MI LAB6

GMM

NAME : Nagaveni L G

SRN :PES2UG21CS315

SEC :5F

Code:

```
import torch
import numpy as np

class GMMModel:
    def __init__(self, n_components):
        '''
        Initialize the Gaussian Mixture Model (GMM).

        Parameters:
            n_components (int): Number of Gaussian components.

        Attributes:
            n_components (int): Number of Gaussian components.
            weights (torch.Tensor): Initial weights for each Gaussian
component.
            means (torch.Tensor): Initial means for each Gaussian component.
            covariances (torch.Tensor): Initial covariances for each Gaussian
component.
        '''
        self.n_components = n_components
        self.weights = torch.ones(n_components) / n_components
        self.means = torch.randn(n_components, 3)
        self.covariances = torch.zeros(n_components, 3, 3)

    def fit(self, X, max_iters=100, tol=1e-4):
        '''
        Fit the Gaussian Mixture Model to the input data.

        Parameters:
            X (torch.Tensor): Input data of shape (n_samples, n_features).
```

```
            max_iters (int, optional): Maximum number of iterations for the EM
algorithm. Default is 100.
            tol (float, optional): Convergence tolerance for parameter
updates. Default is 1e-4.
        '''
        n_samples = X.shape[0]
        n_features = X.shape[1]

        for iteration in range(max_iters):
            # Expectation step
            responsibilities = self._e_step(X)

            # Maximization step
            self._m_step(X, responsibilities)

            if self._is_converged(X, responsibilities, tol):
                break

    def _e_step(self, X):
        '''
        Perform the Expectation step.

        Parameters:
            X (torch.Tensor): Input data of shape (n_samples, n_features).

        Returns:
            torch.Tensor: Responsibilities of shape (n_components, n_samples).
        '''
        S_responsibilities = torch.zeros(self.n_components, X.shape[0])

        for k in range(self.n_components):
            numerator = self.weights[k] * torch.exp(-0.5 * torch.sum(((X -
self.means[k]) @ self._inverse(self.covariances[k])) * (X - self.means[k]),
dim=1))
            S_responsibilities[k] = numerator

        S_responsibilities = S_responsibilities / S_responsibilities.sum(0)
        return S_responsibilities

    def _m_step(self, X, responsibilities):
        '''
        Perform the Maximization step.

        Parameters:
            X (torch.Tensor): Input data of shape (n_samples, n_features).
            responsibilities (torch.Tensor): Responsibilities of shape
(n_components, n_samples).
        '''
```

```
        self.weights = responsibilities.mean(1)

        for i in range(self.n_components):
            self.means[i] = torch.sum(responsibilities[i, None].T * X, dim=0)
/ responsibilities[i].sum()
            diff = X - self.means[i]
            self.covariances[i] = (responsibilities[i, None].T * diff).T @
diff / responsibilities[i].sum()

    def _inverse(self, matrix):
        '''
        Calculate the inverse of a matrix with regularization.

        Parameters:
            matrix (torch.Tensor): Input matrix.

        Returns:
            torch.Tensor: Inverse of the input matrix.
        '''
        return torch.inverse(matrix + torch.eye(matrix.shape[0]) * 1e-6)

    def _is_converged(self, X, responsibilities, tol):
        '''
        Check for convergence based on log likelihood.

        Parameters:
            X (torch.Tensor): Input data of shape (n_samples, n_features).
            responsibilities (torch.Tensor): Responsibilities of shape
(n_components, n_samples).
            tol (float): Convergence tolerance.

        Returns:
            bool: True if the model has converged, False otherwise.
        '''
        prev_log_likelihood = self._log_likelihood(X, responsibilities)
        responsibilities = self._e_step(X)
        current_log_likelihood = self._log_likelihood(X, responsibilities)
        return abs(current_log_likelihood - prev_log_likelihood) < tol

    def _log_likelihood(self, X, responsibilities):
        '''
        Calculate the log likelihood of the data.

        Parameters:
            X (torch.Tensor): Input data of shape (n_samples, n_features).
            responsibilities (torch.Tensor): Responsibilities of shape
(n_components, n_samples).
```

```
        Returns:
            float: Log likelihood of the data.
        '''
        log_likelihood = torch.log(responsibilities.sum(0)).sum()
        return log_likelihood

    def predict(self, X):
        '''
        Predict the cluster labels for the input data.

        Parameters:
            X (torch.Tensor): Input data of shape (n_samples, n_features).

        Returns:
            torch.Tensor: Predicted cluster labels of shape (n_samples,).
        '''
        responsibilities = self._e_step(X)
        labels = torch.argmax(responsibilities, dim=0)
        return labels

    def get_cluster_means(self):
        '''
        Get the cluster means.

        Returns:
            torch.Tensor: Cluster means of shape (n_components, n_features).
        '''
        return self.means

    def get_cluster_covariances(self):
        '''
        Get the cluster covariances.

        Returns:
            torch.Tensor: Cluster covariances of shape (n_components,
n_features, n_features).
        '''
        return self.covariances
```

Output:

```
PS C:\Users\Praka\OneDrive\Documents\5thSem\MI\GMM(Student)> python Test.py --ID EC_F_PES2UG21CS315_Lab6
Test Case 1 for GMM fitting and prediction PASSED
Test Case 2 for getting cluster means PASSED
Test Case 3 for getting cluster covariances PASSED
Test Case 4 for GMM prediction PASSED
PS C:\Users\Praka\OneDrive\Documents\5thSem\MI\GMM(Student)>
```