Implementation of Sigmoid Regression from Scratch.

This Notebook has two Section:

1. Building Helper Function: In this section we will implement various helper function required for scratch implementation of Sigmoid Regression.
2. Putting Helper Function to Action: In this section we will compile all our helper function to train and test the Sigmoid Regression on Provided Dataset.

Building a Helper Function

Sigmoid Function:

A function $[\sigma: R → R] $ is said to be a sigmoid function, if the function is bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point and exactly on inflection point. The sigmoid fucntion has a characterstic "S" shaped curved also known as sigmoid curve. [--- Wikipedia]

A comon example of a sigmoid function is the logistic function described as below:

$$\sigma(x) = \frac{1}{1+e^{-x}},$$

for $x \in R$.

The next two code blocks construct and plot this function. ___

```
def logistic_function(x):
  """
  Computes the logistic function applied to any value of x.
  Arguments:
    x: scalar or numpy array of any size.
  Returns:
    y: logistic function applied to x.
  """
  import numpy as np
  y = 1/(1 + np.exp(-x))
  return y

import numpy as np

def test_logistic_function():
    """
    Test cases for the logistic_function.
    """
    # Test with scalar input
    x_scalar = 0
    expected_output_scalar = round(1 / (1 + np.exp(0)), 3)  # Expected
output: 0.5
    assert round(logistic_function(x_scalar), 3) ==
expected_output_scalar, "Test failed for scalar input"

    # Test with positive scalar input
    x_pos = 2
    expected_output_pos = round(1 / (1 + np.exp(-2)), 3)  # Expected
output: ~0.881
    assert round(logistic_function(x_pos), 3) == expected_output_pos,
"Test failed for positive scalar input"

    # Test with negative scalar input
    x_neg = -3
    expected_output_neg = round(1 / (1 + np.exp(3)), 3)  # Expected
output: ~0.047
    assert round(logistic_function(x_neg), 3) == expected_output_neg,
"Test failed for negative scalar input"

    # Test with numpy array input
    x_array = np.array([0, 2, -3])
    expected_output_array = np.array([0.5, 0.881, 0.047])  # Adjusted
expected values rounded to 3 decimals
    # Use np.round to round the array element-wise and compare
    assert np.all(np.round(logistic_function(x_array), 3) ==
expected_output_array), "Test failed for numpy array input"

    print("All tests passed!")

# Run the test case
test_logistic_function()
```

```
All tests passed!
```

Explanation of the test cases:

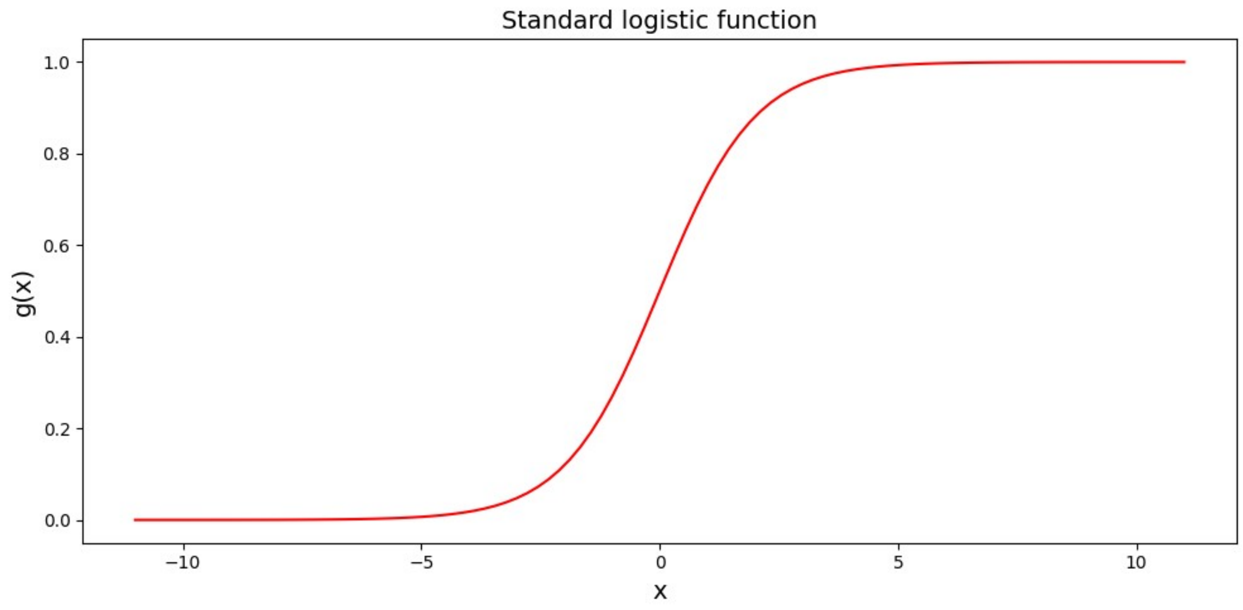
Scalar input test: We check if the function gives the correct output for a scalar input (e.g., 0, 2, -3).

Numpy array test: We verify that the function can handle a numpy array as input and return the correct element-wise logistic function values.

Assertions: np.isclose and np.allclose are used for comparison to avoid issues with floating-point precision errors.

Running this test will check that the function works for both scalar and array inputs and ensures the correctness of the output values.

```python
# Plooting the sigmoid function:
import matplotlib.pyplot as plt
import numpy as np
plt.figure(figsize = (10, 5))
x = np.linspace(-11, 11, 100)
plt.plot(x, logistic_function(x), color = 'red')
plt.xlabel("x", fontsize = 14)
plt.ylabel("g(x)", fontsize = 14)
plt.title("Standard logistic function", fontsize = 14)
plt.tight_layout()
plt.show()
```



Loss Function:

In general, loss function corresponds to observed error value between target value and predicted value for single observation/data points. For Sigmoid Regression and Binary Classification we use log - loss given by:

$$L(y, \hat{y}) = -y \log(\hat{y}) - (1-y) \log(1-\hat{y}),$$

Where:

- y : True target value (taking values 0 or 1)
- $\hat{y}$: Predicted target value (predicted probability of y being 1 and vice versa.)

The basic intution behind the log-loss function is, the loss value should be minimum when our predicted probability values are closer to true target value.

```python
def log_loss(y_true, y_pred):
    """
```

```
  Computes log loss for true target value y ={0 or 1} and predicted
target value y' inbetween {0-1}.
  Arguments:
    y_true (scalar): true target value {0 or 1}.
    y_pred (scalar): predicted taget value {0-1}.
  Returns:
    loss (float): loss/error value
  """
  import numpy as np
  # Ensure y_pred is clipped to avoid log(0)
  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
```

Verify the Intution.

```
# Test function:
y_true, y_pred = 0, 0.1
print(f'log loss({y_true}, {y_pred}) ==> {log_loss(y_true, y_pred)}')
print("+++++++++++++--------------------------+++++++++++++++++++++++
+")
y_true, y_pred = 1, 0.9
print(f'log loss({y_true}, {y_pred}) ==> {log_loss(y_true, y_pred)}')
```

```
log loss(0, 0.1) ==> 0.10536051565782628
+++++++++++++--------------------------++++++++++++++++++++++++
log loss(1, 0.9) ==> 0.10536051565782628
```

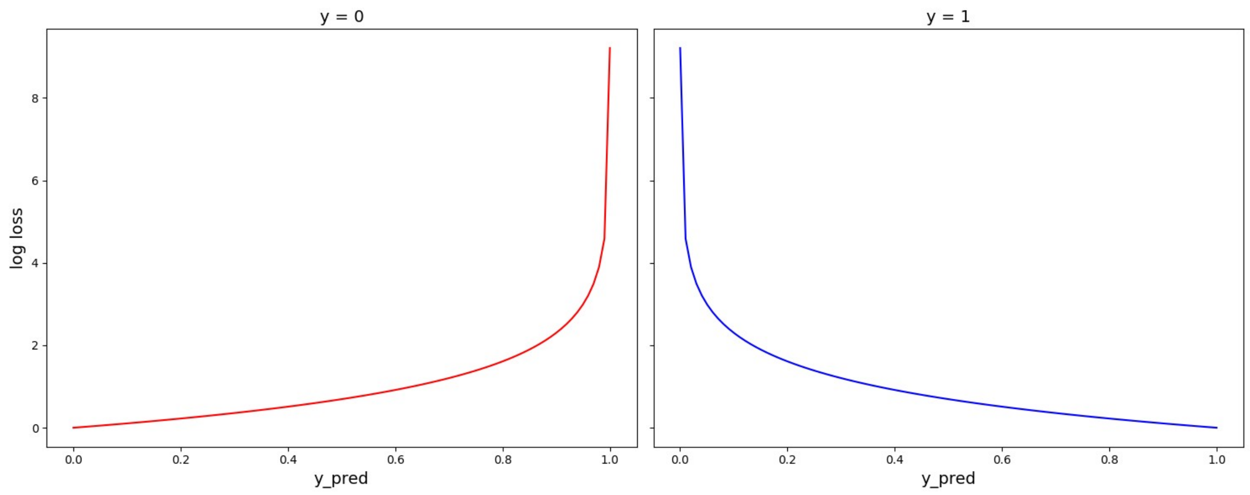
```
# Test function:for
y_true, y_pred = 0, 0.9
print(f'log loss({y_true}, {y_pred}) ==> {log_loss(y_true, y_pred)}')
print("+++++++++++++--------------------------+++++++++++++++++++++++
+")
y_true, y_pred = 1, 0.1
print(f'log loss({y_true}, {y_pred}) ==> {log_loss(y_true, y_pred)}')
```

```
log loss(0, 0.9) ==> 2.302585092994046
+++++++++++++--------------------------++++++++++++++++++++++++
log loss(1, 0.1) ==> 2.3025850929940455
```

