Lab Instructions - Session 13

Classification, Regression, and Linear Models

Linear Regression from Scratch

Linear regression is one of the most basic and widely used supervised learning algorithms. In this part, you'll implement linear regression from scratch using the normal equation approach. You'll learn how to calculate the optimal weights directly by minimizing the sum of squared errors between predicted and actual values. This implementation focuses on a simple univariate case (one feature) to clearly demonstrate the mathematical principles.

LinearRegression.py

```python
import numpy as np
import matplotlib.pyplot as plt


np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)


split_idx = int(0.8 * len(X))
X_train, X_test = X[:split_idx], X[split_idx:]
y_train, y_test = y[:split_idx], y[split_idx:]

# TODO: Implement Linear Regression from scratch
class LinearRegression:
    def __init__(self):
        self.w = None
        self.b = None

    def fit(self, X, y):

        X_mean = np.mean(X)
        y_mean = np.mean(y)

        # Calculate the numerator and denominator for the slope
        numerator = np.sum((X - X_mean) * (y - y_mean))
        denominator = np.sum((X - X_mean)**2)
```

```
        # Calculate slope (w) and intercept (b)
        self.w = numerator / denominator
        self.b = y_mean - self.w * X_mean

    def predict(self, X):
        return self.w * X + self.b


model = LinearRegression()
model.fit(X_train, y_train)


y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)


def mean_squared_error(y_true, y_pred):
    return np.mean((y_true - y_pred)**2)

train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)


print(f"Model parameters: w = {model.w}, b = {model.b}")
print(f"Training MSE: {train_mse}")
print(f"Testing MSE: {test_mse}")


plt.figure(figsize=(10, 6))
plt.scatter(X_train, y_train, color='blue', label='Training data')
plt.scatter(X_test, y_test, color='green', label='Testing data')
plt.plot(X, model.predict(X), color='red', linewidth=2,
label='Prediction')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.title('Linear Regression')
plt.show()
```

- Explain how the slope and intercept are calculated in the `fit` method.

- What happens if you increase the noise in the data generation? Try modifying the random noise term.
- How would you extend this implementation to handle multiple features?

Logistic Regression for Binary Classification

Logistic regression extends linear regression to classification problems by applying a sigmoid function to transform the output into a probability value between 0 and 1. In this part, you'll implement logistic regression from scratch using gradient descent to optimize the parameters. You'll create a model that can classify data points into two classes and visualize the resulting decision boundary. This implementation demonstrates how to handle binary classification problems and introduces the concept of iterative optimization.

LogisticRegression.py

```
import numpy as np
import matplotlib.pyplot as plt


np.random.seed(42)
X = np.random.randn(100, 2)
y = (X[:, 0] + X[:, 1] > 0).astype(int).reshape(-1, 1)


split_idx = int(0.8 * len(X))
X_train, X_test = X[:split_idx], X[split_idx:]
y_train, y_test = y[:split_idx], y[split_idx:]

# TODO: Implement Logistic Regression from scratch
class LogisticRegression:
    def __init__(self, learning_rate=0.01, iterations=1000):
        self.learning_rate = learning_rate
        self.iterations = iterations
        self.weights = None
        self.bias = None

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

    def fit(self, X, y):

        n_samples, n_features = X.shape
        self.weights = np.zeros((n_features, 1))
        self.bias = 0
```

```
        # Gradient descent
        for _ in range(self.iterations):

            linear_model = np.dot(X, self.weights) + self.bias
            y_predicted = self.sigmoid(linear_model)


            dw = (1/n_samples) * np.dot(X.T, (y_predicted - y))
            db = (1/n_samples) * np.sum(y_predicted - y)


            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict_probability(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        return self.sigmoid(linear_model)

    def predict(self, X, threshold=0.5):
        return (self.predict_probability(X) >= threshold).astype(int)


model = LogisticRegression(learning_rate=0.1, iterations=1000)
model.fit(X_train, y_train)


y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# Calculate accuracy
def accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)

train_acc = accuracy(y_train, y_train_pred)
test_acc = accuracy(y_test, y_test_pred)

print(f"Training accuracy: {train_acc:.4f}")
print(f"Testing accuracy: {test_acc:.4f}")


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

