Lab 6: Gradient Descent for Logistic Regression (02/10/2024)

Matthew Loh

Table of contents


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

```
X_train = np.array([[0.5, 1.5], [1, 1], [1.5, 0.5], [3, 0.5], [2, 2], [1, 2.5]])
y_train = np.array([0, 0, 0, 1, 1, 1])
```

```
def sigmoid(x: np.ndarray) -> np.ndarray:
    """
    Compute the sigmoid function
    Args:
      x (ndarray): input to the function
    Returns:
      ndarray: output of the sigmoid function
    """
    return 1 / (1 + np.exp(-x))
```

```
def compute_gradient_logistic(
    X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float
) -> None:
    """
    Compute the gradient for linear regression
    Args:
      X (ndarray (m, n)): Data, m examples with n features
      y (ndarray (m, )): target values
```

```
      w (ndarray(n, )): model parameters
      b (scalar): model parameter
    """
    m, n = X.shape
    dj_dw = np.zeros((n,))
    dj_db = 0.0

    for i in range(m):
        f_wb_i = sigmoid(np.dot(X[i], w) + b)
        err_i = f_wb_i - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err_i * X[i, j]
        dj_db = dj_db + err_i

    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_db, dj_dw
```

```
X_tmp = np.array([[0.5, 1.5], [1, 1], [1.5, 0.5], [3, 0.5], [2, 2], [1, 2.5]])
y_tmp = np.array([0, 0, 0, 1, 1, 1])
w_tmp = np.array([2.0, 3.0])
b_tmp = 1.0

dj_db_tmp, dj_dw_tmp = compute_gradient_logistic(X_tmp, y_tmp, w_tmp, b_tmp)
print(f"dj_db: {dj_db_tmp}")
print(f"dj_dw: {dj_dw_tmp.tolist()}")
```

```
dj_db: 0.49861806546328574
dj_dw: [0.498333393278696, 0.49883942983996693]
```

```
def compute_cost_logistic(X, y, w, b):
    """
    Compute cost
    Args:
      X (ndarray (m, n)): Data, m examples with n features
      y (ndarray (m, )): target values
      w (ndarray (n, )): model parameters
      b (scalar): model parameter
    Returns:
      cost (scalar): cost
```

```
    """
    m = X.shape[0]
    cost = 0.0
    for i in range(m):
        f_wb_i = sigmoid(np.dot(X[i], w) + b)
        cost = cost + y[i] * np.log(f_wb_i) + (1 - y[i]) * np.log(1 - f_wb_i)
    cost = -cost / m
    return cost
```

```
import copy
import math


def gradient_descent(X, y, w_in, b_in, alpha, num_iters):
    """
    Perform batch gradient descent

    Args:
      X (ndarray (m, n)): Data, m examples with n features
      y (ndarray (m, )): target values
      w_in (ndarray(n, )): Initial values of model parameters
      b_in (scalar): Initial values of model parameter
      alpha (float): Learning rate
      num_iters (int): number of iterations to run gradient descent
    Returns:
      w (ndarray(n, )): Updated values of parameters
      b (scalar): Updated value of parameter
      J_history (list): History of cost function values
    """

    J_history = []
    w = copy.deepcopy(w_in)
    b = b_in

    for i in range(num_iters):
        dj_db, dj_dw = compute_gradient_logistic(X, y, w, b)
        w = w - alpha * dj_dw
        b = b - alpha * dj_db
        # Save cost J at each iteration
        if i < 100000:
            J_history.append(compute_cost_logistic(X, y, w, b))
        # Print cost every at intervals 10 times or as many iterations if < 10
```

```
        if i % math.ceil(num_iters / 10) == 0:
            print(f"Iteration {i:4d}: Cost = {J_history[-1]}")
    return w, b, J_history
```

