Centro de Investigación en Matemáticas, A.C.

Reconocimiento de Patrones

Jose Miguel Saavedra Aguilar

Examen 2. Ejercicio 2

Para el segundo ejercicio del examen, se implementará un algoritmo Expectation-Maximization (EM) para ajustar una mezcla de k distribuciones gaussianas a un conjunto de datos.

Para este fin, nos basamos en el libro de Bishop, "Pattern Recognition and Machine Learning" del 2009, pues en clase solo estudiamos el algoritmo para variables reales.

In [ ]:
```
# Import necessary libraries for image processing, numerical operations, and statis
from PIL import Image  # For image loading and manipulation
import numpy as np     # For numerical operations and array handling
from scipy.stats import multivariate_normal  # For multivariate Gaussian distributi
```

Segun Bishop, para maximizar la verosimilitud de los datos, se deriva la función de log-verosimilitud con respecto a las medias e igualamos a cero, obteniendo la siguiente expresión:

$$\gamma(z_{nk}) = \sum_j \pi_j \mathcal{N}(x_n|\mu_j, \Sigma_j)$$

$$0 = -\sum_{n=1}^{N} \frac{\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)}{\gamma(z_{nk})} \Sigma_k (x_n - \mu_k).$$

A $\gamma(z_{nk})$ se le conoce como la responsabilidad, o la probabilidad posterior de que el punto x_n pertenezca al componente k de la mezcla, mientras que π_k es el peso, o probabilidad a priori de la mezcla del componente k.

De esta forma, suponiendo que Σ_k es no singular, la expresión anterior se puede reescribir como:

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) x_n,$$

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}),$$

podemos interpretar N_k como el número de puntos asignados al componente k de la mezcla.\

De forma similar, para la matriz de covarianzas, se sigue un camino análogo derivando con respecto a Σ_k, obteniendo:

$$\Sigma_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(x_n - \mu_k)(x_n - \mu_k)^\top.$$

Finalmente, para los pesos de la mezcla, por medio del lagrangiano, se obtiene:

$$\pi_k = \frac{N_k}{N}.$$

En el algoritmo EM, se alternan los pasos de Expectation (E) y Maximization (M) hasta que la convergencia sea alcanzada. En el paso E, se calcula la responsabilidad $\gamma(z_{nk})$ para cada punto x_n y componente k, mientras que en el paso M, se actualizan los parámetros del modelo (medias, covarianzas y pesos) utilizando las responsabilidades calculadas.

A continuación, se presenta la implementación del algoritmo EM para ajustar una mezcla de k distribuciones gaussianas a un conjunto de datos:

In [ ]:

```
class GaussianMixtureModel:
    """
    A Gaussian Mixture Model (GMM) implementation using the Expectation-Maximizatio
    """
    def __init__(self, n_components, max_iters=100, tol=1e-4, random_state=None):
        """
        Initialize the Gaussian Mixture Model.
        Parameters:
        - n_components: Number of Gaussian components in the mixture.
        Optional parameters:
        - max_iters: Maximum number of iterations for the EM algorithm.
        - tol: Convergence tolerance for log-likelihood.
        - random_state: Seed for random number generator for reproducibility.
        """
        # Number of Gaussian components
        self.k = n_components
        # Maximum number of EM iterations
        self.max_iters = max_iters
        # Convergence tolerance for log-likelihood
        self.tol = tol
        # Random seed for reproducibility
        self.random_state = random_state
```

```
    def _initialize_parameters(self, X):
        """
        Initialize the parameters of the GMM.
        Randomly selects initial means from the data points and initializes covaria
        """
        # Number of samples and features
        n_samples, n_features = X.shape
        if self.random_state is not None:
            np.random.seed(self.random_state)

        # Randomly choose initial means from data points
        indices = np.random.choice(n_samples, self.k, replace=False)
        self.means_ = X[indices]
        # Initialize covariances to the sample covariance, regularized
        self.covariances_ = np.array([np.cov(X.T) + 1e-6 * np.eye(n_features) for _
        # Initialize equal priors
        self.prior_ = np.full(self.k, 1 / self.k)

    def _e_step(self, X):
        """
        Perform the Expectation step of the EM algorithm.
        Computes the posterior probabilities (responsibilities) for each component

        Args:
            X (np.ndarray): Input data of shape (n_samples, n_features).
        """
        n_samples = X.shape[0]
        # Posterior probabilities (responsibilities)
        self.posterior_ = np.zeros((n_samples, self.k))

        for i in range(self.k):
            # Compute probability of each point under component i
            rv = multivariate_normal(mean=self.means_[i], cov=self.covariances_[i])
            self.posterior_[:, i] = self.prior_[i] * rv.pdf(X)

        # Normalize responsibilities across components
        self.posterior_ /= self.posterior_.sum(axis=1, keepdims=True)

    def _m_step(self, X):
        """
        Perform the Maximization step of the EM algorithm.
        Updates the parameters (means, covariances, and priors) based on the curren
        Args:
            X (np.ndarray): Input data of shape (n_samples, n_features).
        """
        n_samples, n_features = X.shape
        # Effective number of points assigned to each component
        Nk = self.posterior_.sum(axis=0)
        # Update priors
        self.prior_ = Nk / n_samples
        # Update means
        self.means_ = np.dot(self.posterior_.T, X) / Nk[:, np.newaxis]

        self.covariances_ = []
        for i in range(self.k):
            # Compute weighted covariance for each component
```

```python
            diff = X - self.means_[i]
            weighted_diff = self.posterior_[:, i][:, np.newaxis] * diff
            cov = weighted_diff.T @ diff / Nk[i]
            cov += 1e-6 * np.eye(n_features)  # regularization for numerical stabil
            self.covariances_.append(cov)
        self.covariances_ = np.array(self.covariances_)

    def _gaussian_log_likelihood(self, X):
        """
        Compute the log-likelihood of the data given the current model parameters.
        Args:
            X (np.ndarray): Input data of shape (n_samples, n_features).
        """
        n_samples = X.shape[0]
        likelihood = np.zeros((n_samples, self.k))
        for i in range(self.k):
            # Compute likelihood of each point under each component
            rv = multivariate_normal(mean=self.means_[i], cov=self.covariances_[i])
            likelihood[:, i] = self.prior_[i] * rv.pdf(X)
        # Return total log-likelihood
        return np.sum(np.log(likelihood.sum(axis=1)))

    def fit(self, X):
        """
        Fit the Gaussian Mixture Model to the data using the EM algorithm.

        Args:
            X (np.ndarray): Input data of shape (n_samples, n_features).
        """
        # Initialize parameters
        self._initialize_parameters(X)
        self.log_likelihood_ = -np.inf

        for _ in range(self.max_iters):
            # E-step: update responsibilities
            self._e_step(X)
            # M-step: update parameters
            self._m_step(X)
            # Compute log-likelihood
            new_log_likelihood = self._gaussian_log_likelihood(X)
            # Check for convergence
            if abs(new_log_likelihood - self.log_likelihood_) < self.tol:
                break
            self.log_likelihood_ = new_log_likelihood

    def predict_proba(self, X):
        """
        Predict the posterior probabilities of each component for the given data.
        Args:
            X (np.ndarray): Input data of shape (n_samples, n_features).
            Returns:
            np.ndarray: Posterior probabilities of shape (n_samples, n_components).
        """
        n_samples = X.shape[0]
        probs = np.zeros((n_samples, self.k))
        for i in range(self.k):
```

```
            # Compute probability of each point under each component
            rv = multivariate_normal(mean=self.means_[i], cov=self.covariances_[i])
            probs[:, i] = self.prior_[i] * rv.pdf(X)
        # Normalize to get probabilities
        probs /= probs.sum(axis=1, keepdims=True)
        return probs

    def predict(self, X):
        """
        Predict the component labels for the given data based on the highest poster
        Args:
            X (np.ndarray): Input data of shape (n_samples, n_features).
        Returns:
            np.ndarray: Predicted component labels of shape (n_samples,).
        """
        # Assign each point to the component with highest probability
        return np.argmax(self.predict_proba(X), axis=1)

    def fit_predict(self, X):
        """
        Fit the model to the data and predict the component labels.
        Args:
            X (np.ndarray): Input data of shape (n_samples, n_features).
        Returns:
            np.ndarray: Predicted component labels of shape (n_samples,).
        """
        self.fit(X)
        return self.predict(X)

    def get_parameters(self):
        """
        Get the parameters of the GMM.
        Returns:
            dict: Dictionary containing means, covariances, and priors.
        """
        return {
            'means': self.means_,
            'covariances': self.covariances_,
            'priors': self.prior_
        }
```

Definimos funciones auxiliares para convertir la imagen a datos, y la función inversa:

In [ ]:
```
# Convert an image to a feature array for clustering
# Each pixel is represented as a 3-dimensional vector (RGB)
def image_to_features(image_path):
    img = Image.open(image_path).convert('RGB')  # Load image and ensure RGB format
    img_np = np.array(img)  # Convert image to numpy array
    n, m, c = img_np.shape  # Get image dimensions
    features = img_np.reshape(-1, 3)  # Flatten image to (num_pixels, 3)
    return features, (n, m)

# Convert clustered features back to image format
def features_to_image(features, shape):
```

```
    n, m = shape  # Unpack original image shape
    return features.reshape(n, m, 3).astype(np.uint8)  # Reshape and convert to uin
```

Para los 4 valores de k, (2, 3, 5 y 10), se ejecuta el algoritmo de E-M, o de mezcla de Gaussianas (GMM) para colorear la imagen:

In [ ]:
```
# List of different numbers of gaussians to try
K = [2, 3, 5, 10]

for k in K:
    # Load and transform the image into features (pixels as RGB vectors)
    features, shape = image_to_features("Figures/foto.jpg")

    # Fit the Gaussian Mixture Model to the pixel features
    em = GaussianMixtureModel(n_components=k, max_iters=30, tol=1e-4, random_state=
    labels = em.fit_predict(features)  # Cluster assignment for each pixel

    parameters = em.get_parameters()  # Get learned means

    output = np.zeros_like(features)  # Prepare array for reconstructed image

    # Assign each pixel the mean color of its cluster
    for i in range(shape[0] * shape[1]):
        output[i] = parameters['means'][labels[i]]

    # Inverse transform: reshape features back to image and save result
    reconstructed_img = features_to_image(output, shape)
    Image.fromarray(reconstructed_img).save(f"gmm_{k}.png")
```

Las imágenes resultantes se encuentran guardadas con el nombre 'gmm_x.png', donde x es el número de gaussianas utilizadas.