```
plt.scatter(X_train[y_train[:, 0] == 0][:, 0], X_train[y_train[:, 0] ==
0][:, 1],
            color='blue', label='Class 0 (train)')
plt.scatter(X_train[y_train[:, 0] == 1][:, 0], X_train[y_train[:, 0] ==
1][:, 1],
            color='red', label='Class 1 (train)')


plt.scatter(X_test[y_test[:, 0] == 0][:, 0], X_test[y_test[:, 0] ==
0][:, 1],
            color='blue', marker='x', label='Class 0 (test)')
plt.scatter(X_test[y_test[:, 0] == 1][:, 0], X_test[y_test[:, 0] ==
1][:, 1],
            color='red', marker='x', label='Class 1 (test)')


x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))
Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contour(xx, yy, Z, colors='green', levels=[0.5], label='Decision
Boundary')

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.title('Logistic Regression Decision Boundary')
plt.show()
```

- Explain the sigmoid function and why it's used in logistic regression.
- How does gradient descent work in this implementation?
- Try changing the learning rate. What happens if it's too small or too large?

Task 1 - Polynomial Regression

Polynomial regression allows us to model nonlinear relationships by adding polynomial features to a linear model. In this task, you'll implement polynomial regression from scratch by creating polynomial features and then applying linear regression techniques. You'll explore how increasing the polynomial degree affects model performance and learn about the bias-variance tradeoff. This implementation demonstrates how to transform data to capture nonlinear patterns and introduces the concept of overfitting.

Task1.py

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate non-linear data
np.random.seed(42)
X = np.linspace(-3, 3, 100).reshape(-1, 1)
y = 0.5 * X**3 + X**2 - 2 * X + 2 + np.random.randn(100, 1) * 3

# Split data
split_idx = int(0.8 * len(X))
X_train, X_test = X[:split_idx], X[split_idx:]
y_train, y_test = y[:split_idx], y[split_idx:]

# TODO: Implement a function to create polynomial features
def create_poly_features(X, degree):
    """
    Create polynomial features up to specified degree.

    Parameters:
    X (array): Input features of shape (n_samples, 1)
    degree (int): Maximum polynomial degree

    Returns:
    array: Polynomial features of shape (n_samples, degree)
    """
    # Your code here
    X_poly = np.zeros((X.shape[0], degree))
    for d in range(degree):
        X_poly[:, d] = X[:, 0]**(d+1)
    return X_poly
```

```
# TODO: Implement polynomial regression using linear regression
def poly_regression(X_train, y_train, X_test, degree):

    # polynomial features
    X_train_poly = create_poly_features(X_train, degree)
    X_test_poly = create_poly_features(X_test, degree)


    X_train_b = np.c_[np.ones((X_train_poly.shape[0], 1)), X_train_poly]

    # normal equation
    theta =
np.linalg.inv(X_train_b.T.dot(X_train_b)).dot(X_train_b.T).dot(y_train)


    bias = theta[0, 0]
    coeffs = theta[1:, 0]


    train_pred = X_train_poly.dot(coeffs) + bias
    test_pred = X_test_poly.dot(coeffs) + bias

    return train_pred, test_pred, coeffs, bias

# Test with different polynomial degrees
degrees = [1, 3, 7, 15]
plt.figure(figsize=(15, 10))

for i, degree in enumerate(degrees):
    # Fit and predictions
    train_pred, test_pred, coeffs, bias = poly_regression(X_train,
y_train, X_test, degree)

    # MSE
    train_mse = np.mean((y_train - train_pred.reshape(-1, 1))**2)
    test_mse = np.mean((y_test - test_pred.reshape(-1, 1))**2)


    plt.subplot(2, 2, i+1)
    plt.scatter(X_train, y_train, color='blue', alpha=0.6,
label='Training data')
    plt.scatter(X_test, y_test, color='green', alpha=0.6, label='Testing
data')
```

```
    # Sort
    X_sorted = np.sort(X, axis=0)
    X_poly = create_poly_features(X_sorted, degree)
    y_poly = X_poly.dot(coeffs) + bias

    plt.plot(X_sorted, y_poly, color='red', linewidth=2,
             label=f'Polynomial (degree={degree})')
    plt.title(f'Degree {degree}\nTrain MSE: {train_mse:.2f}, Test MSE:
{test_mse:.2f}')
    plt.xlabel('X')
    plt.ylabel('y')
    plt.legend()

plt.tight_layout()
plt.show()
```

- Which polynomial degree provides the best fit for this data? How can you tell?
- What happens to the training and testing MSE as the degree increases? Explain this behavior.
- How would you prevent overfitting in high-degree polynomial models?
- Modify the code to add a regularization term to the polynomial regression. How does this affect the results?

