task i

Implementation of Softmax Function:

```
import numpy as np

def logistic_function(x):
    y = 1 / (1 + np.exp(-x))
    return y

test_val = 0
result = logistic_function(test_val)
print(result)
```

```
0.5
```

Test Cases for Softmax Function:

```
import numpy as np

def logistic_function(x):
    y = 1 / (1 + np.exp(-x))
    return y

def test_logistic_function():
    x_scalar = 0
    expected_output_scalar = round(1 / (1 + np.exp(0)), 3)
    assert round(logistic_function(x_scalar), 3) == expected_output_scalar, "Test failed for scalar input"

    x_pos = 2
    expected_output_pos = round(1 / (1 + np.exp(-2)), 3)
    assert round(logistic_function(x_pos), 3) == expected_output_pos, "Test failed for positive scalar input"

    x_neg = -3
    expected_output_neg = round(1 / (1 + np.exp(3)), 3)
    assert round(logistic_function(x_neg), 3) == expected_output_neg, "Test failed for negative scalar input"

    x_array = np.array([0, 2, -3])
    expected_output_array = np.array([0.5, 0.881, 0.047])
    assert np.all(np.round(logistic_function(x_array), 3) == expected_output_array), "Test failed for numpy array input"



test_logistic_function()
```

Implementation of Categorical Log-Loss Function:

```
import numpy as np

def log_loss(y_true, y_pred):
    y_pred = np.clip(y_pred, 1e-10, 1 - 1e-10)
    loss = -y_true * np.log(y_pred) - (1 - y_true) * np.log(1 - y_pred)
    return loss

print(log_loss(1, 0.9))
```

```
0.10536051565782628
```

Test Case for log - loss function

```
import numpy as np

def log_loss(y_true, y_pred):
    y_pred = np.clip(y_pred, 1e-10, 1 - 1e-10)
    loss = -y_true * np.log(y_pred) - (1 - y_true) * np.log(1 - y_pred)
    return loss

def test_log_loss():
    y_true = 1
```

```
    y_pred = 0.2
    expected_loss = -(1 * np.log(0.2)) - (0 * np.log(0.8))
    assert np.isclose(log_loss(y_true, y_pred), expected_loss, atol=1e-6)
    print(log_loss(y_true, y_pred))

test_log_loss()
```

```
1.6094379124341003
```

Implemenetation of Cost Function:

```
import numpy as np

def cost_function(y_true, y_pred):
    assert len(y_true) == len(y_pred), "Length of true values and length of predicted values do not match"
    n = len(y_true)
    loss_vec = log_loss(y_true, y_pred)
    cost = np.sum(loss_vec) / n
    return cost

# Running one test case
y_t = np.array([1, 0, 1])
y_p = np.array([0.9, 0.1, 0.8])
print(cost_function(y_t, y_p))
```

```
0.14462152754328741
```

Testing the Cost Function:

```
import numpy as np

def cost_function(y_true, y_pred):
    assert len(y_true) == len(y_pred), "Length of true values and length of predicted values do not match"
    n = len(y_true)
    loss_vec = log_loss(y_true, y_pred)
    cost = np.sum(loss_vec) / n
    return cost

def test_cost_function():
    y_true = np.array([1, 0, 1])
    y_pred = np.array([0.9, 0.1, 0.8])

    expected_cost = (-(1 * np.log(0.9)) - (0 * np.log(0.1)) +
                     -(0 * np.log(0.1)) - (1 * np.log(0.9)) +
                     -(1 * np.log(0.8)) - (0 * np.log(0.2))) / 3

    result = cost_function(y_true, y_pred)
    assert np.isclose(result, expected_cost, atol=1e-6), f"Test failed: {result} != {expected_cost}"
    print(result)

test_cost_function()
```

```
0.14462152754328741
```

Implementation of Cost Function for Logistic/Sigmoid Regression

```
import numpy as np

def costfunction_logreg(X, y, w, b):
    n, d = X.shape
    assert len(y) == n, "Number of feature observations and number of target observations do not match."
    assert len(w) == d, "Number of features and number of weight parameters do not match."

    # Compute z using np.dot (z = Xw + b)
    z = np.dot(X, w) + b

    # Compute predictions using logistic function (sigmoid)
    y_pred = logistic_function(z)

    # Compute the cost using the cost function
    cost = cost_function(y, y_pred)
    return cost
```

```
# Testing the Function:
X, y, w, b = np.array([[10, 20], [-10, 10]]), np.array([1, 0]), np.array([0.5, 1.5]), 1
print(f"cost for logistic regression(X = {X}, y = {y}, w = {w}, b = {b}) = {costfunction_logreg(X, y, w, b)}")
```

```
cost for logistic regression(X = [[ 10  20]
 [-10  10]], y = [1 0], w = [0.5 1.5], b = 1) = 5.500008350834906
```