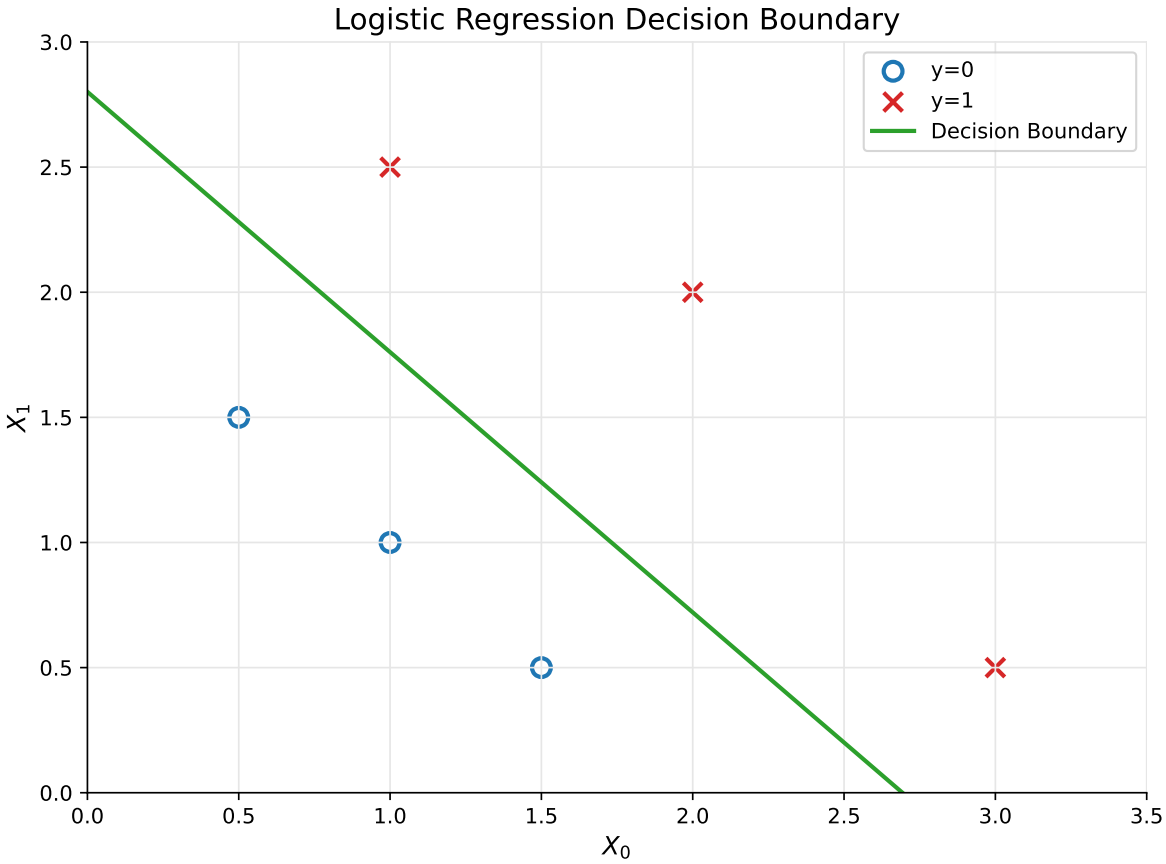
```
w_tmp = np.zeros_like(X_train[0])
b_tmp = 0.0
alph = 0.1
iters = 10000

w_out, b_out, J_hist = gradient_descent(X_train, y_train, w_tmp, b_tmp, alph, iters)
print(f"\nupdated parameters: w = {w_out}, b = {b_out}")
```

```
Iteration    0: Cost = 0.684610468560574
Iteration 1000: Cost = 0.1590977666870456
Iteration 2000: Cost = 0.08460064176930081
Iteration 3000: Cost = 0.05705327279402531
Iteration 4000: Cost = 0.042907594216820076
Iteration 5000: Cost = 0.034338477298845684
Iteration 6000: Cost = 0.028603798022120097
Iteration 7000: Cost = 0.024501569608793
Iteration 8000: Cost = 0.02142370332569295
Iteration 9000: Cost = 0.019030137124109114

updated parameters: w = [5.28123029 5.07815608], b = -14.222409982019837
```

Plot the Decision Boundary

```
plt.style.use("default")
plt.rcParams["figure.facecolor"] = "white"
plt.rcParams["axes.facecolor"] = "white"
plt.rcParams["axes.grid"] = True
plt.rcParams["grid.color"] = "#E6E6E6"
plt.rcParams["axes.spines.top"] = False
plt.rcParams["axes.spines.right"] = False

plt.figure(figsize=(8, 6))

plt.scatter(
    X_train[y_train == 0][:, 0],
```

```
    X_train[y_train == 0][:, 1],
    marker="o",
    s=80,
    facecolors="none",
    edgecolors="#1f77b4",
    linewidth=2,
    label="y=0",
)
plt.scatter(
    X_train[y_train == 1][:, 0],
    X_train[y_train == 1][:, 1],
    marker="x",
    s=80,
    c="#d62728",
    linewidth=2,
    label="y=1",
)

x0_min, x0_max = X_train[:, 0].min() - 0.5, X_train[:, 0].max() + 0.5
x1 = np.array([x0_min, x0_max])
x2 = -(b_out + w_out[0] * x1) / w_out[1]
plt.plot(x1, x2, color="#2ca02c", linestyle="-", linewidth=2, label="Decision Boundary")

plt.xlabel("$X_0$", fontsize=12)
plt.ylabel("$X_1$", fontsize=12)
plt.title("Logistic Regression Decision Boundary", fontsize=14)

plt.xlim(x0_min, x0_max)
y_min, y_max = X_train[:, 1].min() - 0.5, X_train[:, 1].max() + 0.5
plt.ylim(y_min, y_max)

plt.legend(fontsize=10)

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


Logistic Regression Decision Boundary
y=0
y=1
Decision Boundary
X_0
X_1
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5