Gradient Descent for Linear Regression

You will:

- Update gradient descent for logistic regression.
- See gradient descent on a familiar data set.

Logistic Gradient Descent

Recall the gradient descent algorithm utilizes the gradient calculation:

$$\begin{align*}
&\text{repeat until convergence: } \lbrace \\
&\quad w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \quad \text{for j := 0..n-1} \tag{1} \\
&\quad b = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \\
&\rbrace
\end{align*}$$

Where each iteration performs simultaneous updates on w_j for all j, where

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_j^{(i)} \tag{2}$$

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \tag{3}$$

- m is the number of training examples in the data set
- $f_{\mathbf{w},b}(x^{(i)})$ is the model's prediction, while $y^{(i)}$ is the target
- For a logistic regression model
  $z = \mathbf{w} \cdot \mathbf{x} + b$
  $f_{\mathbf{w},b}(x) = g(z)$
  where $g(z)$ is the sigmoid function:
  $g(z) = \frac{1}{1+e^{-z}}$

Calculating the Gradient, Code Description

Implements equation (2),(3) above for all w_j and b. There are many ways to implement this. Outlined below is this:

- initialize variables to accumulate `dj_dw` and `dj_db`
- for each example
  - calculate the error for that example $g(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) - \mathbf{y}^{(i)}$
  - for each input value $x_j^{(i)}$ in this example,
    - multiply the error by the input $x_j^{(i)}$, and add to the corresponding element of `dj_dw`. (equation 2 above)
  - add the error to `dj_db` (equation 3 above)
- divide `dj_db` and `dj_dw` by total number of examples (m)
- note that $\mathbf{x}^{(i)}$ in numpy `X[i,:]` or `X[i]` and $x_j^{(i)}$ is `X[i,j]`

Python - imports

```python
import numpy as np
import matplotlib.pyplot as plt
import math, copy
```

✓ 0.5s Python

Python - imports

```python
def sigmoid(z):
    """
    compute sigmoid

    Args:
        z (scalar):  logistic function, f_wb

    Return:
        g (scalar):  sigmoid of f_wb
    """
    return 1/(1 + np.exp(-z))
```

✓ 0.0s Python

C - imports

```c
double sigmoid(double z){
    /*
    compute sigmoid

    Args:
        z (scalar):  logistic function, f_wb

    Return:
        g (scalar):  sigmoid of f_wb
    */

    return 1 / (1 + exp(-z));
}
```

Python - gradient

```python
def compute_gradient_logistic(X, y, w, b):
    """
    Computes the gradient for linear regression

    Args:
        X (adarray (m,n)):  Data, m examples with n features
        y (ndarray (m, )):  target values
        w (ndarray (n, )):  model parameters
        b (scalar)       :  model parameter

    Returns:
        dj_dw (ndarray (n, )):  The gradient of the cost w.r.t the parameters w.
        dj_db (scalar)       :  The gradient of the cost w.r.t the parameter b.
    """

    m,n = X.shape
    dj_dw = np.zeros((n,))              #(n,)
    dj_db = 0.

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

    return dj_dw,dj_db
```

✓ 0.0s Python

C - gradient

```c
struct caches gradient_descent(float x_in[][n], int y_in[], double **w_in, double b_in, float alpha, int num_iters){

    /*
     Performs 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 (scalar) : number of iterations to run gradient descent

    Returns:
      w (ndarray (n,))   : Updated values of parameters
      b (scalar)         : Updated value of parameter
    */

    struct caches cache;
    struct Grads grads;

    cache.J_history = (double *)calloc(num_iters, sizeof(double));
    cache.w = deepcopy(w_in);                   //avoid modifying global w within function
    cache.b=b_in;

    for (int i = 0; i < num_iters; i++)
    {
        //calculate gradient and update the parameters
        grads=compute_gradient_logistic(x_in, y_in, cache.w, cache.b);

        //update parameters using w, b, alpha and gradient
        for (int j = 0; j < n; j++)
        {
            cache.w[j][0] = cache.w[j][0] - (grads.djdw[j] * alpha);
        }
        cache.b = cache.b - (grads.djdb * alpha);

        double J=compute_cost_logistic(x_in, y_in, cache.w, cache.b);

        cache.J_history[i] = J;

        if (i % (num_iters / 10)==0)
        {
        printf("Iteration %d \t Cost %f\n", i, cache.J_history[i]);
        }

    }

    return cache;
}
```