Test the Loss Function

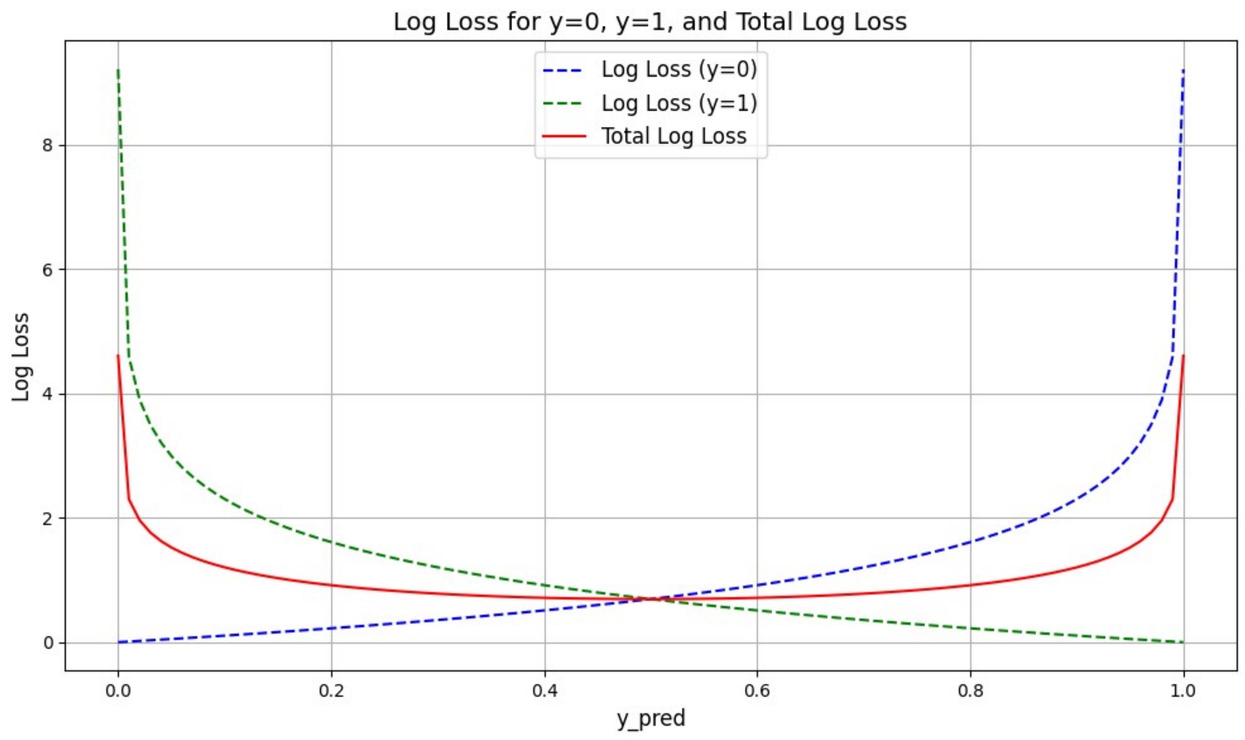
```
def test_log_loss():
    """
    Test cases for the log_loss function.
    """
    import numpy as np

    # Test case 1: Perfect prediction (y_true = 1, y_pred = 1)
    y_true = 1
    y_pred = 1
    expected_loss = 0.0  # Log loss is 0 for perfect prediction
    assert np.isclose(log_loss(y_true, y_pred), expected_loss), "Test
failed for perfect prediction (y_true=1, y_pred=1)"

    # Test case 2: Perfect prediction (y_true = 0, y_pred = 0)
    y_true = 0
    y_pred = 0
    expected_loss = 0.0  # Log loss is 0 for perfect prediction
    assert np.isclose(log_loss(y_true, y_pred), expected_loss), "Test
failed for perfect prediction (y_true=0, y_pred=0)"

    # Test case 3: Incorrect prediction (y_true = 1, y_pred = 0)
    y_true = 1
    y_pred = 0
    try:
        log_loss(y_true, y_pred)  # This should raise an error due to
log(0)
    except ValueError:
        pass  # Test passed if ValueError is raised for log(0)

    # Test case 4: Incorrect prediction (y_true = 0, y_pred = 1)
    y_true = 0
    y_pred = 1
    try:
        log_loss(y_true, y_pred)  # This should raise an error due to
log(0)
    except ValueError:
        pass  # Test passed if ValueError is raised for log(0)

    # Test case 5: Partially correct prediction
    y_true = 1
    y_pred = 0.8
    expected_loss = -(1 * np.log(0.8)) - (0 * np.log(0.2))  # ~0.2231
    assert np.isclose(log_loss(y_true, y_pred), expected_loss,
atol=1e-6), "Test failed for partially correct prediction (y_true=1,
y_pred=0.8)"

    y_true = 0
    y_pred = 0.2
    expected_loss = -(0 * np.log(0.2)) - (1 * np.log(0.8))  # ~0.2231
    assert np.isclose(log_loss(y_true, y_pred), expected_loss,
atol=1e-6), "Test failed for partially correct prediction (y_true=0,
y_pred=0.2)"

    print("All tests passed!")

# Run the test case
test_log_loss()
```

```
All tests passed!

# Plot the loss Function:
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 2, figsize = (15, 6), sharex = True, sharey
= True)
y_pred = np.linspace(0.0001, 0.9999, 100)
ax[0].plot(y_pred, log_loss(0, y_pred), color = 'red')
ax[0].set_title("y = 0", fontsize = 14)
ax[0].set_xlabel("y_pred", fontsize = 14)
ax[0].set_ylabel("log loss", fontsize = 14)
ax[1].plot(y_pred, log_loss(1, y_pred), color = 'blue')
ax[1].set_title("y = 1", fontsize = 14)
ax[1].set_xlabel("y_pred", fontsize = 14)
plt.tight_layout()
plt.show()
```



plot for total loss

```
# Generate predicted probabilities
y_pred = np.linspace(0.0001, 0.9999, 100)

# Compute log losses for y = 0 and y = 1
log_loss_0 = log_loss(0, y_pred)
log_loss_1 = log_loss(1, y_pred)

# Compute total log loss as a weighted sum (assuming equal weights
here for simplicity)
total_log_loss = 0.5 * log_loss_0 + 0.5 * log_loss_1

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(y_pred, log_loss_0, label="Log Loss (y=0)", color="blue",
linestyle="--")
plt.plot(y_pred, log_loss_1, label="Log Loss (y=1)", color="green",
linestyle="--")
plt.plot(y_pred, total_log_loss, label="Total Log Loss", color="red")
plt.title("Log Loss for y=0, y=1, and Total Log Loss", fontsize=14)
plt.xlabel("y_pred", fontsize=12)
plt.ylabel("Log Loss", fontsize=12)
plt.legend(fontsize=12)
plt.grid(True)
plt.tight_layout()
plt.show()
```



Cost Function - Binary Classification:

As described above, we determine the cost function as an average of loss function value calculated for each observation/datapoints.

Let $y=[y_1, \ldots, y_n]$ be the true target values{(0 or 1)}

and $\hat{y}=[\hat{y}_1, \ldots, \hat{y}_n]$ be the corresponding predicted target values in between $\{[0 \le \hat{y} \le 1]\}$,

Then the cost function be:

$$Cost(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \hat{y}_i).$$

```
# Cost function - using vectorization
def cost_function(y_true, y_pred):
    """
    Computes log loss for inputs true value (0 or 1) and predicted
value (between 0 and 1)
    Args:
      y_true    (array_like, shape (m,)): array of true values (0 or
1)
      y_pred (array_like, shape (m,)): array of predicted values
(probability of y_pred being 1)
    Returns:
      cost (float): nonnegative cost corresponding to y_true and
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 = np.array([log_loss(y_true[i], y_pred[i]) for i in
range(n)])
    cost = np.dot(loss_vec, np.ones(n)) / n
    return cost

y_true, y_pred = np.array([0, 1, 0]), np.array([0.4, 0.6, 0.25])
print(f"cost_function({y_true}, {y_pred}) = {cost_function(y_true,
y_pred)}")
```

```
cost_function([0 1 0], [0.4  0.6  0.25]) = 0.43644443999458743
```

```
import numpy as np

# Test function for the cost_function
def test_cost_function():
    # Test case 1: Simple example with known expected cost
    y_true = np.array([1, 0, 1])
    y_pred = np.array([0.9, 0.1, 0.8])

    # Expected output: Manually calculate cost for these values
    # log_loss(y_true, y_pred) for each example
    expected_cost = (-(1 * np.log(0.9)) - (1 - 1) * np.log(1 - 0.9) +
                     -(0 * np.log(0.1)) - (1 - 0) * np.log(1 - 0.1) +
                     -(1 * np.log(0.8)) - (1 - 1) * np.log(1 - 0.8)) /
3

    # Call the cost_function to get the result
    result = cost_function(y_true, y_pred)

    # Assert that the result is close to the expected cost with a
tolerance of 1e-6
    assert np.isclose(result, expected_cost, atol=1e-6), f"Test
failed: {result} != {expected_cost}"

    print("Test passed for simple case!")
```

```
# Run the test case
test_cost_function()

Test passed for simple case!
```

Extending the cost function for logistic regression to be used with model parameters.

Function we are estimating:

$$\hat{y} = \sigma\left(x \cdot w^T + b\right) = \frac{1}{1 + e^{-\left(x \cdot w^T + b\right)}}.$$

Where:

- w parameters (Coefficinet of feature variable) also known as weights. -b parameters (intercept of the function) also known as bias.

Assume;

- $X_{n \times d} \in R$: Feature Matirix of $n \times d$ independent variables;
- Y_n: Vector of n dependent variables;
- $\hat{Y}_n$: Vector of n predicted variables; and are represented as follows:

$$ \mathbf{X_{n\times d}} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,d} \newline x_{2,1} & x_{2,2} & \cdots & x_{2,d} \newline \vdots & \vdots & \ddots & \vdots \newline x_{n,1} & x_{n,2} & \cdots & x_{n,d} \end{pmatrix},\;\;\;\; \mathbf{Y_n} = \begin{pmatrix} y_1 \newline y_2 \newline \vdots \newline y_n \end{pmatrix},\;\;\;\; \mathbf{\hat{Y}_n} = \begin{pmatrix} \sigma\left(\mathbf{x_1} \cdot \mathbf{{w^T}} + b_1\right) \newline \sigma\left(\mathbf{x_2} \cdot \mathbf{w^T} + b_2\right) \newline \vdots \newline \sigma\left(\mathbf{x_n} \cdot \mathbf{w^T} + b_n\right) \end{pmatrix}. \tag{2} $$

Here:

$$x_i = x_{i,1} \cdot w_{i,1}^T + \ldots + x_{i,d} \cdot w_{i,d}^T + b_i$$

Now rewrite cost funtion defined in (0) for model parameters as:

$$L(w,b) := C(y, \hat{y} \vee X, w, b) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, \frac{1}{1 + e^{-\left(x_i \cdot w + b\right)}}\right)$$

$$¿ \frac{1}{n}\sum_{i=1}^{n}\left[-y_i \log\left(\frac{1}{1 + e^{-\left(x_i \cdot w + b\right)}}\right) - \left(1 - y_i\right)\log\left(1 - \frac{1}{1 + e^{-\left(x_i \cdot w + b\right)}}\right)\right].$$

```
# Function to compute cost function in terms of model parameters -
using vectorization
def costfunction_logreg(X, y, w, b):
    """
    Computes the cost function, given data and model parameters.
    Args:
      X (ndarray, shape (m,n)): data on features, m observations with
n features.
      y (array_like, shape (m,)): array of true values of target (0 or
1).
      w (array_like, shape (n,)): weight parameters of the model.
      b (float): bias parameter of the model.
    Returns:
      cost (float): nonnegative cost corresponding to y and y_pred.
    """
    m, n = X.shape
    assert len(y) == m, "Number of feature observations and number of
target observations do not match."
    assert len(w) == n, "Number of features and number of weight
parameters do not match."

    # Compute z using np.dot
    z = np.dot(X, w) + b  # Matrix-vector multiplication and adding
bias

    # 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
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
```

```
def compute_gradient(X, y, w, b):
    """
    Computes gradients of the cost function with respect to model
parameters.
    Args:
      X (ndarray, shape (n,d)): Input data, n observations with d
features
      y (array_like, shape (n,)): True labels (0 or 1)
      w (array_like, shape (d,)): Weight parameters of the model
      b (float): Bias parameter of the model
    Returns:
      grad_w (array_like, shape (d,)): Gradients of the cost function
with respect to the weight parameters
```

```
        grad_b (float): Gradient of the cost function with respect to
the bias parameter
    """
    n, d = X.shape  # X has shape (n, d)
    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)
    y_pred = logistic_function(np.dot(X, w) + b)  # Compute z = X * w
+ b

    # Compute gradients
    grad_w = np.dot(X.T, (y_pred - y)) / n  # Gradient w.r.t weights,
shape (d,)
    grad_b = np.dot(np.ones(n), (y_pred - y)) / n  # Gradient w.r.t
bias, scalar

    return grad_w, grad_b
```

