Univariate Linear Regression model

1.0 Problem statement

Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet.

- You would like to expand your business to cities that may give your restaurant higher profits.
- The chain already has restaurants in various cities and you have data for profits and populations from the cities.
- You also have data on cities that are candidates for a new restaurant.
    - For these cities, you have the city population.

Can you use the data to help you identify which cities may potentially give your business higher profits?

> Note:

- `X` is the population of a city
- `y` is the profit of a restaurant in that city. A negative value for profit indicates a loss.
    - Both `X` and `y` are arrays.

Population of a city (x 10,000) as x	**Profit of a restaurent (x $10,000) as $f_{w,b}(x^{(i)})$ or y**
6.1101	17.592
5.5277	9.1302
8.5186	13.662
7.0032	11.854
5.8598	6.8233

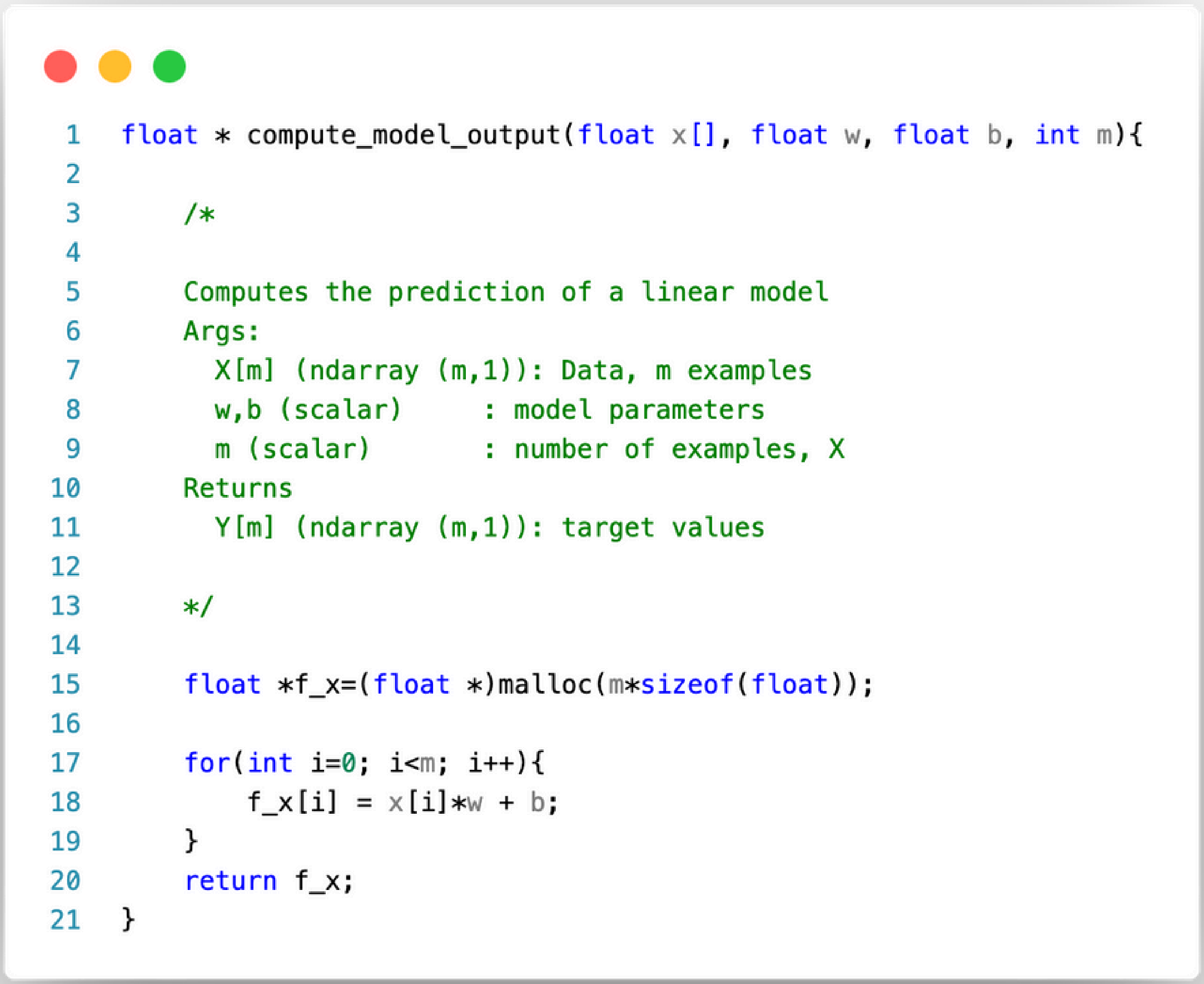
Number of training example (size (1000 sqft) as x) m

In this case $m = 5$

2.0 Model Function

The model function for linear regression (which is a function that maps from `x` to `y` is represented as

$$f_{w,b}(x^{(i)}) = w * x^{(i)} + b \tag{1}$$



```c
float * compute_model_output(float x[], float w, float b, int m){

    /*

    Computes the prediction of a linear model
    Args:
      X[m] (ndarray (m,1)): Data, m examples
      w,b (scalar)     : model parameters
      m (scalar)       : number of examples, X
    Returns
      Y[m] (ndarray (m,1)): target values

    */

    float *f_x=(float *)malloc(m*sizeof(float));

    for(int i=0; i<m; i++){
        f_x[i] = x[i]*w + b;
    }
    return f_x;
}
```

3.0 Compute Cost

Cost is the measure of how well our model will predict the target output well, in this case target output is Profit of a restaurent.