Computing Gradients for Sigmoid Regression:

```
import numpy as np

def compute_gradient(X, y, w, b):
    n, d = X.shape
    assert len(y) == n, f"Expected y to have {n} elements, but got {len(y)}"
    assert len(w) == d, f"Expected w to have {d} elements, but got {len(w)}"

    # Compute predictions using logistic function (sigmoid)
    z = np.dot(X, w) + b
    y_pred = logistic_function(z)

    # Compute gradients
    err = y_pred - y
    grad_w = (1/n) * np.dot(X.T, err)
    grad_b = (1/n) * np.sum(err)

    return grad_w, grad_b

# Running one test case
X_test = np.array([[0.5, 0.2], [0.1, 0.8]])
y_test = np.array([1, 0])
w_test = np.array([0.1, -0.1])
b_test = 0.0

print(compute_gradient(X_test, y_test, w_test, b_test))
```

```
(array([-0.09899978,  0.1437528 ]), np.float64(-0.004996710057884712))
```

A simple assertion test for compute gradient function:

```
import numpy as np

def compute_gradient(X, y, w, b):
    n, d = X.shape
    z = np.dot(X, w) + b
    y_pred = logistic_function(z)

    err = y_pred - y
    grad_w = (1/n) * np.dot(X.T, err)
    grad_b = (1/n) * np.sum(err)

    return grad_w, grad_b

# Simple test case
X = np.array([[10, 20], [-10, 10]])
y = np.array([1, 0])
w = np.array([0.5, 1.5])
b = 1

try:
    grad_w, grad_b = compute_gradient(X, y, w, b)
    print("Gradients computed successfully.")
    print(f"grad_w: {grad_w}")
    print(f"grad_b: {grad_b}")
except AssertionError as e:
    print(f"Assertion error: {e}")
```

```
Gradients computed successfully.
grad_w: [-4.99991649  4.99991649]
grad_b: 0.4999916492890759
```

Gradient Descent for Sigmoid Regression:

```
import numpy as np
```

```
def gradient_descent(X, y, w, b, alpha, n_iter, show_cost=False, show_params=True):
    n, d = X.shape
    assert len(y) == n, "Number of observations in X and y do not match"
    assert len(w) == d, "Number of features in X and w do not match"

    cost_history = []
    params_history = []

    for i in range(n_iter):
        # Compute gradients
        grad_w, grad_b = compute_gradient(X, y, w, b)

        # Update weights and bias
        w -= alpha * grad_w
        b -= alpha * grad_b

        # Compute cost
        cost = costfunction_logreg(X, y, w, b)

        # Store cost and parameters
        cost_history.append(cost)
        params_history.append((w.copy(), b))

        # Optionally print cost and parameters
        if show_cost and (i % 100 == 0 or i == n_iter - 1):
            print(f"Iteration {i}: Cost = {cost:.6f}")
        if show_params and (i % 100 == 0 or i == n_iter - 1):
            print(f"Iteration {i}: w = {w}, b = {b:.6f}")

    return w, b, cost_history, params_history

# Test the gradient_descent function with sample data
X = np.array([[0.1, 0.2], [-0.1, 0.1]])
y = np.array([1, 0])
w = np.zeros(X.shape[1])
b = 0.0
alpha = 0.1
n_iter = 1000

# Perform gradient descent (reduced iterations for quick check)
w_out, b_out, cost_history, params_history = gradient_descent(X, y, w, b, alpha, n_iter, show_cost=True, show_params=False)

print("\nFinal parameters:")
print(f"w: {w_out}, b: {b_out}")
print(f"Final cost: {cost_history[-1]:.6f}")
```

```
======================================================================
TESTING SOFTMAX FUNCTION
======================================================================

Test Case 1: Basic 2D Input (3 samples, 3 classes)
----------------------------------------------------------------------
Input logits:
[[1. 2. 3.]
 [1. 1. 1.]
 [3. 2. 1.]]

Softmax probabilities:
[[0.09003057 0.24472847 0.66524096]
 [0.33333333 0.33333333 0.33333333]
 [0.66524096 0.24472847 0.09003057]]

Row sums (should all be 1.0): [1. 1. 1.]
✓ PASSED

======================================================================
TESTING LOSS FUNCTION
======================================================================

True labels:  [0 1 0]
Predictions:  [0.09 0.67 0.24]
Loss:         0.400478
Expected:     0.400478
✓ PASSED
```

```
Start coding or generate with AI.
```