Simple Test for Gradient Computations:

```
# Simple test case
X = np.array([[10, 20], [-10, 10]])  # shape (2, 2)
y = np.array([1, 0])  # shape (2,)
w = np.array([0.5, 1.5])  # shape (2,)
b = 1  # scalar

# Assertion tests
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
```

Implementation of Gradient Descent for Sigmoid Regresssion.

```
def gradient_descent(X, y, w, b, alpha, n_iter, show_cost=False,
show_params=True):
    """
    Implements batch gradient descent to optimize logistic regression
parameters.
```

```
    Args:
      X (ndarray, shape (n,d)): Data on features, n observations with
d features
      y (array_like, shape (n,)): True values of target (0 or 1)
      w (array_like, shape (d,)): Initial weight parameters
      b (float): Initial bias parameter
      alpha (float): Learning rate
      n_iter (int): Number of iterations
      show_cost (bool): If True, displays cost every 100 iterations
      show_params (bool): If True, displays parameters every 100
iterations

    Returns:
      w (array_like, shape (d,)): Optimized weight parameters
      b (float): Optimized bias parameter
      cost_history (list): List of cost values over iterations
      params_history (list): List of parameters (w, b) over iterations
    """
    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]])  # Shape (2, 2)
y = np.array([1, 0])                      # Shape (2,)
w = np.zeros(X.shape[1])                  # Shape (2,) - same as number
of features
b = 0.0                                   # Scalar
alpha = 0.1                               # Learning rate
n_iter = 100000                            # Number of iterations

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

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

