```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist

# Load and preprocess the MNIST dataset
def load_preprocess_data():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    # Flatten the 28x28 images into 784-dimensional vectors
    x_train = x_train.reshape(x_train.shape[0], 28*28).astype('float32') / 255.0
    x_test = x_test.reshape(x_test.shape[0], 28*28).astype('float32') / 255.0

    # Binarize the labels: 0-4 are class 0, 5-9 are class 1
    y_train_binary = np.where(y_train < 5, 0, 1)
    y_test_binary = np.where(y_test < 5, 0, 1)

    return x_train, y_train_binary, x_test, y_test_binary

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

# 3. Logistic loss function
def logistic_loss(y_true, z, y_pred):
    # return np.mean(y_true * np.log(1 + np.exp(-z)) + (1 - y_true) * z)
    return np.mean(y_true * np.log(1 + np.exp(-y_pred)) + (1 - y_true) * y_pred)

# 4. Logistic regression model class
class LogisticRegression:
    def __init__(self, input_size):
        # Initialize weights
        self.w = np.random.randn(input_size) * 0.01

    def predict(self, X):
        # Linear combination of inputs and weights
        z = np.dot(X, self.w)
        # Apply sigmoid function to get probabilities
        y_pred = sigmoid(z)
        y_pred = np.clip(y_pred, 1e-9, 1 - 1e-9) # Avoiding infinite or zero log result
        return z, y_pred

    def compute_gradient(self, X, y_true, y_pred):
        # Compute the gradient of the loss function with respect to weights
        return np.dot(X.T, (1 - 2 * y_true + y_true * y_pred)) / X.shape[0]

    def update_weights(self, gradient, learning_rate):
        # Update weights using the gradient and learning rate
        self.w -= learning_rate * gradient
```

```
# # 5. Training function using SGD
# def train_model(model, X_train, y_train, epochs, batch_size, learning_rate):
#     losses = []

#     for epoch in range(epochs):
#         # Shuffle the training data
#         perm = np.random.permutation(len(X_train))
#         X_train_shuffled = X_train[perm]
#         y_train_shuffled = y_train[perm]

#         batch_losses = []

#         # Mini-batch gradient descent
#         for i in range(0, len(X_train), batch_size):
#             X_batch = X_train_shuffled[i:i + batch_size]
#             y_batch = y_train_shuffled[i:i + batch_size]

#             # Forward pass: make predictions
#             z, y_pred = model.predict(X_batch)

#             # Compute the loss
#             loss = logistic_loss(y_batch, z, y_pred)
#             batch_losses.append(loss)

#             # Backward pass: compute gradient
#             gradient = model.compute_gradient(X_batch, y_batch, y_pred)

#             # Update the model weights
#             model.update_weights(gradient, learning_rate)

#         # Average loss for the epoch
#         epoch_loss = np.mean(batch_losses)
#         losses.append(epoch_loss)
#         if (epoch+1) % 10 == 0:
#             print(f'Epoch {epoch+1}/{epochs}, Loss: {epoch_loss}')

#     return losses

# 5. Training function using SGD and tracking accuracy for each epoch
def train_model_with_accuracy(model, X_train, y_train, X_test, y_test, epochs, batch_size,
learning_rate):
    train_losses = []
    test_accuracies = []

    for epoch in range(epochs):
        # Shuffle the training data
        perm = np.random.permutation(len(X_train))
        X_train_shuffled = X_train[perm]
        y_train_shuffled = y_train[perm]
```

```
        batch_losses = []

        # Mini-batch gradient descent
        for i in range(0, len(X_train), batch_size):
            X_batch = X_train_shuffled[i:i + batch_size]
            y_batch = y_train_shuffled[i:i + batch_size]

            # Forward pass: make predictions
            z, y_pred = model.predict(X_batch)

            # Compute the training loss for this batch
            loss = logistic_loss(y_batch, z, y_pred)
            batch_losses.append(loss)

            # Backward pass: compute gradient
            gradient = model.compute_gradient(X_batch, y_batch, y_pred)

            # Update the model weights
            model.update_weights(gradient, learning_rate)

        # Average training loss for the epoch
        epoch_train_loss = np.mean(batch_losses)
        train_losses.append(epoch_train_loss)

        # Compute the test accuracy at the end of each epoch
        _, y_test_pred_prob = model.predict(X_test)
        y_test_pred_labels = (y_test_pred_prob >= 0.5).astype(int)
        test_accuracy = np.mean(y_test_pred_labels == y_test)  # Calculate test accuracy
        test_accuracies.append(test_accuracy)

        # Print training loss and test accuracy every 10 epochs
        if (epoch+1) % 10 == 0:
            print(f'Epoch {epoch+1}/{epochs}, Train Loss: {epoch_train_loss:.4f}, Test Accuracy: 
{test_accuracy * 100:.2f}%')

    return train_losses, test_accuracies


# Main code to execute the training process
x_train, y_train, x_test, y_test = load_preprocess_data()
input_size = x_train.shape[1]  # Input size is 784 (28x28 flattened)

# Initialize the logistic regression model
logistic_model = LogisticRegression(input_size)

# Train the model using SGD
epochs = 100
batch_size = 256
```

```
learning_rate = 0.001

# Train the model and track accuracy over epochs
train_losses, test_accuracies = train_model_with_accuracy(logistic_model, x_train, y_train,
x_test, y_test, epochs, batch_size, learning_rate)

# Plot training loss over epochs
plt.plot(range(1, len(train_losses)+1), train_losses, label='Train Loss')
plt.title('Training Loss over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.savefig('p3_training_loss_lr005')
plt.clf()

# Plot test accuracy over epochs
plt.plot(range(1, len(test_accuracies)+1), test_accuracies, label='Test Accuracy')
plt.title('Test Accuracy over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.savefig('p3_testing_accuracy_lr005')
```