Practical no: 6

Aim: Probabilistic Models

a. Implement Bayesian Linear Regression to explore prior and posterior distribution.

Description:

Bayesian Linear Regression is a **probabilistic approach** to linear regression that models **uncertainty in the weights** of the linear model. Unlike classical regression, which provides a single best-fit line, Bayesian regression treats the model parameters as **random variables** and provides a **distribution over possible models**.

Bayesian Linear Regression is a probabilistic approach to linear regression that treats the model parameters (such as the slope and intercept) as random variables with probability distributions. Unlike traditional linear regression, which provides a single best-fit line, Bayesian Linear Regression provides a full distribution over possible models. This allows it to naturally incorporate uncertainty and make probabilistic predictions.

Before observing any data, we define a prior distribution that expresses our beliefs about the likely values of the model parameters. Once we observe data, we update these beliefs using Bayes' Theorem, resulting in a posterior distribution. This posterior combines the prior knowledge and the information from the observed data.

After computing the posterior, we can make predictions for new inputs by integrating over the uncertainty in the model parameters. This results in a predictive distribution that gives not only the expected value of the prediction but also a measure of the uncertainty around it.

Bayesian Linear Regression is especially useful when working with small datasets, when model uncertainty is important, or when incorporating prior knowledge into the learning process. It also helps avoid overfitting by accounting for uncertainty rather than relying on a single set of parameter values.

In practice, Bayesian Linear Regression provides:

- A distribution over model parameters (posterior)
- A probabilistic prediction for new inputs (predictive distribution)
- Visual insights into uncertainty through prior/posterior samples and prediction intervals