```
Iteration 0: Cost = 0.692835
Iteration 100: Cost = 0.662662
Iteration 200: Cost = 0.634332
Iteration 300: Cost = 0.607704
Iteration 400: Cost = 0.582671
Iteration 500: Cost = 0.559128
Iteration 600: Cost = 0.536977
Iteration 700: Cost = 0.516126
Iteration 800: Cost = 0.496487
Iteration 900: Cost = 0.477978
Iteration 1000: Cost = 0.460524
Iteration 1100: Cost = 0.444052
Iteration 1200: Cost = 0.428497
Iteration 1300: Cost = 0.413797
Iteration 1400: Cost = 0.399895
Iteration 1500: Cost = 0.386736
Iteration 1600: Cost = 0.374272
Iteration 1700: Cost = 0.362457
Iteration 1800: Cost = 0.351248
Iteration 1900: Cost = 0.340607
Iteration 2000: Cost = 0.330495
Iteration 2100: Cost = 0.320880
Iteration 2200: Cost = 0.311730
Iteration 2300: Cost = 0.303016
Iteration 2400: Cost = 0.294710
Iteration 2500: Cost = 0.286789
Iteration 2600: Cost = 0.279228
Iteration 2700: Cost = 0.272007
Iteration 2800: Cost = 0.265104
Iteration 2900: Cost = 0.258502
Iteration 3000: Cost = 0.252182
Iteration 3100: Cost = 0.246129
Iteration 3200: Cost = 0.240328
Iteration 3300: Cost = 0.234764
Iteration 3400: Cost = 0.229425
Iteration 3500: Cost = 0.224299
Iteration 3600: Cost = 0.219373
Iteration 3700: Cost = 0.214637
Iteration 3800: Cost = 0.210081
Iteration 3900: Cost = 0.205697
Iteration 4000: Cost = 0.201474
Iteration 4100: Cost = 0.197406
Iteration 4200: Cost = 0.193484
Iteration 4300: Cost = 0.189700
Iteration 4400: Cost = 0.186049
Iteration 4500: Cost = 0.182524
Iteration 4600: Cost = 0.179119
Iteration 4700: Cost = 0.175828
Iteration 4800: Cost = 0.172646
Iteration 4900: Cost = 0.169568
Iteration 5000: Cost = 0.166589
Iteration 5100: Cost = 0.163705
Iteration 5200: Cost = 0.160912
Iteration 5300: Cost = 0.158205
Iteration 5400: Cost = 0.155581
Iteration 5500: Cost = 0.153036
Iteration 5600: Cost = 0.150568
Iteration 5700: Cost = 0.148172
Iteration 5800: Cost = 0.145846
Iteration 5900: Cost = 0.143586
Iteration 6000: Cost = 0.141392
Iteration 6100: Cost = 0.139258
Iteration 6200: Cost = 0.137184
Iteration 6300: Cost = 0.135167
Iteration 6400: Cost = 0.133205
Iteration 6500: Cost = 0.131295
Iteration 6600: Cost = 0.129436
Iteration 6700: Cost = 0.127625
Iteration 6800: Cost = 0.125862
Iteration 6900: Cost = 0.124143
Iteration 7000: Cost = 0.122469
Iteration 7100: Cost = 0.120836
Iteration 7200: Cost = 0.119243
Iteration 7300: Cost = 0.117690
Iteration 7400: Cost = 0.116175
Iteration 7500: Cost = 0.114695
Iteration 7600: Cost = 0.113251
Iteration 7700: Cost = 0.111841
Iteration 7800: Cost = 0.110464
Iteration 7900: Cost = 0.109118
Iteration 8000: Cost = 0.107804
Iteration 8100: Cost = 0.106518
Iteration 8200: Cost = 0.105262
Iteration 8300: Cost = 0.104033
Iteration 8400: Cost = 0.102832
Iteration 8500: Cost = 0.101656
Iteration 8600: Cost = 0.100506
Iteration 8700: Cost = 0.099380
Iteration 8800: Cost = 0.098278
Iteration 8900: Cost = 0.097199
Iteration 9000: Cost = 0.096142
Iteration 9100: Cost = 0.095107
Iteration 9200: Cost = 0.094093
Iteration 9300: Cost = 0.093100
Iteration 9400: Cost = 0.092126
Iteration 9500: Cost = 0.091172
Iteration 9600: Cost = 0.090236
Iteration 9700: Cost = 0.089318
Iteration 9800: Cost = 0.088418
Iteration 9900: Cost = 0.087536
Iteration 10000: Cost = 0.086670
Iteration 10100: Cost = 0.085820
Iteration 10200: Cost = 0.084986
Iteration 10300: Cost = 0.084168
Iteration 10400: Cost = 0.083364
Iteration 10500: Cost = 0.082575
Iteration 10600: Cost = 0.081800
Iteration 10700: Cost = 0.081039
Iteration 10800: Cost = 0.080292
Iteration 10900: Cost = 0.079558
Iteration 11000: Cost = 0.078836
Iteration 11100: Cost = 0.078127
Iteration 11200: Cost = 0.077430
Iteration 11300: Cost = 0.076745
Iteration 11400: Cost = 0.076072
Iteration 11500: Cost = 0.075409
Iteration 11600: Cost = 0.074758
Iteration 11700: Cost = 0.074118
Iteration 11800: Cost = 0.073488
Iteration 11900: Cost = 0.072868
Iteration 12000: Cost = 0.072259
Iteration 12100: Cost = 0.071659
Iteration 12200: Cost = 0.071068
Iteration 12300: Cost = 0.070487
Iteration 12400: Cost = 0.069915
Iteration 12500: Cost = 0.069352
Iteration 12600: Cost = 0.068798
Iteration 12700: Cost = 0.068252
Iteration 12800: Cost = 0.067714
Iteration 12900: Cost = 0.067185
Iteration 13000: Cost = 0.066663
Iteration 13100: Cost = 0.066149
Iteration 13200: Cost = 0.065643
Iteration 13300: Cost = 0.065145
Iteration 13400: Cost = 0.064653
Iteration 13500: Cost = 0.064169
Iteration 13600: Cost = 0.063692
Iteration 13700: Cost = 0.063221
Iteration 13800: Cost = 0.062757
Iteration 13900: Cost = 0.062300
Iteration 14000: Cost = 0.061849
Iteration 14100: Cost = 0.061405
Iteration 14200: Cost = 0.060966
Iteration 14300: Cost = 0.060534
Iteration 14400: Cost = 0.060108
Iteration 14500: Cost = 0.059687
Iteration 14600: Cost = 0.059272
Iteration 14700: Cost = 0.058862
Iteration 14800: Cost = 0.058459
Iteration 14900: Cost = 0.058060
Iteration 15000: Cost = 0.057667
Iteration 15100: Cost = 0.057278
Iteration 15200: Cost = 0.056895
Iteration 15300: Cost = 0.056517
Iteration 15400: Cost = 0.056144
Iteration 15500: Cost = 0.055775
Iteration 15600: Cost = 0.055411
Iteration 15700: Cost = 0.055052
Iteration 15800: Cost = 0.054697
Iteration 15900: Cost = 0.054346
Iteration 16000: Cost = 0.054000
Iteration 16100: Cost = 0.053659
Iteration 16200: Cost = 0.053321
Iteration 16300: Cost = 0.052987
Iteration 16400: Cost = 0.052658
Iteration 16500: Cost = 0.052333
Iteration 16600: Cost = 0.052011
Iteration 16700: Cost = 0.051693
Iteration 16800: Cost = 0.051379
Iteration 16900: Cost = 0.051069
Iteration 17000: Cost = 0.050762
Iteration 17100: Cost = 0.050459
Iteration 17200: Cost = 0.050159
Iteration 17300: Cost = 0.049863
Iteration 17400: Cost = 0.049571
Iteration 17500: Cost = 0.049281
Iteration 17600: Cost = 0.048995
Iteration 17700: Cost = 0.048712
Iteration 17800: Cost = 0.048432
Iteration 17900: Cost = 0.048156
Iteration 18000: Cost = 0.047882
Iteration 18100: Cost = 0.047612
Iteration 18200: Cost = 0.047344
Iteration 18300: Cost = 0.047079
Iteration 18400: Cost = 0.046818
Iteration 18500: Cost = 0.046559
Iteration 18600: Cost = 0.046302
Iteration 18700: Cost = 0.046049
Iteration 18800: Cost = 0.045798
Iteration 18900: Cost = 0.045550
Iteration 19000: Cost = 0.045305
Iteration 19100: Cost = 0.045062
Iteration 19200: Cost = 0.044822
Iteration 19300: Cost = 0.044584
Iteration 19400: Cost = 0.044348
Iteration 19500: Cost = 0.044115
Iteration 19600: Cost = 0.043885
Iteration 19700: Cost = 0.043656
Iteration 19800: Cost = 0.043430
Iteration 19900: Cost = 0.043207
Iteration 20000: Cost = 0.042985
Iteration 20100: Cost = 0.042766
Iteration 20200: Cost = 0.042549
Iteration 20300: Cost = 0.042334
Iteration 20400: Cost = 0.042121
Iteration 20500: Cost = 0.041911
Iteration 20600: Cost = 0.041702
Iteration 20700: Cost = 0.041495
Iteration 20800: Cost = 0.041291
Iteration 20900: Cost = 0.041088
Iteration 21000: Cost = 0.040888
Iteration 21100: Cost = 0.040689
Iteration 21200: Cost = 0.040492
Iteration 21300: Cost = 0.040297
Iteration 21400: Cost = 0.040104
Iteration 21500: Cost = 0.039912
Iteration 21600: Cost = 0.039722
Iteration 21700: Cost = 0.039535
Iteration 21800: Cost = 0.039348
Iteration 21900: Cost = 0.039164
Iteration 22000: Cost = 0.038981
Iteration 22100: Cost = 0.038800
Iteration 22200: Cost = 0.038621
Iteration 22300: Cost = 0.038443
Iteration 22400: Cost = 0.038267
Iteration 22500: Cost = 0.038092
Iteration 22600: Cost = 0.037919
Iteration 22700: Cost = 0.037748
Iteration 22800: Cost = 0.037577
Iteration 22900: Cost = 0.037409
Iteration 23000: Cost = 0.037242
Iteration 23100: Cost = 0.037076
Iteration 23200: Cost = 0.036912
Iteration 23300: Cost = 0.036749
Iteration 23400: Cost = 0.036588
Iteration 23500: Cost = 0.036428
Iteration 23600: Cost = 0.036270
Iteration 23700: Cost = 0.036112
Iteration 23800: Cost = 0.035956
Iteration 23900: Cost = 0.035802
Iteration 24000: Cost = 0.035649
Iteration 24100: Cost = 0.035497
Iteration 24200: Cost = 0.035346
Iteration 24300: Cost = 0.035196
Iteration 24400: Cost = 0.035048
Iteration 24500: Cost = 0.034901
Iteration 24600: Cost = 0.034755
Iteration 24700: Cost = 0.034611
Iteration 24800: Cost = 0.034467
Iteration 24900: Cost = 0.034325
Iteration 25000: Cost = 0.034184
Iteration 25100: Cost = 0.034044
Iteration 25200: Cost = 0.033905
Iteration 25300: Cost = 0.033767
Iteration 25400: Cost = 0.033631
Iteration 25500: Cost = 0.033495
Iteration 25600: Cost = 0.033361
Iteration 25700: Cost = 0.033227
Iteration 25800: Cost = 0.033095
Iteration 25900: Cost = 0.032963
Iteration 26000: Cost = 0.032833
Iteration 26100: Cost = 0.032704
Iteration 26200: Cost = 0.032575
Iteration 26300: Cost = 0.032448
Iteration 26400: Cost = 0.032322
Iteration 26500: Cost = 0.032196
Iteration 26600: Cost = 0.032072
Iteration 26700: Cost = 0.031948
Iteration 26800: Cost = 0.031826
Iteration 26900: Cost = 0.031704
Iteration 27000: Cost = 0.031583
Iteration 27100: Cost = 0.031463
Iteration 27200: Cost = 0.031344
Iteration 27300: Cost = 0.031226
Iteration 27400: Cost = 0.031109
Iteration 27500: Cost = 0.030993
Iteration 27600: Cost = 0.030877
Iteration 27700: Cost = 0.030763
Iteration 27800: Cost = 0.030649
Iteration 27900: Cost = 0.030536
Iteration 28000: Cost = 0.030424
Iteration 28100: Cost = 0.030312
Iteration 28200: Cost = 0.030202
Iteration 28300: Cost = 0.030092
Iteration 28400: Cost = 0.029983
Iteration 28500: Cost = 0.029875
Iteration 28600: Cost = 0.029768
Iteration 28700: Cost = 0.029661
Iteration 28800: Cost = 0.029555
Iteration 28900: Cost = 0.029450
Iteration 29000: Cost = 0.029345
Iteration 29100: Cost = 0.029242
Iteration 29200: Cost = 0.029139
Iteration 29300: Cost = 0.029036
Iteration 29400: Cost = 0.028935
Iteration 29500: Cost = 0.028834
Iteration 29600: Cost = 0.028734
Iteration 29700: Cost = 0.028634
Iteration 29800: Cost = 0.028535
Iteration 29900: Cost = 0.028437
Iteration 30000: Cost = 0.028340
Iteration 30100: Cost = 0.028243
Iteration 30200: Cost = 0.028147
Iteration 30300: Cost = 0.028051
Iteration 30400: Cost = 0.027956
Iteration 30500: Cost = 0.027862
Iteration 30600: Cost = 0.027768
Iteration 30700: Cost = 0.027675
Iteration 30800: Cost = 0.027583
Iteration 30900: Cost = 0.027491
Iteration 31000: Cost = 0.027400
Iteration 31100: Cost = 0.027309
Iteration 31200: Cost = 0.027219
Iteration 31300: Cost = 0.027130
Iteration 31400: Cost = 0.027041
Iteration 31500: Cost = 0.026953
Iteration 31600: Cost = 0.026865
Iteration 31700: Cost = 0.026778
Iteration 31800: Cost = 0.026691
Iteration 31900: Cost = 0.026605
Iteration 32000: Cost = 0.026519
Iteration 32100: Cost = 0.026435
Iteration 32200: Cost = 0.026350
Iteration 32300: Cost = 0.026266
Iteration 32400: Cost = 0.026183
Iteration 32500: Cost = 0.026100
Iteration 32600: Cost = 0.026018
Iteration 32700: Cost = 0.025936
Iteration 32800: Cost = 0.025854
Iteration 32900: Cost = 0.025774
Iteration 33000: Cost = 0.025693
Iteration 33100: Cost = 0.025613
Iteration 33200: Cost = 0.025534
Iteration 33300: Cost = 0.025455
Iteration 33400: Cost = 0.025377
Iteration 33500: Cost = 0.025299
