1. Polynomial Regression on Data1.csv

Code:

```
python
Copy code
import numpy as np
import pandas as pd

# Load the data
data = pd.read_csv("Data1.csv")  # Replace with the correct file path
X = data['X'].values
Y = data['Y'].values

# Fit a second-degree polynomial (quadratic regression)
X_poly = np.column_stack((X ** 2, X, np.ones(len(X))))
coefficients = np.linalg.lstsq(X_poly, Y, rcond=None)[0]

print("Coefficients: ", coefficients)
# Coefficients are for X^2, X, and intercept
```

---

2. Logistic Regression on Pima Indian Diabetes Dataset

Code:

```
python
Copy code
# Load dataset
data = pd.read_csv("pima_diabetes.csv")  # Replace with the correct file path
X = data.iloc[:, :-1].values  # Independent variables
Y = data.iloc[:, -1].values   # Target (0 or 1)

# Initialize parameters
n_features = X.shape[1]
weights = np.zeros(n_features)
bias = 0
learning_rate = 0.01
epochs = 1000

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

# Logistic regression training
for epoch in range(epochs):
    linear_model = np.dot(X, weights) + bias
    predictions = sigmoid(linear_model)
    errors = Y - predictions

    # Gradient descent
    weights += learning_rate * np.dot(X.T, errors) / len(Y)
    bias += learning_rate * np.sum(errors) / len(Y)

print("Trained Weights: ", weights)
print("Trained Bias: ", bias)
```

---

3. Logistic Regression Classifier for Digits

Assumptions:

- The dataset (e.g., MNIST) is preprocessed and available as `train_images.csv` and `train_labels.csv` for training, and `test_images.csv` and `test_labels.csv` for testing.
- Images are flattened into vectors (each row corresponds to one image).
- Labels are integers representing the digit (0–9).

Code:

```
python
Copy code
import numpy as np

# Load the dataset (replace with correct file paths)
train_images = np.loadtxt("train_images.csv", delimiter=",")
train_labels = np.loadtxt("train_labels.csv", delimiter=",")
test_images = np.loadtxt("test_images.csv", delimiter=",")
test_labels = np.loadtxt("test_labels.csv", delimiter=",")

# Normalize the images
train_images = train_images / 255.0
test_images = test_images / 255.0

# One-hot encode the labels for multi-class logistic regression
def one_hot_encode(labels, num_classes=10):
    encoded = np.zeros((labels.size, num_classes))
    encoded[np.arange(labels.size), labels.astype(int)] = 1
    return encoded

train_labels_encoded = one_hot_encode(train_labels)
num_classes = train_labels_encoded.shape[1]

# Initialize parameters
num_features = train_images.shape[1]
weights = np.zeros((num_features, num_classes))
bias = np.zeros(num_classes)
learning_rate = 0.01
epochs = 1000

# Softmax function
def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / exp_z.sum(axis=1, keepdims=True)

# Training the model
for epoch in range(epochs):
    # Compute predictions
    linear_model = np.dot(train_images, weights) + bias
    predictions = softmax(linear_model)

    # Compute the gradient
    errors = predictions - train_labels_encoded
    dW = np.dot(train_images.T, errors) / train_images.shape[0]
    dB = np.sum(errors, axis=0) / train_images.shape[0]

    # Update weights and bias
    weights -= learning_rate * dW
    bias -= learning_rate * dB
```

```
    if epoch % 100 == 0:
        loss = -np.mean(np.sum(train_labels_encoded * np.log(predictions +
1e-15), axis=1))
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Testing the model
def predict(images, weights, bias):
    linear_model = np.dot(images, weights) + bias
    probabilities = softmax(linear_model)
    return np.argmax(probabilities, axis=1)

# Predict on test data
predicted_labels = predict(test_images, weights, bias)

# Calculate accuracy
accuracy = np.mean(predicted_labels == test_labels) * 100
print(f"Test Accuracy: {accuracy:.2f}%")
```

---

4. K-means Clustering

Code:

```
python
Copy code
import numpy as np

# Data
X = np.array([4, 5, 10, 4, 3, 11, 14, 6, 10, 12])
Y = np.array([21, 19, 24, 17, 16, 25, 24, 22, 21, 21])
data = np.column_stack((X, Y))

# Initialize centroids
k = 2  # Number of clusters
centroids = data[:k]

for _ in range(10):  # Iterate to refine centroids
    clusters = [[] for _ in range(k)]

    # Assign data to the nearest centroid
    for point in data:
        distances = np.linalg.norm(point - centroids, axis=1)
        cluster_idx = np.argmin(distances)
        clusters[cluster_idx].append(point)

    # Recalculate centroids
    centroids = [np.mean(cluster, axis=0) for cluster in clusters]

print("Centroids:", centroids)
```

---

5. Hierarchical Clustering

Code:

```
python
Copy code
import scipy.cluster.hierarchy as sch
```

```
import matplotlib.pyplot as plt
import pandas as pd

# Load dataset
data = pd.read_csv("Mall_Customers_data.csv")  # Replace with the correct file
path
X = data.iloc[:, [3, 4]].values  # Columns to cluster on

# Create dendrogram
dendrogram = sch.dendrogram(sch.linkage(X, method='ward'))
plt.title("Dendrogram")
plt.xlabel("Customers")
plt.ylabel("Euclidean Distances")
plt.show()
```

---

6. Decision Tree with Information Gain

Code:

```
python
Copy code
# Manually calculate entropy and information gain
import numpy as np

# Dataset
weather = ["sunny", "sunny", "overcast", "rain", "rain", "rain", "overcast",
"sunny", "sunny", "rain"]
play = ["no", "no", "yes", "yes", "yes", "no", "yes", "no", "yes", "yes"]

# Calculate entropy
def entropy(values):
    unique, counts = np.unique(values, return_counts=True)
    probs = counts / len(values)
    return -np.sum(probs * np.log2(probs))

# Information gain
def info_gain(parent, children):
    parent_entropy = entropy(parent)
    weighted_entropy = sum((len(child) / len(parent)) * entropy(child) for child
in children)
    return parent_entropy - weighted_entropy

# Example calculation for 'sunny'
parent_entropy = entropy(play)
sunny_idx = [i for i, val in enumerate(weather) if val == "sunny"]
not_sunny_idx = [i for i in range(len(weather)) if i not in sunny_idx]
sunny_play = [play[i] for i in sunny_idx]
not_sunny_play = [play[i] for i in not_sunny_idx]

ig_sunny = info_gain(play, [sunny_play, not_sunny_play])
print(f"Information Gain for sunny: {ig_sunny}")
```

---

7. SVM for Optimal Hyperplane

Code:

```
python
Copy code
```

```
import numpy as np

# Dataset
positive = np.array([[3, 1], [3, -1], [6, 1], [6, -1]])
negative = np.array([[1, 0], [0, 1], [0, -1], [-1, 0]])

# Combine data
X = np.vstack((positive, negative))
Y = np.array([1] * len(positive) + [-1] * len(negative))  # Labels

# Initialize weights and bias
weights = np.zeros(2)
bias = 0
learning_rate = 0.01
epochs = 1000

# Training loop
for _ in range(epochs):
    for i in range(len(X)):
        condition = Y[i] * (np.dot(X[i], weights) + bias) >= 1
        if condition:
            weights -= learning_rate * 2 * 1 / epochs * weights
        else:
            weights -= learning_rate * (2 * 1 / epochs * weights - np.dot(X[i],
Y[i]))
            bias -= learning_rate * Y[i]

print(f"Optimal Hyperplane Weights: {weights}, Bias: {bias}")
```