Project 1: Linear Regression (From Scratch) – Code Explanation

1. **Objective**
   The goal of this code is to predict house prices using Linear Regression, implemented from scratch without using machine learning libraries. It uses basic concept like gradient descent matrix operation and mean squared error for learning
2. **Important Libraries**
   Import numPy as np
   - Numpy is used for mathematical operations (arrays , matrix multiplication, mean ,etc).

   Import pandas as pd
   - Pandas is used to load and handle the dataset easily.

3. **Loading the Dataset**
   data = pd.read_csv('HousingData.csv')
   X = data.drop('MEDV',axis=1).
   Values y = data['MEDV'].values.reshape(-1,1)

   - The dataset contain information about house ,
     X – independent feature (area , rooms, etc.)
   - Y – Target (house price column MEDV).
4. **Normalizing the data**
   X = (x- x.mean(axis=0))/x.std(axis=0)
   - Each feature is standardized to have mean = 0 and standard deviation = 1. This ensure all feature contribute equally and helps the model train faster.
5. Splitting the Data
   A custom function splits the dataset into 80% training and 20% testing.
   Def train_test_split(X,y,test_size=0.2):
   n= len(X) idx = np.arange(n)
   np.random.shuffle(idx)
   split = int(n*(1-test_size))
   return X[idx[:split]], X[idx[split:]], y[idx[:split]],y[idx[split:]]
6. **Initializing Model Parameters**
   n_features = X_train.shape[1] W = np.zeros((n_features,1)) b = 0
   ir = 0.01
   epochs = 500
   - W – weight (initially zeros)
   - b – bias
   - epochs - number of iterations

7. **Training the Model (Gradient Descent)**

```
for i in range(epochs):
y_pred = X_train @W + b
Error = y_pred – y_train
loss = np_mean(error**2)
dW = (2/len(X_train))*(X_train.T @ error)
db = (2/len(X_train)) * np.sum(error)
W -= lr*dW
b-= lr*db
```

- Predict – compute y_pred
- Compute error & loss
- Calculate gradients
- Update weight
- Print loss every 100 epochs

8. **Model Evalution**

```
Y_pred_test = X_test @ W  + b
mse = np.mean((y_test – y_pred_test)**2)
r2 = 1 – np.sum((y_test – y_pred_test)**2)/ np.sum((y_test- y_test.mean())**2)
```

- MSE – average error
- R^2 = how well model explains data

9. **Output Example**

Epoch 0, Loss: 532.74
Epoch 100, Loss: 50.43
Epoch 200, Loss: 28.12
Test MSE: 25.84
Test R2: 0.86
Loss decreases - model
learning R^2 near 1 - good fit

10. **Summary**

Load Data - Normalize - Split - Initialize - Train - Evaluate.
I implemented Linear Regression from scratch using gradient descent, manually computing predictions, errors, and gradients to update weights and bias. Finally, evaluated model with MSE and R^2 metrics.

Project 2:

1. **Loading the Dataset**

   iris = pd.read_csv('iris_dataset.csv')

   X = iris.iloc[:, :-1].values

   y = iris.iloc[:,1].values

   •The dataset contains flower measurements and species names.

   • X - features (petal length, sepal width, etc.)

   •y - labels (flower species).

2. **Encoding and Normalizing the Data**.

   •Convert string labels into numeric values.

   •Select only two classes for binary classification.

   •Normalize all features using:

   X = (X - X.mean(axis=0)) / X.std(axis=0)

3. **Train-Test Split.**

   A custom train-test split function divides the dataset into 80% training and 20% testing for performance evaluation.

4. **Initializing Parameters**

   W = np.zeros((X_train.shape[1], 1))

   b = 0

   lr = 0.1

   epochs = 500

   •W: weights for each feature

   •b: bias term

   •lr: learning rate

   •epochs: number of iterations

5. **Sigmoid Function**

   def sigmoid(z): return 1 / (1 + np.exp(-z))

   The sigmoid function converts linear output into probabilities between 0 and 1.