Iteration 33600: Cost = 0.025222
Iteration 33700: Cost = 0.025145
Iteration 33800: Cost = 0.025068
Iteration 33900: Cost = 0.024992
Iteration 34000: Cost = 0.024916
Iteration 34100: Cost = 0.024841
Iteration 34200: Cost = 0.024767
Iteration 34300: Cost = 0.024692
Iteration 34400: Cost = 0.024619
Iteration 34500: Cost = 0.024545
Iteration 34600: Cost = 0.024472
Iteration 34700: Cost = 0.024400
Iteration 34800: Cost = 0.024328
Iteration 34900: Cost = 0.024256
Iteration 35000: Cost = 0.024185
Iteration 35100: Cost = 0.024114
Iteration 35200: Cost = 0.024043
Iteration 35300: Cost = 0.023973
Iteration 35400: Cost = 0.023904
Iteration 35500: Cost = 0.023834
Iteration 35600: Cost = 0.023766
Iteration 35700: Cost = 0.023697
Iteration 35800: Cost = 0.023629
Iteration 35900: Cost = 0.023561
Iteration 36000: Cost = 0.023494
Iteration 36100: Cost = 0.023427
Iteration 36200: Cost = 0.023361
Iteration 36300: Cost = 0.023295
Iteration 36400: Cost = 0.023229
Iteration 36500: Cost = 0.023163
Iteration 36600: Cost = 0.023098
Iteration 36700: Cost = 0.023034
Iteration 36800: Cost = 0.022969
Iteration 36900: Cost = 0.022905
Iteration 37000: Cost = 0.022842
Iteration 37100: Cost = 0.022778
Iteration 37200: Cost = 0.022715
Iteration 37300: Cost = 0.022653
Iteration 37400: Cost = 0.022590
Iteration 37500: Cost = 0.022529
Iteration 37600: Cost = 0.022467
Iteration 37700: Cost = 0.022406
Iteration 37800: Cost = 0.022345
Iteration 37900: Cost = 0.022284
Iteration 38000: Cost = 0.022224
Iteration 38100: Cost = 0.022164
Iteration 38200: Cost = 0.022104
Iteration 38300: Cost = 0.022045
Iteration 38400: Cost = 0.021986
Iteration 38500: Cost = 0.021927
Iteration 38600: Cost = 0.021869
Iteration 38700: Cost = 0.021811
Iteration 38800: Cost = 0.021753
Iteration 38900: Cost = 0.021696
Iteration 39000: Cost = 0.021638
Iteration 39100: Cost = 0.021581
Iteration 39200: Cost = 0.021525
Iteration 39300: Cost = 0.021469
Iteration 39400: Cost = 0.021413
Iteration 39500: Cost = 0.021357
Iteration 39600: Cost = 0.021301
Iteration 39700: Cost = 0.021246
Iteration 39800: Cost = 0.021191
Iteration 39900: Cost = 0.021137
Iteration 40000: Cost = 0.021083
Iteration 40100: Cost = 0.021029
Iteration 40200: Cost = 0.020975
Iteration 40300: Cost = 0.020921
Iteration 40400: Cost = 0.020868
Iteration 40500: Cost = 0.020815
Iteration 40600: Cost = 0.020762
Iteration 40700: Cost = 0.020710
Iteration 40800: Cost = 0.020658
Iteration 40900: Cost = 0.020606
Iteration 41000: Cost = 0.020554
Iteration 41100: Cost = 0.020503
Iteration 41200: Cost = 0.020452
Iteration 41300: Cost = 0.020401
Iteration 41400: Cost = 0.020350
Iteration 41500: Cost = 0.020300
Iteration 41600: Cost = 0.020250
Iteration 41700: Cost = 0.020200
Iteration 41800: Cost = 0.020150
Iteration 41900: Cost = 0.020101
Iteration 42000: Cost = 0.020052
Iteration 42100: Cost = 0.020003
Iteration 42200: Cost = 0.019954
Iteration 42300: Cost = 0.019906
Iteration 42400: Cost = 0.019857
Iteration 42500: Cost = 0.019809
Iteration 42600: Cost = 0.019762
Iteration 42700: Cost = 0.019714
Iteration 42800: Cost = 0.019667
Iteration 42900: Cost = 0.019620
Iteration 43000: Cost = 0.019573
Iteration 43100: Cost = 0.019526
Iteration 43200: Cost = 0.019480
Iteration 43300: Cost = 0.019434
Iteration 43400: Cost = 0.019388
Iteration 43500: Cost = 0.019342
Iteration 43600: Cost = 0.019296
Iteration 43700: Cost = 0.019251
Iteration 43800: Cost = 0.019206
Iteration 43900: Cost = 0.019161
Iteration 44000: Cost = 0.019116
Iteration 44100: Cost = 0.019072
Iteration 44200: Cost = 0.019027
Iteration 44300: Cost = 0.018983
Iteration 44400: Cost = 0.018939
Iteration 44500: Cost = 0.018896
Iteration 44600: Cost = 0.018852
Iteration 44700: Cost = 0.018809
Iteration 44800: Cost = 0.018766
Iteration 44900: Cost = 0.018723
Iteration 45000: Cost = 0.018680
Iteration 45100: Cost = 0.018638
Iteration 45200: Cost = 0.018595
Iteration 45300: Cost = 0.018553
Iteration 45400: Cost = 0.018511
Iteration 45500: Cost = 0.018469
Iteration 45600: Cost = 0.018428
Iteration 45700: Cost = 0.018386
Iteration 45800: Cost = 0.018345
Iteration 45900: Cost = 0.018304
Iteration 46000: Cost = 0.018263
Iteration 46100: Cost = 0.018223
Iteration 46200: Cost = 0.018182
Iteration 46300: Cost = 0.018142
Iteration 46400: Cost = 0.018102
Iteration 46500: Cost = 0.018062
Iteration 46600: Cost = 0.018022
Iteration 46700: Cost = 0.017982
Iteration 46800: Cost = 0.017943
Iteration 46900: Cost = 0.017904
Iteration 47000: Cost = 0.017864
Iteration 47100: Cost = 0.017826
Iteration 47200: Cost = 0.017787
Iteration 47300: Cost = 0.017748
Iteration 47400: Cost = 0.017710
Iteration 47500: Cost = 0.017671
Iteration 47600: Cost = 0.017633
Iteration 47700: Cost = 0.017595
Iteration 47800: Cost = 0.017558
Iteration 47900: Cost = 0.017520
Iteration 48000: Cost = 0.017483
Iteration 48100: Cost = 0.017445
Iteration 48200: Cost = 0.017408
Iteration 48300: Cost = 0.017371
Iteration 48400: Cost = 0.017334
Iteration 48500: Cost = 0.017298
Iteration 48600: Cost = 0.017261
Iteration 48700: Cost = 0.017225
Iteration 48800: Cost = 0.017189
Iteration 48900: Cost = 0.017152
Iteration 49000: Cost = 0.017117
Iteration 49100: Cost = 0.017081
Iteration 49200: Cost = 0.017045
Iteration 49300: Cost = 0.017010
Iteration 49400: Cost = 0.016974
Iteration 49500: Cost = 0.016939
Iteration 49600: Cost = 0.016904
Iteration 49700: Cost = 0.016869
Iteration 49800: Cost = 0.016834
Iteration 49900: Cost = 0.016800
Iteration 50000: Cost = 0.016765
Iteration 50100: Cost = 0.016731
Iteration 50200: Cost = 0.016697
Iteration 50300: Cost = 0.016663
Iteration 50400: Cost = 0.016629
Iteration 50500: Cost = 0.016595
Iteration 50600: Cost = 0.016561
Iteration 50700: Cost = 0.016528
Iteration 50800: Cost = 0.016495
Iteration 50900: Cost = 0.016461
Iteration 51000: Cost = 0.016428
Iteration 51100: Cost = 0.016395
Iteration 51200: Cost = 0.016362
Iteration 51300: Cost = 0.016330
Iteration 51400: Cost = 0.016297
Iteration 51500: Cost = 0.016265
Iteration 51600: Cost = 0.016232
Iteration 51700: Cost = 0.016200
Iteration 51800: Cost = 0.016168
Iteration 51900: Cost = 0.016136
Iteration 52000: Cost = 0.016104
Iteration 52100: Cost = 0.016073
Iteration 52200: Cost = 0.016041
Iteration 52300: Cost = 0.016010
Iteration 52400: Cost = 0.015978
Iteration 52500: Cost = 0.015947
Iteration 52600: Cost = 0.015916
Iteration 52700: Cost = 0.015885
Iteration 52800: Cost = 0.015854
Iteration 52900: Cost = 0.015823
Iteration 53000: Cost = 0.015793
Iteration 53100: Cost = 0.015762
Iteration 53200: Cost = 0.015732
Iteration 53300: Cost = 0.015702
Iteration 53400: Cost = 0.015672
Iteration 53500: Cost = 0.015641
Iteration 53600: Cost = 0.015612
Iteration 53700: Cost = 0.015582
Iteration 53800: Cost = 0.015552
Iteration 53900: Cost = 0.015522
Iteration 54000: Cost = 0.015493
Iteration 54100: Cost = 0.015464
Iteration 54200: Cost = 0.015434
Iteration 54300: Cost = 0.015405
Iteration 54400: Cost = 0.015376
Iteration 54500: Cost = 0.015347
Iteration 54600: Cost = 0.015318
Iteration 54700: Cost = 0.015290
Iteration 54800: Cost = 0.015261
Iteration 54900: Cost = 0.015233
Iteration 55000: Cost = 0.015204
Iteration 55100: Cost = 0.015176
Iteration 55200: Cost = 0.015148
Iteration 55300: Cost = 0.015120
Iteration 55400: Cost = 0.015092
Iteration 55500: Cost = 0.015064
Iteration 55600: Cost = 0.015036
Iteration 55700: Cost = 0.015008
Iteration 55800: Cost = 0.014981
Iteration 55900: Cost = 0.014953
Iteration 56000: Cost = 0.014926
Iteration 56100: Cost = 0.014899
Iteration 56200: Cost = 0.014872
Iteration 56300: Cost = 0.014845
Iteration 56400: Cost = 0.014818
Iteration 56500: Cost = 0.014791
Iteration 56600: Cost = 0.014764
Iteration 56700: Cost = 0.014737
Iteration 56800: Cost = 0.014711
Iteration 56900: Cost = 0.014684
Iteration 57000: Cost = 0.014658
Iteration 57100: Cost = 0.014632
Iteration 57200: Cost = 0.014605
Iteration 57300: Cost = 0.014579
Iteration 57400: Cost = 0.014553
Iteration 57500: Cost = 0.014527
Iteration 57600: Cost = 0.014501
Iteration 57700: Cost = 0.014476
Iteration 57800: Cost = 0.014450
Iteration 57900: Cost = 0.014424
Iteration 58000: Cost = 0.014399
Iteration 58100: Cost = 0.014374
Iteration 58200: Cost = 0.014348
Iteration 58300: Cost = 0.014323
Iteration 58400: Cost = 0.014298
Iteration 58500: Cost = 0.014273
Iteration 58600: Cost = 0.014248
Iteration 58700: Cost = 0.014223
Iteration 58800: Cost = 0.014198
Iteration 58900: Cost = 0.014174
Iteration 59000: Cost = 0.014149
Iteration 59100: Cost = 0.014124
Iteration 59200: Cost = 0.014100
Iteration 59300: Cost = 0.014076
Iteration 59400: Cost = 0.014051
Iteration 59500: Cost = 0.014027
Iteration 59600: Cost = 0.014003
Iteration 59700: Cost = 0.013979
Iteration 59800: Cost = 0.013955
Iteration 59900: Cost = 0.013931
Iteration 60000: Cost = 0.013907
Iteration 60100: Cost = 0.013884
Iteration 60200: Cost = 0.013860
Iteration 60300: Cost = 0.013837
Iteration 60400: Cost = 0.013813
Iteration 60500: Cost = 0.013790
Iteration 60600: Cost = 0.013766
Iteration 60700: Cost = 0.013743
Iteration 60800: Cost = 0.013720
Iteration 60900: Cost = 0.013697
Iteration 61000: Cost = 0.013674
Iteration 61100: Cost = 0.013651
Iteration 61200: Cost = 0.013628
Iteration 61300: Cost = 0.013606
Iteration 61400: Cost = 0.013583
Iteration 61500: Cost = 0.013560
Iteration 61600: Cost = 0.013538
Iteration 61700: Cost = 0.013515
Iteration 61800: Cost = 0.013493
Iteration 61900: Cost = 0.013471
Iteration 62000: Cost = 0.013448
Iteration 62100: Cost = 0.013426
Iteration 62200: Cost = 0.013404
Iteration 62300: Cost = 0.013382
Iteration 62400: Cost = 0.013360
Iteration 62500: Cost = 0.013338
Iteration 62600: Cost = 0.013316
Iteration 62700: Cost = 0.013295
Iteration 62800: Cost = 0.013273
Iteration 62900: Cost = 0.013251
Iteration 63000: Cost = 0.013230
Iteration 63100: Cost = 0.013208
Iteration 63200: Cost = 0.013187
Iteration 63300: Cost = 0.013166
Iteration 63400: Cost = 0.013144
Iteration 63500: Cost = 0.013123
Iteration 63600: Cost = 0.013102
Iteration 63700: Cost = 0.013081
Iteration 63800: Cost = 0.013060
Iteration 63900: Cost = 0.013039
Iteration 64000: Cost = 0.013018
Iteration 64100: Cost = 0.012997
Iteration 64200: Cost = 0.012977
Iteration 64300: Cost = 0.012956
Iteration 64400: Cost = 0.012935
Iteration 64500: Cost = 0.012915
Iteration 64600: Cost = 0.012895
Iteration 64700: Cost = 0.012874
Iteration 64800: Cost = 0.012854
Iteration 64900: Cost = 0.012834
Iteration 65000: Cost = 0.012813
Iteration 65100: Cost = 0.012793
Iteration 65200: Cost = 0.012773
Iteration 65300: Cost = 0.012753
Iteration 65400: Cost = 0.012733
Iteration 65500: Cost = 0.012713
Iteration 65600: Cost = 0.012693
Iteration 65700: Cost = 0.012674
Iteration 65800: Cost = 0.012654
Iteration 65900: Cost = 0.012634
Iteration 66000: Cost = 0.012615
Iteration 66100: Cost = 0.012595
Iteration 66200: Cost = 0.012576
Iteration 66300: Cost = 0.012556
Iteration 66400: Cost = 0.012537
Iteration 66500: Cost = 0.012518
Iteration 66600: Cost = 0.012498
Iteration 66700: Cost = 0.012479