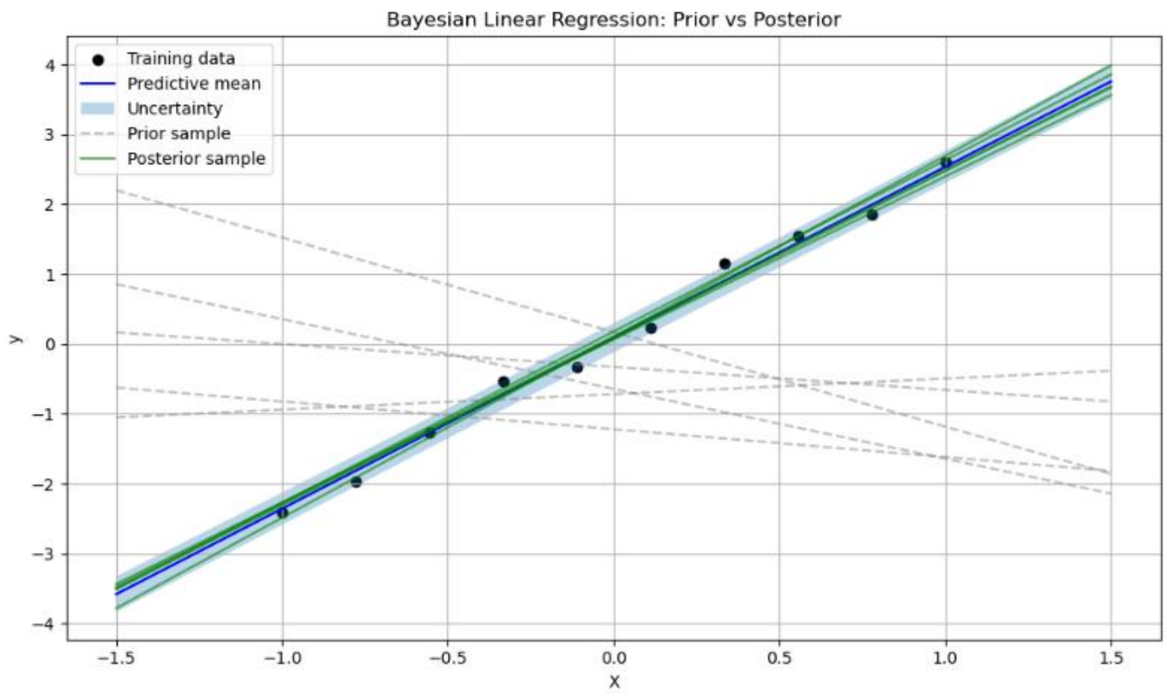
Code:

```
import numpy as np

import matplotlib.pyplot as plt


# Set random seed for reproducibility

np.random.seed(42)


# Generate synthetic data

def generate_data(n_samples=10):

    X = np.linspace(-1, 1, n_samples)

    y = 2.5 * X + np.random.normal(0, 0.2, size=X.shape)

    return X.reshape(-1, 1), y


# Add bias term

def add_bias(X):

    return np.hstack([np.ones((X.shape[0], 1)), X])


# Bayesian Linear Regression

class BayesianLinearRegression:

    def __init__(self, alpha=2.0, beta=25.0):

        self.alpha = alpha  # Prior precision (inverse variance)

        self.beta = beta    # Noise precision (1 / sigma^2)


    def fit(self, X, y):

        X_bias = add_bias(X)

        N, D = X_bias.shape

        I = np.eye(D)
        # Posterior covariance
        S_inv = self.alpha * I + self.beta * X_bias.T @ X_bias
        self.S = np.linalg.inv(S_inv)

        # Posterior mean
        self.m = self.beta * self.S @ X_bias.T @ y

    def predict(self, X_test, return_std=False):
        X_test_bias = add_bias(X_test)
        y_mean = X_test_bias @ self.m
        y_var = 1 / self.beta + np.sum(X_test_bias @ self.S * X_test_bias, axis=1)
        if return_std:
            return y_mean, np.sqrt(y_var)
        return y_mean

    def sample_weights(self, n_samples=5):
        return np.random.multivariate_normal(self.m, self.S, size=n_samples)

# Generate data
X_train, y_train = generate_data(10)

# Fit model
model = BayesianLinearRegression(alpha=2.0, beta=25.0)
model.fit(X_train, y_train)

# Test points for prediction
X_test = np.linspace(-1.5, 1.5, 100).reshape(-1, 1)
```

```
# Posterior predictive mean and uncertainty

y_pred, y_std = model.predict(X_test, return_std=True)


# Plot

plt.figure(figsize=(10, 6))


# Plot training data

plt.scatter(X_train, y_train, color='black', label='Training data')


# Plot predictive mean

plt.plot(X_test, y_pred, color='blue', label='Predictive mean')


# Uncertainty region

plt.fill_between(X_test.ravel(), y_pred - y_std, y_pred + y_std, alpha=0.3,
label='Uncertainty')


# Prior weight samples

prior_samples = np.random.multivariate_normal(mean=np.zeros(2),
cov=(1/model.alpha)*np.eye(2), size=5)

for w in prior_samples:

   y_sample = add_bias(X_test) @ w

   plt.plot(X_test, y_sample, '--', color='gray', alpha=0.5, label='Prior sample' if 'Prior
sample' not in plt.gca().get_legend_handles_labels()[1] else "")


# Posterior weight samples

posterior_samples = model.sample_weights(5)

for w in posterior_samples:

   y_sample = add_bias(X_test) @ w
```

```
    plt.plot(X_test, y_sample, '-', color='green', alpha=0.6, label='Posterior sample' if 'Posterior sample' not in plt.gca().get_legend_handles_labels()[1] else "")


plt.title('Bayesian Linear Regression: Prior vs Posterior')

plt.xlabel('X')

plt.ylabel('y')

plt.legend()

plt.grid(True)

plt.tight_layout()

plt.show()
```

Output:



Learnings:

- Understand priors and posteriors
- Visualize model uncertainty
- Predict with confidence intervals
- Compare Bayesian vs. traditional regression
- Apply probabilistic thinking to real data

b. Implement Gaussian Mixture Models for density estimation and unsupervised clustering.

Description:

Gaussian Mixture Models (GMMs) are probabilistic models used for unsupervised clustering and density estimation. A GMM assumes that the data is generated from a mixture of several Gaussian distributions, each representing a different cluster. Each component (cluster) has its own mean and covariance, and the overall model learns both the parameters of these Gaussians and their mixing proportions.

GMMs use the Expectation-Maximization (EM) algorithm to iteratively assign data points to clusters (soft assignments based on probabilities) and update the model parameters. Unlike methods like K-Means, GMMs allow for elliptical clusters, varying sizes, and overlapping regions, making them more flexible and suitable for real-world data.