6. **Training using Gradient Descent.**

   for i in range(epochs):

   z = X_train @ W + b

   y_pred = sigmoid(z)

   loss = -np.mean(y_train*np.log(y_pred+1e-8) + (1-y_train)*np.log(1-y_pred+1e-8))

   dW = (1/len(X_train)) * (X_train.T @ (y_pred - y_train))

   db = (1/len(X_train)) * np.sum(y_pred - y_train)

   W -= lr * dW

   b -= lr * db

   • Compute predictions, loss, gradients, and update parameters.

7. **Model Evaluation**
   y_pred_test = sigmoid(X_test @ W + b) >= 0.5
   acc = np.mean(y_pred_test == y_test)
   •Model predicts probabilities and classifies using threshold 0.5.
   •Accuracy is the fraction of correct predictions.
8. **Output Example**
   Epoch 0, Loss: 0.6931
   Epoch 100, Loss: 0.2789 Test Accuracy: 0.92
   Loss decreases over time - model learning correctly.
9. **Summary**
   I implemented logistic regression from scratch using the sigmoid activation and gradient descent optimization. The model minimizes binary cross-entropy loss and achieves good accuracy on the Iris dataset.

Project 3 : K-Means Clustering

1. **Objective**
   In this project, my goal was to group mall customers into different categories based on their annual income and spending score.
2. **Loading the Dataset.**
   mall = pd.read_csv("Mall_Customer.csv")
   X = mall[['Annual Income (k$)', 'Spending Score (1-1)']].values

   The dataset contains information about different customers.
   I selected two features Annual Income (k$) and Spending Score (1–100) because these two are enough to understand the spending patterns.
3. **K-Means Concept**
   K-Means is an unsupervised learning algorithm, meaning it doesn't need labels or predefined outputs.
   It simply groups similar data points together.
   The steps are:
   Choose the number of clusters (here k = 3).
   Randomly select k points as initial centroids.
   Assign each data point to the nearest centroid.
   Move centroids to the average position of all points in their cluster.
   Repeat until centroids stop moving.

Support Vector Machine (SVM)

1. **Objective**
   The objective of this code is to implement a Support Vector Machine (SVM) from scratch for binary classification. SVM aims to find the best hyperplane that separates two classes with the maximum margin.
2. **Concept Overview**
   SVM tries to find weights (W) and bias (b) that maximize the margin between two classes, while minimizing classification errors. The optimization is based on the hinge loss function
3. **Importing Libraries**
   import numpy as np
   Only numpy is used for mathematical operations and array manipulation.
4. **Model Initialization**
   W= np.zeros(X_train.shape[1])
   B = 0.0
5. **Training Function (Fit)**

```
# Reuse iris loaded above
X = iris.iloc[:, :-1].values
y = iris.iloc[:, -1].values
labels, y = np.unique(y, return_inverse=True)
mask = y < 2
X, y = X[mask], y[mask]
y = np.where(y == 0, -1, 1)  # convert 0 -> -1, 1 -> 1

# normalize (simple, same as before)
X = (X - X.mean(axis=0)) / X.std(axis=0)

# train-test split (reuse function)
X_train, X_test, y_train, y_test = train_test_split(X, y.reshape(-1, 1))
y_train = y_train.flatten()
y_test = y_test.flatten()

# initialize weights
w = np.zeros(X_train.shape[1])
b = 0.0
```

6. **Prediction Function**

   def svm_predict(X):

     scores = X @ w + b

     return np.where(scores >= 0, 1, -1)

   Calculates class label (+1 or -1) depending on which side of the hyperplane a sample lies.

7. **Loss Function Explanation**

   The hinge loss used by SVM is:

   Loss = 1/N * Summation (max(0,1 – y (w*x - b))) + lambda||W||^2

   •The first term penalizes points inside the margin or on the wrong side.

   •The second term (lambda||W||^2) helps control overfitting.

8. **Hyperparameters**

   •lr - learning rate

   •lambda_param - regularization term

   •n_iters - number of iterations

   These control the model's learning speed and generalization strength.

9. **Evaluation**

   Predictions are compared with actual labels to calculate accuracy:

   accuracy = np.mean(y_pred == y_test)

10. **Summary**

    The SVM implementation uses hinge loss and gradient descent to find the optimal separating hyperplane. It works effectively for binary classification problems, providing a strong margin-based decision boundary.