Iteration 66800: Cost = 0.012460
Iteration 66900: Cost = 0.012441
Iteration 67000: Cost = 0.012422
Iteration 67100: Cost = 0.012403
Iteration 67200: Cost = 0.012384
Iteration 67300: Cost = 0.012365
Iteration 67400: Cost = 0.012347
Iteration 67500: Cost = 0.012328
Iteration 67600: Cost = 0.012309
Iteration 67700: Cost = 0.012291
Iteration 67800: Cost = 0.012272
Iteration 67900: Cost = 0.012254
Iteration 68000: Cost = 0.012235
Iteration 68100: Cost = 0.012217
Iteration 68200: Cost = 0.012199
Iteration 68300: Cost = 0.012180
Iteration 68400: Cost = 0.012162
Iteration 68500: Cost = 0.012144
Iteration 68600: Cost = 0.012126
Iteration 68700: Cost = 0.012108
Iteration 68800: Cost = 0.012090
Iteration 68900: Cost = 0.012072
Iteration 69000: Cost = 0.012054
Iteration 69100: Cost = 0.012036
Iteration 69200: Cost = 0.012018
Iteration 69300: Cost = 0.012001
Iteration 69400: Cost = 0.011983
Iteration 69500: Cost = 0.011965
Iteration 69600: Cost = 0.011948
Iteration 69700: Cost = 0.011930
Iteration 69800: Cost = 0.011913
Iteration 69900: Cost = 0.011895
Iteration 70000: Cost = 0.011878
Iteration 70100: Cost = 0.011861
Iteration 70200: Cost = 0.011843
Iteration 70300: Cost = 0.011826
Iteration 70400: Cost = 0.011809
Iteration 70500: Cost = 0.011792
Iteration 70600: Cost = 0.011775
Iteration 70700: Cost = 0.011758
Iteration 70800: Cost = 0.011741
Iteration 70900: Cost = 0.011724
Iteration 71000: Cost = 0.011707
Iteration 71100: Cost = 0.011690
Iteration 71200: Cost = 0.011673
Iteration 71300: Cost = 0.011656
Iteration 71400: Cost = 0.011640
Iteration 71500: Cost = 0.011623
Iteration 71600: Cost = 0.011607
Iteration 71700: Cost = 0.011590
Iteration 71800: Cost = 0.011574
Iteration 71900: Cost = 0.011557
Iteration 72000: Cost = 0.011541
Iteration 72100: Cost = 0.011524
Iteration 72200: Cost = 0.011508
Iteration 72300: Cost = 0.011492
Iteration 72400: Cost = 0.011475
Iteration 72500: Cost = 0.011459
Iteration 72600: Cost = 0.011443
Iteration 72700: Cost = 0.011427
Iteration 72800: Cost = 0.011411
Iteration 72900: Cost = 0.011395
Iteration 73000: Cost = 0.011379
Iteration 73100: Cost = 0.011363
Iteration 73200: Cost = 0.011347
Iteration 73300: Cost = 0.011331
Iteration 73400: Cost = 0.011316
Iteration 73500: Cost = 0.011300
Iteration 73600: Cost = 0.011284
Iteration 73700: Cost = 0.011269
Iteration 73800: Cost = 0.011253
Iteration 73900: Cost = 0.011237
Iteration 74000: Cost = 0.011222
Iteration 74100: Cost = 0.011206
Iteration 74200: Cost = 0.011191
Iteration 74300: Cost = 0.011176
Iteration 74400: Cost = 0.011160
Iteration 74500: Cost = 0.011145
Iteration 74600: Cost = 0.011130
Iteration 74700: Cost = 0.011114
Iteration 74800: Cost = 0.011099
Iteration 74900: Cost = 0.011084
Iteration 75000: Cost = 0.011069
Iteration 75100: Cost = 0.011054
Iteration 75200: Cost = 0.011039
Iteration 75300: Cost = 0.011024
Iteration 75400: Cost = 0.011009
Iteration 75500: Cost = 0.010994
Iteration 75600: Cost = 0.010979
Iteration 75700: Cost = 0.010964
Iteration 75800: Cost = 0.010950
Iteration 75900: Cost = 0.010935
Iteration 76000: Cost = 0.010920
Iteration 76100: Cost = 0.010906
Iteration 76200: Cost = 0.010891
Iteration 76300: Cost = 0.010876
Iteration 76400: Cost = 0.010862
Iteration 76500: Cost = 0.010847
Iteration 76600: Cost = 0.010833
Iteration 76700: Cost = 0.010818
Iteration 76800: Cost = 0.010804
Iteration 76900: Cost = 0.010790
Iteration 77000: Cost = 0.010775
Iteration 77100: Cost = 0.010761
Iteration 77200: Cost = 0.010747
Iteration 77300: Cost = 0.010733
Iteration 77400: Cost = 0.010719
Iteration 77500: Cost = 0.010704
Iteration 77600: Cost = 0.010690
Iteration 77700: Cost = 0.010676
Iteration 77800: Cost = 0.010662
Iteration 77900: Cost = 0.010648
Iteration 78000: Cost = 0.010634
Iteration 78100: Cost = 0.010620
Iteration 78200: Cost = 0.010607
Iteration 78300: Cost = 0.010593
Iteration 78400: Cost = 0.010579
Iteration 78500: Cost = 0.010565
Iteration 78600: Cost = 0.010551
Iteration 78700: Cost = 0.010538
Iteration 78800: Cost = 0.010524
Iteration 78900: Cost = 0.010510
Iteration 79000: Cost = 0.010497
Iteration 79100: Cost = 0.010483
Iteration 79200: Cost = 0.010470
Iteration 79300: Cost = 0.010456
Iteration 79400: Cost = 0.010443
Iteration 79500: Cost = 0.010429
Iteration 79600: Cost = 0.010416
Iteration 79700: Cost = 0.010403
Iteration 79800: Cost = 0.010389
Iteration 79900: Cost = 0.010376
Iteration 80000: Cost = 0.010363
Iteration 80100: Cost = 0.010350
Iteration 80200: Cost = 0.010337
Iteration 80300: Cost = 0.010323
Iteration 80400: Cost = 0.010310
Iteration 80500: Cost = 0.010297
Iteration 80600: Cost = 0.010284
Iteration 80700: Cost = 0.010271
Iteration 80800: Cost = 0.010258
Iteration 80900: Cost = 0.010245
Iteration 81000: Cost = 0.010232
Iteration 81100: Cost = 0.010219
Iteration 81200: Cost = 0.010207
Iteration 81300: Cost = 0.010194
Iteration 81400: Cost = 0.010181
Iteration 81500: Cost = 0.010168
Iteration 81600: Cost = 0.010155
Iteration 81700: Cost = 0.010143
Iteration 81800: Cost = 0.010130
Iteration 81900: Cost = 0.010117
Iteration 82000: Cost = 0.010105
Iteration 82100: Cost = 0.010092
Iteration 82200: Cost = 0.010080
Iteration 82300: Cost = 0.010067
Iteration 82400: Cost = 0.010055
Iteration 82500: Cost = 0.010042
Iteration 82600: Cost = 0.010030
Iteration 82700: Cost = 0.010018
Iteration 82800: Cost = 0.010005
Iteration 82900: Cost = 0.009993
Iteration 83000: Cost = 0.009981
Iteration 83100: Cost = 0.009968
Iteration 83200: Cost = 0.009956
Iteration 83300: Cost = 0.009944
Iteration 83400: Cost = 0.009932
Iteration 83500: Cost = 0.009920
Iteration 83600: Cost = 0.009908
Iteration 83700: Cost = 0.009895
Iteration 83800: Cost = 0.009883
Iteration 83900: Cost = 0.009871
Iteration 84000: Cost = 0.009859
Iteration 84100: Cost = 0.009847
Iteration 84200: Cost = 0.009835
Iteration 84300: Cost = 0.009824
Iteration 84400: Cost = 0.009812
Iteration 84500: Cost = 0.009800
Iteration 84600: Cost = 0.009788
Iteration 84700: Cost = 0.009776
Iteration 84800: Cost = 0.009764
Iteration 84900: Cost = 0.009753
Iteration 85000: Cost = 0.009741
Iteration 85100: Cost = 0.009729
Iteration 85200: Cost = 0.009718
Iteration 85300: Cost = 0.009706
Iteration 85400: Cost = 0.009694
Iteration 85500: Cost = 0.009683
Iteration 85600: Cost = 0.009671
Iteration 85700: Cost = 0.009660
Iteration 85800: Cost = 0.009648
Iteration 85900: Cost = 0.009637
Iteration 86000: Cost = 0.009625
Iteration 86100: Cost = 0.009614
Iteration 86200: Cost = 0.009603
Iteration 86300: Cost = 0.009591
Iteration 86400: Cost = 0.009580
Iteration 86500: Cost = 0.009569
Iteration 86600: Cost = 0.009557
Iteration 86700: Cost = 0.009546
Iteration 86800: Cost = 0.009535
Iteration 86900: Cost = 0.009524
Iteration 87000: Cost = 0.009513
Iteration 87100: Cost = 0.009501
Iteration 87200: Cost = 0.009490
Iteration 87300: Cost = 0.009479
Iteration 87400: Cost = 0.009468
Iteration 87500: Cost = 0.009457
Iteration 87600: Cost = 0.009446
Iteration 87700: Cost = 0.009435
Iteration 87800: Cost = 0.009424
Iteration 87900: Cost = 0.009413
Iteration 88000: Cost = 0.009402
Iteration 88100: Cost = 0.009391
Iteration 88200: Cost = 0.009381
Iteration 88300: Cost = 0.009370
Iteration 88400: Cost = 0.009359
Iteration 88500: Cost = 0.009348
Iteration 88600: Cost = 0.009337
Iteration 88700: Cost = 0.009327
Iteration 88800: Cost = 0.009316
Iteration 88900: Cost = 0.009305
Iteration 89000: Cost = 0.009295
Iteration 89100: Cost = 0.009284
Iteration 89200: Cost = 0.009273
Iteration 89300: Cost = 0.009263
Iteration 89400: Cost = 0.009252
Iteration 89500: Cost = 0.009242
Iteration 89600: Cost = 0.009231
Iteration 89700: Cost = 0.009221
Iteration 89800: Cost = 0.009210
Iteration 89900: Cost = 0.009200
Iteration 90000: Cost = 0.009189
Iteration 90100: Cost = 0.009179
Iteration 90200: Cost = 0.009168
Iteration 90300: Cost = 0.009158
Iteration 90400: Cost = 0.009148
Iteration 90500: Cost = 0.009138
Iteration 90600: Cost = 0.009127
Iteration 90700: Cost = 0.009117
Iteration 90800: Cost = 0.009107
Iteration 90900: Cost = 0.009096
Iteration 91000: Cost = 0.009086
Iteration 91100: Cost = 0.009076
Iteration 91200: Cost = 0.009066
Iteration 91300: Cost = 0.009056
Iteration 91400: Cost = 0.009046
Iteration 91500: Cost = 0.009036
Iteration 91600: Cost = 0.009026
Iteration 91700: Cost = 0.009016
Iteration 91800: Cost = 0.009006
Iteration 91900: Cost = 0.008996
Iteration 92000: Cost = 0.008986
Iteration 92100: Cost = 0.008976
Iteration 92200: Cost = 0.008966
Iteration 92300: Cost = 0.008956
Iteration 92400: Cost = 0.008946
Iteration 92500: Cost = 0.008936
Iteration 92600: Cost = 0.008926
Iteration 92700: Cost = 0.008916
Iteration 92800: Cost = 0.008907
Iteration 92900: Cost = 0.008897
Iteration 93000: Cost = 0.008887
Iteration 93100: Cost = 0.008877
Iteration 93200: Cost = 0.008868
Iteration 93300: Cost = 0.008858
Iteration 93400: Cost = 0.008848
Iteration 93500: Cost = 0.008839
Iteration 93600: Cost = 0.008829
Iteration 93700: Cost = 0.008819
Iteration 93800: Cost = 0.008810
Iteration 93900: Cost = 0.008800
Iteration 94000: Cost = 0.008791
Iteration 94100: Cost = 0.008781
Iteration 94200: Cost = 0.008772
Iteration 94300: Cost = 0.008762
Iteration 94400: Cost = 0.008753
Iteration 94500: Cost = 0.008743
Iteration 94600: Cost = 0.008734
Iteration 94700: Cost = 0.008725
Iteration 94800: Cost = 0.008715
Iteration 94900: Cost = 0.008706
Iteration 95000: Cost = 0.008696
Iteration 95100: Cost = 0.008687
Iteration 95200: Cost = 0.008678
Iteration 95300: Cost = 0.008669
Iteration 95400: Cost = 0.008659
Iteration 95500: Cost = 0.008650
Iteration 95600: Cost = 0.008641
Iteration 95700: Cost = 0.008632
Iteration 95800: Cost = 0.008622
Iteration 95900: Cost = 0.008613
Iteration 96000: Cost = 0.008604
Iteration 96100: Cost = 0.008595
Iteration 96200: Cost = 0.008586
Iteration 96300: Cost = 0.008577
Iteration 96400: Cost = 0.008568
Iteration 96500: Cost = 0.008559
Iteration 96600: Cost = 0.008550
Iteration 96700: Cost = 0.008541
Iteration 96800: Cost = 0.008532
Iteration 96900: Cost = 0.008523
Iteration 97000: Cost = 0.008514
Iteration 97100: Cost = 0.008505
Iteration 97200: Cost = 0.008496
Iteration 97300: Cost = 0.008487
Iteration 97400: Cost = 0.008478
Iteration 97500: Cost = 0.008469
Iteration 97600: Cost = 0.008460
Iteration 97700: Cost = 0.008452
Iteration 97800: Cost = 0.008443
Iteration 97900: Cost = 0.008434
Iteration 98000: Cost = 0.008425
Iteration 98100: Cost = 0.008416
Iteration 98200: Cost = 0.008408
Iteration 98300: Cost = 0.008399
Iteration 98400: Cost = 0.008390
Iteration 98500: Cost = 0.008382
Iteration 98600: Cost = 0.008373
Iteration 98700: Cost = 0.008364
Iteration 98800: Cost = 0.008356
Iteration 98900: Cost = 0.008347
Iteration 99000: Cost = 0.008339
Iteration 99100: Cost = 0.008330
Iteration 99200: Cost = 0.008321
Iteration 99300: Cost = 0.008313
Iteration 99400: Cost = 0.008304
Iteration 99500: Cost = 0.008296
Iteration 99600: Cost = 0.008287
Iteration 99700: Cost = 0.008279
Iteration 99800: Cost = 0.008270
Iteration 99900: Cost = 0.008262
Iteration 99999: Cost = 0.008254