In addition to clustering, GMMs can estimate the underlying probability density of the data, which is useful for anomaly detection, generative modeling, and pattern recognition.

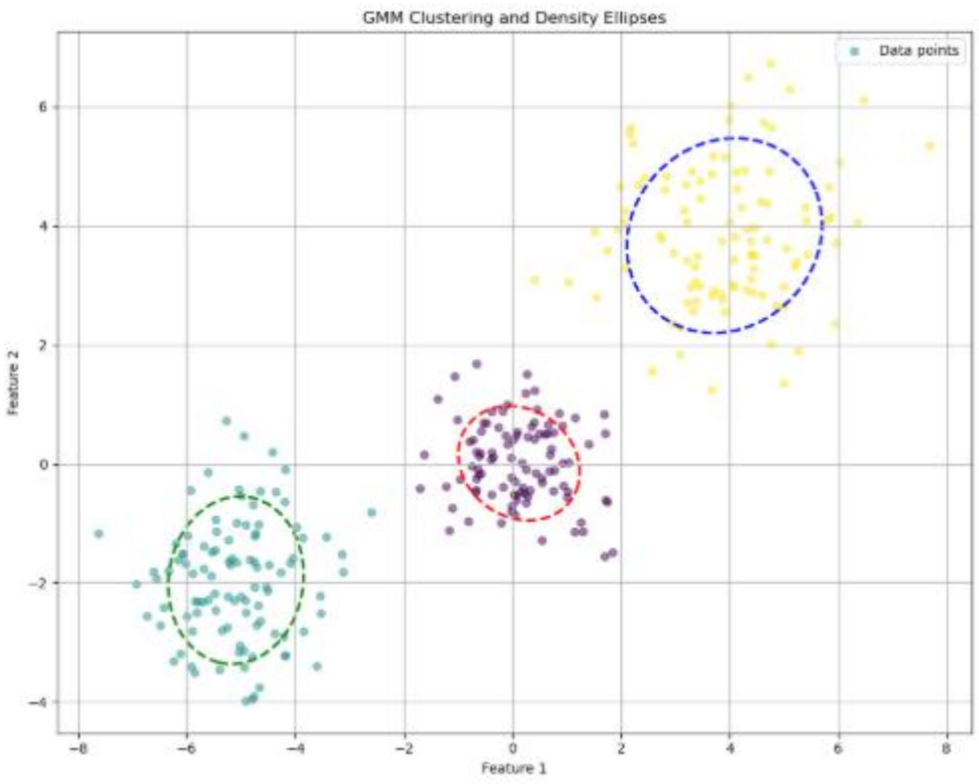
Code:

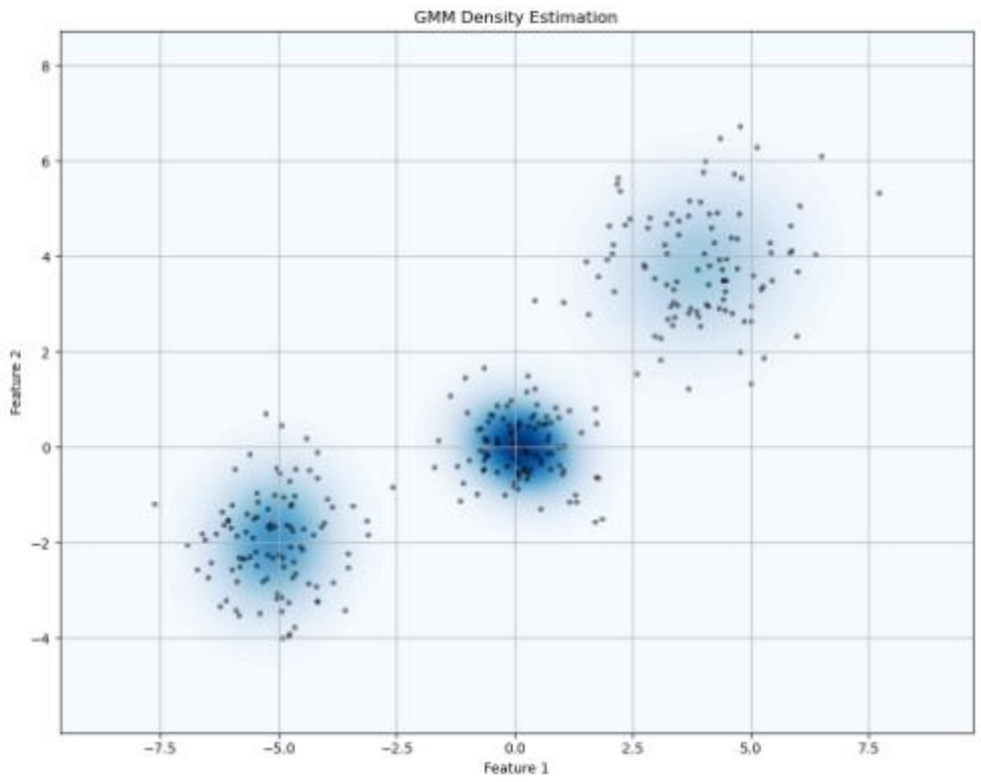
```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.mixture import GaussianMixture

from sklearn.datasets import make_blobs

from matplotlib.patches import Ellipse


# Step 1: Generate synthetic 2D data

def generate_data(n_samples=300):

    centers = [(-5, -2), (0, 0), (4, 4)]

    cluster_std = [1.0, 0.8, 1.2]

    X, y_true = make_blobs(n_samples=n_samples, centers=centers,
cluster_std=cluster_std, random_state=42)

    return X, y_true


# Step 2: Fit GMM model
```

```
def fit_gmm(X, n_components=3):
    gmm = GaussianMixture(n_components=n_components, covariance_type='full',
random_state=42)
    gmm.fit(X)
    return gmm


# Step 3: Plot GMM clusters and ellipses
def plot_gmm(gmm, X, y_pred):
    plt.figure(figsize=(10, 8))
    ax = plt.gca()


    plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis', s=30, marker='o', alpha=0.5,
label='Data points')


    colors = ['red', 'green', 'blue', 'purple', 'orange']
    for i, (mean, covar) in enumerate(zip(gmm.means_, gmm.covariances_)):
        # Eigen-decomposition for ellipse
        v, w = np.linalg.eigh(covar)
        v = 2.0 * np.sqrt(2.0) * np.sqrt(v)
        u = w[0] / np.linalg.norm(w[0])


        angle = np.arctan2(u[1], u[0])
        angle = np.degrees(angle)


        ell = Ellipse(mean, v[0], v[1], 180.0 + angle, edgecolor=colors[i % len(colors)],
                  lw=2, facecolor='none', linestyle='--')
        ax.add_patch(ell)


    plt.title("GMM Clustering and Density Ellipses")
    plt.xlabel("Feature 1")

    plt.ylabel("Feature 2")

    plt.grid(True)

    plt.legend()

    plt.tight_layout()

    plt.show()


# Step 4: Visualize density estimation

def plot_density(gmm, X):

    x = np.linspace(np.min(X[:, 0]) - 2, np.max(X[:, 0]) + 2, 300)

    y = np.linspace(np.min(X[:, 1]) - 2, np.max(X[:, 1]) + 2, 300)

    Xgrid, Ygrid = np.meshgrid(x, y)

    XX = np.array([Xgrid.ravel(), Ygrid.ravel()]).T


    Z = -gmm.score_samples(XX)

    Z = Z.reshape(Xgrid.shape)


    plt.figure(figsize=(10, 8))

    plt.contourf(Xgrid, Ygrid, np.exp(-Z), levels=50, cmap='Blues')

    plt.scatter(X[:, 0], X[:, 1], c='black', s=10, alpha=0.4)

    plt.title("GMM Density Estimation")

    plt.xlabel("Feature 1")

    plt.ylabel("Feature 2")

    plt.grid(True)

    plt.tight_layout()

    plt.show()


# Run the full pipeline
```

```
X, y_true = generate_data()

gmm = fit_gmm(X, n_components=3)

y_pred = gmm.predict(X)


# Visualizations

plot_gmm(gmm, X, y_pred)

plot_density(gmm, X)
```

Output:





Learnings:

Understand how GMMs model data using multiple Gaussian distributions

Learn the Expectation-Maximization (EM) algorithm for parameter estimation

Perform unsupervised clustering with soft assignments

Visualize clusters and probability density contours

Compare GMMs with K-Means and other clustering methods

Apply GMMs for tasks like density estimation and anomaly detection