Python - cost

```python
def compute_cost_logistic(X, y, w, b):
    """
    Compute cost
    Args:
        X (ndarray (m,n)):  input data
        y (ndarray (m,)):   output data
        w (ndarray (n,)):   model parametrs
        b (scalar):         model parameter

    Return:
        J (sclar):  Cost of logistic model
    """

    J = 0.0
    f_wb_i = 0.0
    m = X.shape[0]

    for i in range(m):
        f_wb_i = sigmoid(np.dot(X[i], w) + b)                       #(m,): scalar
        J += -y[i]*np.log(f_wb_i) - (1 - y[i])*np.log(1 - f_wb_i)
    J /= m

    return J
```

✓ 0.0s Python

C - cost

```c
double compute_cost_logistic(float x[][n], int y[], double **w, double b){

    /*
    Compute cost
    Args:
        X (ndarray (m,n)):  input data
        y (ndarray (m,)):   output data
        w (ndarray (n,)):   model parametrs
        b (scalar):         model parameter

    Return:
        J (sclar):  Cost of logistic model
    */

    double J=0;
    double f_x;
    double f_wb;

    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < 1; j++)
        {
            f_wb = 0;
            f_x=0;
            for (int k = 0; k < n; k++)
            {
                f_x += x[i][k] * w[k][j];
            }
            f_x +=b;
            f_wb = sigmoid(f_x);
            J += (-y[i]*log(f_wb)) - ((1 - y[i])*log(1 - f_wb));

        }

    }
    J /= m;

    return J;
}
```

Python - Gradient Descent

```python
def gradient_descent(X, y, w_in, b_in, alpha, num_iter):
    """
    Performs 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 (scalar) : number of iterations to run gradient descent

    Returns:
      w (ndarray (n,))   : Updated values of parameters
      b (scalar)         : Updated value of parameter
    """

    #An array to store cost J and w's each iteration primarily for graphing later
    J_history = []
    w = copy.deepcopy(w_in)     #avoid modifying global w within function
    b = b_in

    for i in range(num_iter):
        #calculate gradient and update the parameters
        dj_dw, dj_db = compute_gradient_logistic(X, y, w, b)

        #Update Parameters using w, b, alpha and gardient
        w = w - alpha * dj_dw
        b = b - alpha * dj_db

        #Save cost J each iteration
        if i<100000:          #prevent resource exhaustion
            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_iter / 10) ==0:
          print(f"Iteration {i:4d}  Cost {J_history[-1]}  ")

    return w, b, J_history                #return final w, b and J history for graphing
```

✓ 0.0s Python

C - Gradient Descent

```c
struct caches gradient_descent(float x_in[][n], int y_in[], double **w_in, double b_in, float alpha, int num_iters){

    /*
     Performs 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 (scalar) : number of iterations to run gradient descent

    Returns:
      w (ndarray (n,))   : Updated values of parameters
      b (scalar)         : Updated value of parameter
    */

    struct caches cache;
    struct Grads grads;

    cache.J_history = (double *)calloc(num_iters, sizeof(double));
    cache.w = deepcopy(w_in);                   //avoid modifying global w within function
    cache.b=b_in;

    for (int i = 0; i < num_iters; i++)
    {
        //calculate gradient and update the parameters
        grads=compute_gradient_logistic(x_in, y_in, cache.w, cache.b);

        //update parameters using w, b, alpha and gradient
        for (int j = 0; j < n; j++)
        {
            cache.w[j][0] = cache.w[j][0] - (grads.djdw[j] * alpha);
        }
        cache.b = cache.b - (grads.djdb * alpha);

        double J=compute_cost_logistic(x_in, y_in, cache.w, cache.b);

        cache.J_history[i] = J;

        if (i % (num_iters / 10)==0)
        {
        printf("Iteration %d \t Cost %f\n", i, cache.J_history[i]);
        }

    }

    return cache;
}
```