Final parameters:
w: [38.51304248 18.83386869], b: -2.8176836626325836
Final cost: 0.008254
```

```
# Simple assertion test for gradient_descent
def test_gradient_descent():
    X = np.array([[0.1, 0.2], [-0.1, 0.1]])  # Shape (2, 2)
    y = np.array([1, 0])                     # Shape (2,)
    w = np.zeros(X.shape[1])                 # Shape (2,)
    b = 0.0                                  # Scalar
    alpha = 0.1                              # Learning rate
    n_iter = 100                             # Number of iterations
    # Run gradient descent
    w_out, b_out, cost_history, _ = gradient_descent(X, y, w, b,
alpha, n_iter, show_cost=False, show_params=False)

    # Assertions
    assert len(cost_history) == n_iter, "Cost history length does not
match the number of iterations"
    assert w_out.shape == w.shape, "Shape of output weights does not
match the initial weights"
    assert isinstance(b_out, float), "Bias output is not a float"
    assert cost_history[-1] < cost_history[0], "Cost did not decrease
over iterations"

    print("All tests passed!")

# Run the test
test_gradient_descent()
```

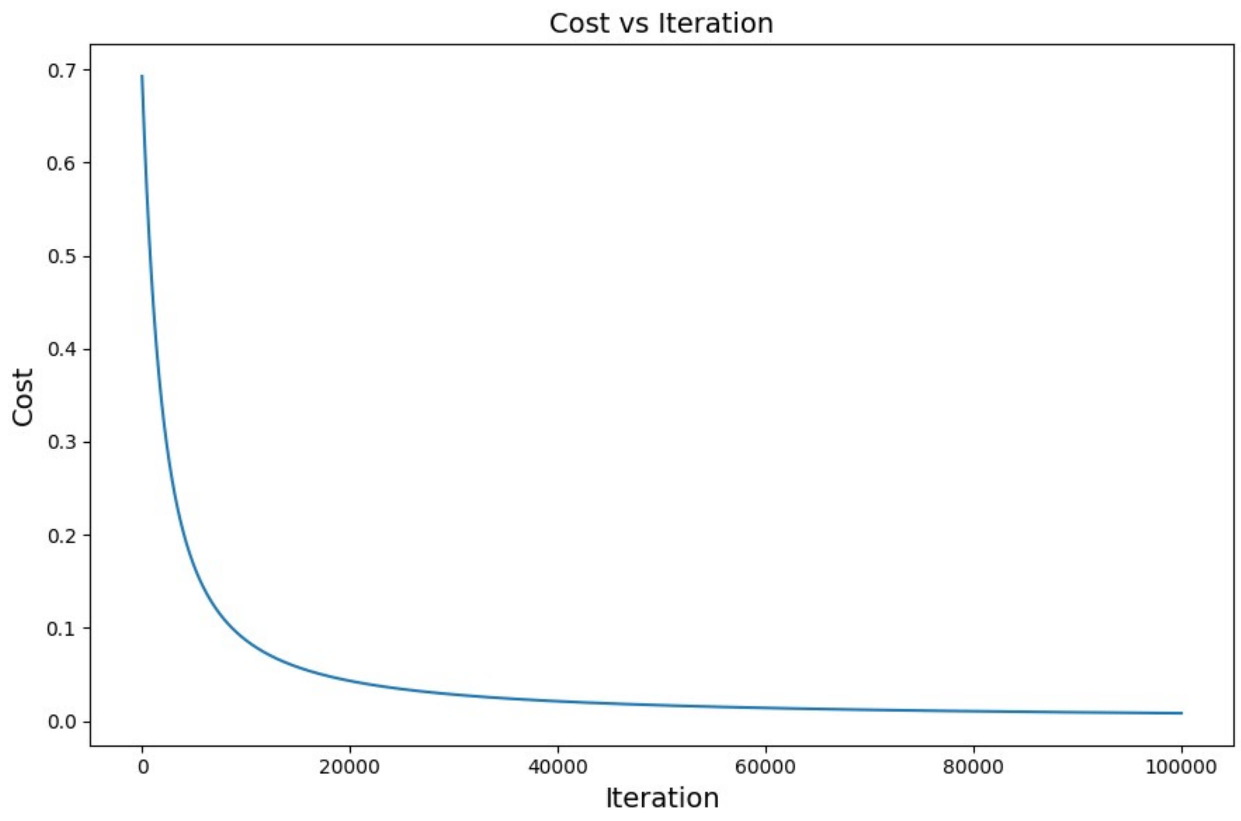
```
All tests passed!
```

Visualizing Convergence of Cost During Gradient Descent:

This plot tracks how the cost decreases over iterations, providing insight into the convergence of the gradient descent algorithm.

It ensures the optimization is progressing as expected, helps detect issues like divergence, and aids in tuning hyperparameters effectively.

```
# Plotting cost over iteration
plt.figure(figsize = (9, 6))
plt.plot(cost_history)
plt.xlabel("Iteration", fontsize = 14)
plt.ylabel("Cost", fontsize = 14)
plt.title("Cost vs Iteration", fontsize = 14)
plt.tight_layout()
plt.show()
```



Decision Function

In Decsion function we perform following:

- Prediction:
  - Utilizing our trained weights and biases we first find the probabilities values y_probab for the x_test. Then, Y_probabiliy value is transformed to discrete class value using Decision boubdry. ___
- Decision Boundry:
  - We first calculate the y_prediction as probabilities value, which is then converted to discrete class by using a threshold value. For instance, we take the threshold to be 0.5,then we classify the observation to class 1 and to class 0 otherwise.

```
import numpy as np

def prediction(X, w, b, threshold=0.5):
    """
    Predicts binary outcomes for given input features based on
logistic regression parameters.

    Arguments:
      X (ndarray, shape (n,d)): Array of test independent variables
(features) with n samples and d features.
```

```
      w (ndarray, shape (d,)): Array of weights learned via gradient
descent.
      b (float): Bias learned via gradient descent.
      threshold (float, optional): Classification threshold for
predicting class labels. Default is 0.5.

    Returns:
      y_pred (ndarray, shape (n,)): Array of predicted dependent
variable (binary class labels: 0 or 1).
    """
    # Compute the predicted probabilities using the logistic function
    y_test_prob = logistic_function(np.dot(X, w) + b)

    # Classify based on the threshold
    y_pred = (y_test_prob >= threshold).astype(int)

    return y_pred

def test_prediction():
    X_test = np.array([[0.5, 1.0], [1.5, -0.5], [-0.5, -1.0]])  #
Shape (3, 2)
    w_test = np.array([1.0, -1.0])                              #
Shape (2,)
    b_test = 0.0                                                #
Scalar bias
    threshold = 0.5                                             #
Default threshold

    # Updated expected output
    expected_output = np.array([0, 1, 1])

    # Call the prediction function
    y_pred = prediction(X_test, w_test, b_test, threshold)

    # Assert that the output matches the expected output
    assert np.array_equal(y_pred, expected_output), f"Expected
{expected_output}, but got {y_pred}"

    print("Test passed!")

test_prediction()

Test passed!
```

Evaluation of the Classifier.

```
# Evaluation Function: Computes confusion matrix, precision, recall,
and F1-score
def evaluate_classification(y_true, y_pred):
    TP = np.sum((y_true == 1) & (y_pred == 1))  # True Positives
    TN = np.sum((y_true == 0) & (y_pred == 0))  # True Negatives
    FP = np.sum((y_true == 0) & (y_pred == 1))  # False Positives
    FN = np.sum((y_true == 1) & (y_pred == 0))  # False Negatives

    # Confusion matrix
    confusion_matrix = np.array([[TN, FP],
                                  [FN, TP]])

    # Precision, recall, and F1-score with safe handling of zero
division
    precision = TP / (TP + FP) if (TP + FP) > 0 else 0.0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0.0
    f1_score = (2 * precision * recall) / (precision + recall) if
(precision + recall) > 0 else 0.0

    # Ensure metrics are floats
    precision = float(precision)
    recall = float(recall)
    f1_score = float(f1_score)

    # Return metrics as a dictionary
    return confusion_matrix, precision, recall, f1_score
```

Putting Helper Function to Action.

Compiling and Training A sigmoid regression on Dataset.

Some Basic Data operations, Loading, Analysis and Cleaning:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
```

Necessary Imports and Loading of the Dataset:

```
# Load dataset
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-
indians-diabetes.data.csv"
```

```
columns = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness',
'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome']
data_pima_diabetes = pd.read_csv(url, names=columns)
```

Some Basic Data Cleaning:

```
# Data cleaning
columns_to_clean = ['Glucose', 'BloodPressure', 'SkinThickness',
'Insulin', 'BMI']
data_pima_diabetes[columns_to_clean] =
data_pima_diabetes[columns_to_clean].replace(0, np.nan)
data_pima_diabetes.fillna(data_pima_diabetes.median(), inplace=True)

data_pima_diabetes.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 101 entries, 0 to 100
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   HouseAge    101 non-null    object
 1   HouseFloor  101 non-null    object
 2   HouseArea   101 non-null    object
 3   HousePrice  101 non-null    object
dtypes: object(4)
memory usage: 3.3+ KB
```

