\_next] \* (self.rewards[s][a][s\_next] + self.gamma \* V[s\_next])

fors\_next in self.states)

return A

while True:

# Policy Evaluation

while True:

delta = 0

for s in self.states:

v = V[s]

V[s] = sum(self.transition\_prob[s][policy[s]][s\_next] \* (self.rewards[s][policy[s]][s\_next] + self.gamma \* V[s\_next])

fors\_next in self.states)

delta = max(delta, abs(v - V[s]))

if delta < 1e-6:

break

# Policy Improvement

policy\_stable = True

for s in self.states:

old\_action = policy[s]

policy[s] = np.argmax(one\_step\_lookahead(s, V))

ifold\_action != policy[s]:

policy\_stable = False

ifpolicy\_stable:

return policy, V

# Example usage

states = [0, 1, 2]

actions = [0, 1]

transition\_prob = {

0: {0: {0: 0.8, 1: 0.2}, 1: {1: 0.9, 2: 0.1}},

1: {0: {0: 0.7, 2: 0.3}, 1: {0: 0.4, 2: 0.6}},

2: {0: {1: 0.5, 2: 0.5}, 1: {0: 0.6, 1: 0.4}}

}

rewards = {

0: {0: {0: 5, 1: 10}, 1: {1: -1, 2: 2}},

1: {0: {0: -1, 2: 1}, 1: {0: 1, 2: 3}},

2: {0: {1: -2, 2: 0}, 1: {0: 0, 1: 1}}

}

gamma = 0.9

mdp = MDP(states, actions, transition\_prob, rewards, gamma)

# Value Iteration

optimal\_values = mdp.value\_iteration()

print("Optimal Values from Value Iteration:", optimal\_values)

# Policy Iteration

optimal\_policy, optimal\_values = mdp.policy\_iteration()

print("Optimal Policy from Policy Iteration:", optimal\_policy)

print("Optimal Values from Policy Iteration:", optimal\_values)

```

Explanation

1. MDP Class: Initializes with states, actions, transition probabilities, rewards, and the discount factor.

2. Value Iteration: Iteratively updates the value function until the changes are smaller than a threshold (`epsilon`).

3. Policy Iteration: Alternates between evaluating the current policy and improving it until the policy stabilizes.

4. Example Usage: Demonstrates the use of both value iteration and policy iteration on a small example MDP.

This implementation can be extended or modified for different environments by adjusting the states, actions, transition probabilities, and rewards. Let me know if you have any questions or need further customization!