Python - main

```python
def main():
    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])

    w_tmp  = np.zeros_like(X_train[0])
    b_tmp  = 0.
    alph = 0.1
    iters = 10000000

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

if __name__=="__main__":
    main()
```

✓ 2m 7.0s Python

```
Iteration    0  Cost 0.684610468560574
Iteration 10000  Cost 0.01711604647887364
Iteration 20000  Cost 0.008523403979166467
Iteration 30000  Cost 0.005672197191107633
Iteration 40000  Cost 0.004250161053834308
Iteration 50000  Cost 0.003398230224179212
Iteration 60000  Cost 0.0028308425601004327
Iteration 70000  Cost 0.002425848306579758
Iteration 80000  Cost 0.0021222573122028584
Iteration 90000  Cost 0.0018862216652143864

updated parameters: w:[8.35313087 8.15226727],    b:-22.690605796630248
```

C - main

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

#define m 6
#define n 2

struct Grads{
    double *djdw;
    double djdb;
} ;

struct caches{
    double *J_history;
    double **w;
    double b;
};

double **deepcopy(double **src);
double compute_cost_logistic(float x[][n], int y[], double **w, double b);
struct Grads compute_gradient_logistic(float x[][n], int y[], double **w, double b);
struct caches gradient_descent(float x_in[][n], int y_in[], double **w_in, double b_in, float alpha, int num_iters);


int main(){

    float X_train[][n] = {{0.5, 1.5}, {1, 1}, {1.5, 0.5}, {3, 0.5}, {2, 2}, {1, 2.5}};
    int y_train[] = {0, 0, 0, 1, 1, 1};

    double **w_init=(double **)calloc(n, sizeof(double *));
    if (w_init == NULL)
    {
        perror("Error in allocating memory");
    }
    for (int i = 0; i < n; i++)
    {
        w_init[i] = (double *)calloc(1, sizeof(double));
        if (w_init[i]==NULL)
        {
            perror("Error in allocating memory");
            for (int j = 0; j < i; j++)
            {
                free(w_init[j]);
            }
            free(w_init);

        }
        w_init[i][0] = 0;

    }

    double b_init = 0;
    double alpha=1e-1;
    int num_iters=10000000;

    struct caches cache;
    cache = gradient_descent(X_train, y_train, w_init, b_init, alpha, num_iters);

    printf("updated parameters: ");
        for (int i = 0; i < n; i++)
        {
            printf("w[%d]: %f \t", i, cache.w[i][0]);
        }
        printf("b: %f\n", cache.b);

    for (int i = 0; i < n; i++)
    {
        free(w_init[i]);
        free(cache.w[i]);
    }
    free(cache.w);
    free(w_init);
    free(cache.J_history);

    return 0;
}
```

```
● venvsuzanodero@suzans-MacBook-Air Gradient_Descent % gcc gradient_descent.c
● venvsuzanodero@suzans-MacBook-Air Gradient_Descent % ./a.out
Iteration 0      Cost 0.684610
Iteration 10000          Cost 0.017116
Iteration 20000          Cost 0.008523
Iteration 30000          Cost 0.005672
Iteration 40000          Cost 0.004250
Iteration 50000          Cost 0.003398
Iteration 60000          Cost 0.002831
Iteration 70000          Cost 0.002426
Iteration 80000          Cost 0.002122
Iteration 90000          Cost 0.001886
updated parameters: w[0]: 8.353131      w[1]: 8.152267  b: -22.690606
```