Summary Statistics:

```
data_pima_diabetes.describe()
```

```
{"summary":"{\n  \"name\": \"data_pima_diabetes\",\n  \"rows\": 4,\n
\"fields\": [\n    {\n      \"column\": \"HouseAge\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 4,\n        \"samples\": [\n          62,\n
\"5\",\n          \"101\"\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"HouseFloor\",\n      \"properties\": {\n
\"dtype\": \"string\",\n        \"num_unique_values\": 4,\n
\"samples\": [\n          6,\n          \"27\",\n          \"101\"\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"HouseArea\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 3,\n        \"samples\": [\n          \"101\",\
n          \"HouseArea\",\n          \"1\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"HousePrice\",\n
\"properties\": {\n        \"dtype\": \"string\",\n
\"num_unique_values\": 3,\n        \"samples\": [\n          \"101\",\
n          \"HousePrice\",\n          \"1\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe"}
```

Train Test Split and Standard Scaling of the Dataset:

Train - Test Split:

```
# Train-test split
X = data_pima_diabetes.drop(columns=['Outcome']).values
y = data_pima_diabetes['Outcome'].values
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42, stratify=y)
```

Standarize the Input Dataset:

```
# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Initialize the parameters and Hyper-parameters:

```
# Initialize parameters
w = np.zeros(X_train_scaled.shape[1])
b = 0.0
alpha = 0.1
n_iter = 1000
```

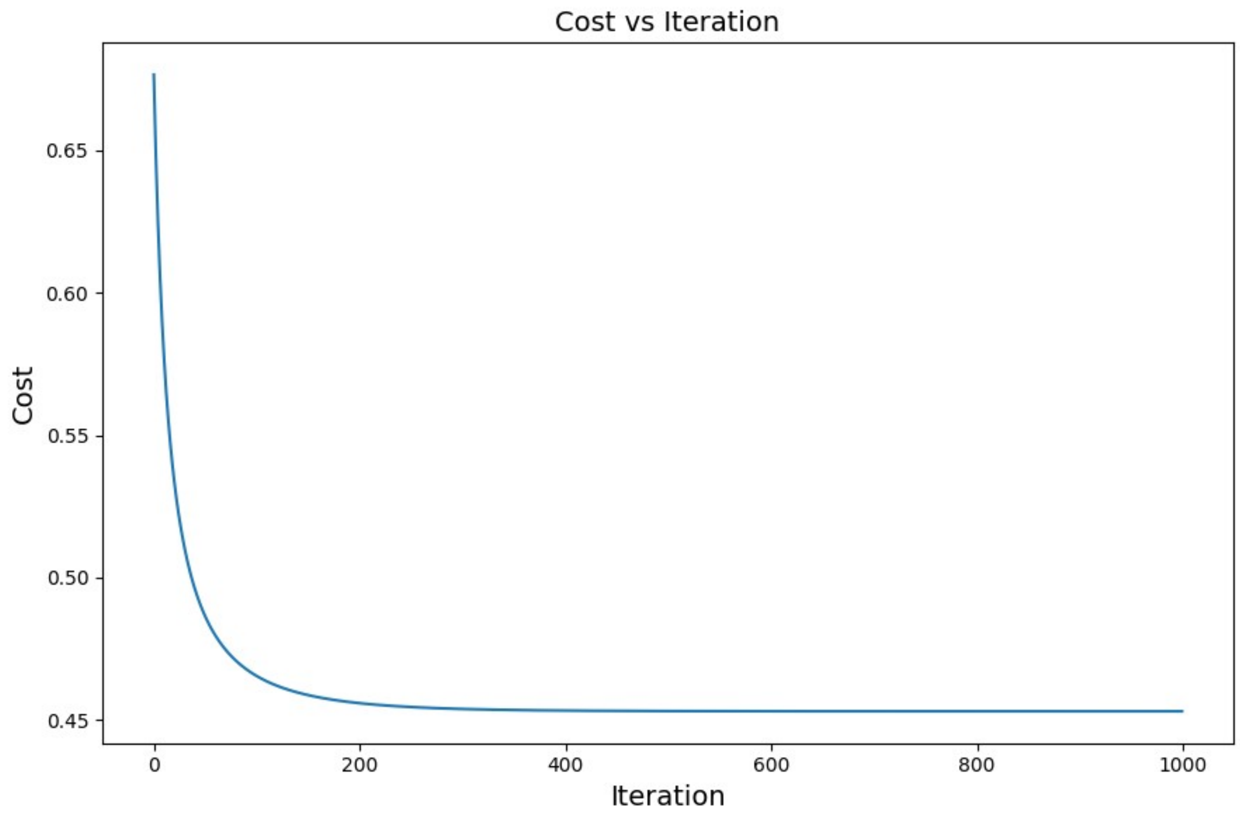
Start the Training:

```
# Train model
print("\nTraining Logistic Regression Model:")
w, b, cost_history,params_history = gradient_descent(X_train_scaled,
y_train, w, b, alpha, n_iter, show_cost=True, show_params=False)
```

```
Training Logistic Regression Model:
Iteration 0: Cost = 0.676575
Iteration 100: Cost = 0.465441
Iteration 200: Cost = 0.455913
Iteration 300: Cost = 0.453874
Iteration 400: Cost = 0.453316
Iteration 500: Cost = 0.453148
Iteration 600: Cost = 0.453096
Iteration 700: Cost = 0.453079
Iteration 800: Cost = 0.453074
Iteration 900: Cost = 0.453072
Iteration 999: Cost = 0.453071
```

Observe the Cost History:

```
# Plot cost history
plt.figure(figsize=(9, 6))
plt.plot(cost_history)
plt.xlabel("Iteration", fontsize=14)
plt.ylabel("Cost", fontsize=14)
plt.title("Cost vs Iteration", fontsize=14)
plt.tight_layout()
plt.show()
```



Did the model overfitt?

```
# Test model
y_train_pred = prediction(X_train_scaled, w, b)
y_test_pred = prediction(X_test_scaled, w, b)

 # Evaluate train and test performance
train_cost = costfunction_logreg(X_train_scaled, y_train, w, b)
test_cost = costfunction_logreg(X_test_scaled, y_test, w, b)
print(f"\nTrain Loss (Cost): {train_cost:.4f}")
print(f"Test Loss (Cost): {test_cost:.4f}")
```

```
Train Loss (Cost): 0.4531
Test Loss (Cost): 0.5146
```

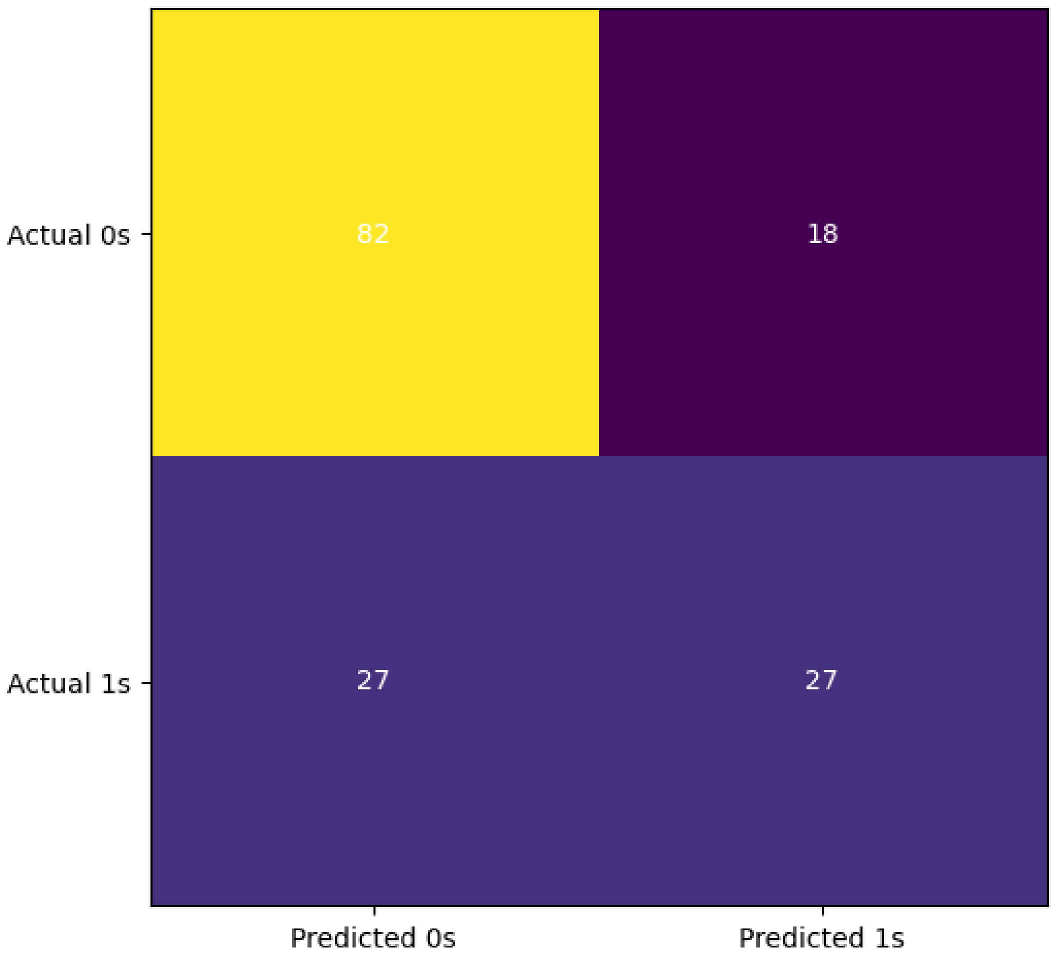
How well my model did?

```
# Accuracy on test data
test_accuracy = np.mean(y_test_pred == y_test) * 100
print(f"\nTest Accuracy: {test_accuracy:.2f}%")

# Evaluation
confusion_matrix, precision, recall, f1_score =
evaluate_classification(y_test, y_test_pred)
print(f"\nConfusion Matrix:\n{confusion_matrix}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-Score: {f1_score:.2f}")
```

```
Test Accuracy: 70.78%

Confusion Matrix:
[[82 18]
 [27 27]]
Precision: 0.60
Recall: 0.50
F1-Score: 0.55
```

Visualize the Confusion Matrix - Optional.

```
# Visualizing Confusion Matrix
fig, ax = plt.subplots(figsize=(6, 6))
ax.imshow(confusion_matrix)
ax.grid(False)
ax.xaxis.set(ticks=(0, 1), ticklabels=('Predicted 0s', 'Predicted
1s'))
ax.yaxis.set(ticks=(0, 1), ticklabels=('Actual 0s', 'Actual 1s'))
ax.set_ylim(1.5, -0.5)
for i in range(2):
    for j in range(2):
        ax.text(j, i, confusion_matrix[i, j], ha='center',
va='center', color='white')
plt.show()
```


Actual 0s
Actual 1s
Predicted 0s
Predicted 1s
82
18
27
27