Task 2 - Neural Network Implementation

Neural networks are powerful models capable of learning complex patterns in data. In this task, you'll implement a simple feedforward neural network with one hidden layer from scratch. You'll train it on a nonlinear classification problem (circular decision boundary) that would be difficult for linear models to solve. This implementation demonstrates the forward and backward propagation algorithms, introduces activation functions, and shows how neural networks can learn complex decision boundaries.

Task2.py

```
import numpy as np
import matplotlib.pyplot as plt

# Generate classification data
np.random.seed(42)
X = np.random.randn(200, 2)
y = (np.sum(X**2, axis=1) < 2).astype(int).reshape(-1, 1)  # Circle pattern
```

```python
# Split data
split_idx = int(0.8 * len(X))
X_train, X_test = X[:split_idx], X[split_idx:]
y_train, y_test = y[:split_idx], y[split_idx:]

# TODO: Implement a neural network with one hidden layer
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size,
learning_rate=0.01):

        self.learning_rate = learning_rate


        self.W1 = np.random.randn(input_size, hidden_size) * 0.01
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size) * 0.01
        self.b2 = np.zeros((1, output_size))

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-np.clip(x, -500, 500)))

    def sigmoid_derivative(self, x):

        s = self.sigmoid(x)
        return s * (1 - s)

    def forward(self, X):
        """Forward pass through the network"""
        # First layer
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1)

        # Output layer
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = self.sigmoid(self.z2)

        return self.a2

    def backward(self, X, y, output):
        """Backward pass to update weights"""
        m = X.shape[0]

        # Output layer gradients
        dz2 = output - y
        dW2 = (1/m) * np.dot(self.a1.T, dz2)
        db2 = (1/m) * np.sum(dz2, axis=0, keepdims=True)
```

```
        # Hidden layer gradients
        dz1 = np.dot(dz2, self.W2.T) * self.sigmoid_derivative(self.z1)
        dW1 = (1/m) * np.dot(X.T, dz1)
        db1 = (1/m) * np.sum(dz1, axis=0, keepdims=True)

        # Update weights
        self.W2 -= self.learning_rate * dW2
        self.b2 -= self.learning_rate * db2
        self.W1 -= self.learning_rate * dW1
        self.b1 -= self.learning_rate * db1

    def train(self, X, y, epochs):
        """Train the neural network"""
        losses = []

        for epoch in range(epochs):
            # Forward pass
            output = self.forward(X)

            # Compute loss
            loss = -np.mean(y * np.log(output + 1e-8) + (1 - y) * np.log(1 -
output + 1e-8))
            losses.append(loss)

            # Backward pass
            self.backward(X, y, output)

            if epoch % 100 == 0:
                print(f"Epoch {epoch}, Loss: {loss:.4f}")

        return losses

    def predict(self, X, threshold=0.5):
        """Make predictions"""
        output = self.forward(X)
        return (output >= threshold).astype(int)


nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1,
learning_rate=0.1)
losses = nn.train(X_train, y_train, epochs=1000)


y_train_pred = nn.predict(X_train)
y_test_pred = nn.predict(X_test)

train_acc = np.mean(y_train == y_train_pred)
test_acc = np.mean(y_test == y_test_pred)
```

```
print(f"Training accuracy: {train_acc:.4f}")
print(f"Testing accuracy: {test_acc:.4f}")


plt.figure(figsize=(10, 4))
plt.plot(losses)
plt.title('Loss During Training')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.show()


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


plt.scatter(X_train[y_train[:, 0] == 0][:, 0], X_train[y_train[:, 0] == 0][:,
1],
            color='blue', label='Class 0 (train)')
plt.scatter(X_train[y_train[:, 0] == 1][:, 0], X_train[y_train[:, 0] == 1][:,
1],
            color='red', label='Class 1 (train)')


x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))


Z = nn.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)


plt.contourf(xx, yy, Z, alpha=0.3)
plt.contour(xx, yy, Z, colors='green', linewidths=0.5)

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.title('Neural Network Decision Boundary')
plt.show()
```

- What is the purpose of the sigmoid activation function in this neural network?
- Explain how backpropagation works in this implementation.
- Try different hidden layer sizes (e.g., 2, 8, 16 neurons). How does this affect model performance?

- How would you modify this network to use a different activation function like ReLU? What changes would be needed?
- The data has a circular decision boundary. Why is a neural network better suited for this problem than logistic regression?