Gradient descent involve repeated steps to adjust the values of `w` and `b` to get smaller and smaller `Cost`, $J(w,b)$.

The equation for Cost with one variable

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

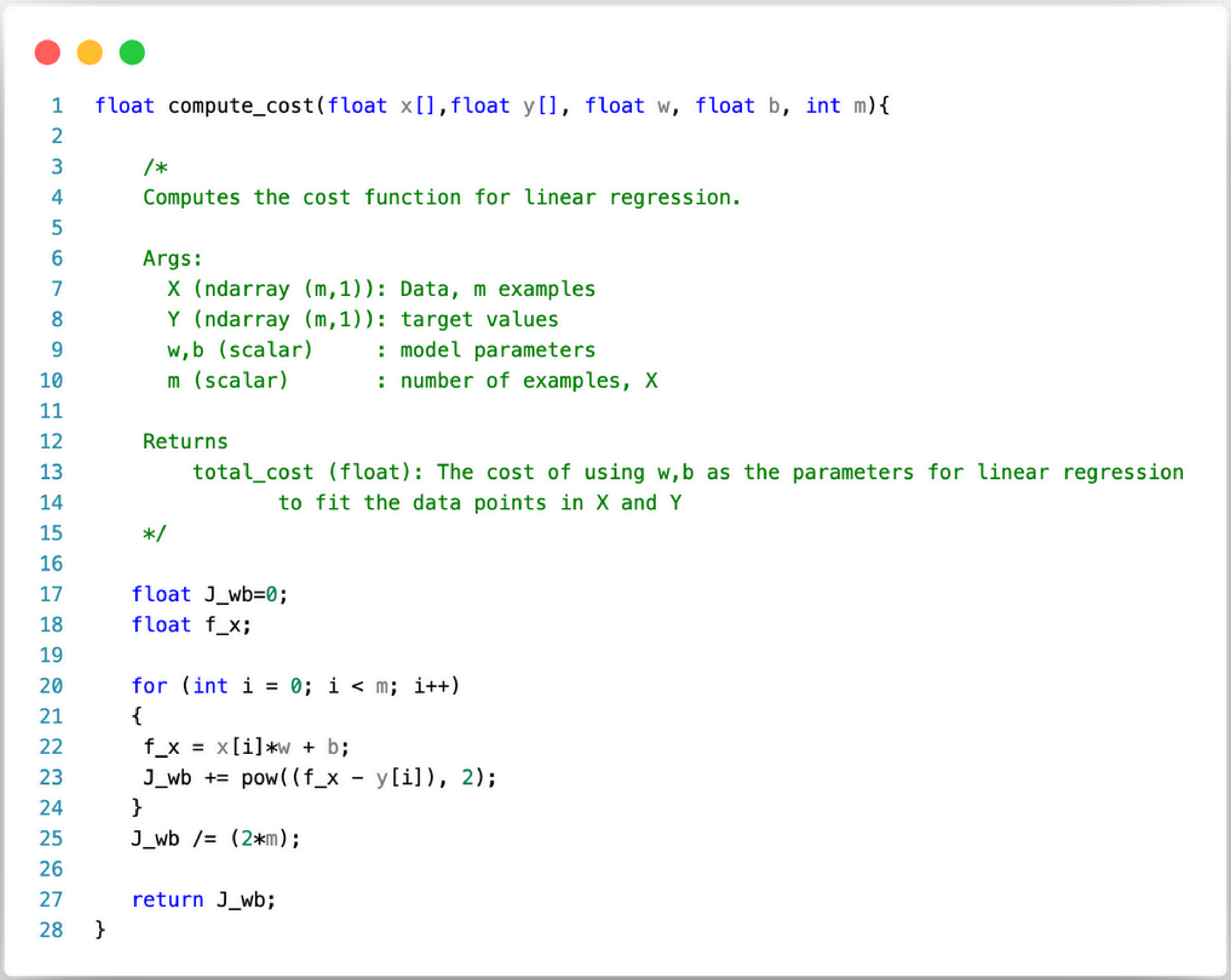
where

$$f_{w,b}(x^{(i)}) = w * x^{(i)} + b \tag{1}$$

- $f_{w,b}(x^{(i)})$ is our prediction for example i using parameters w, b.
- $(f_{w,b}(x^{(i)}) - y^{(i)})^2$ is the squared difference between the target value and the prediction.
- These differences are summed over all the m examples and divided by `2*m` to produce the cost, $J(w,b)$.

> Note,

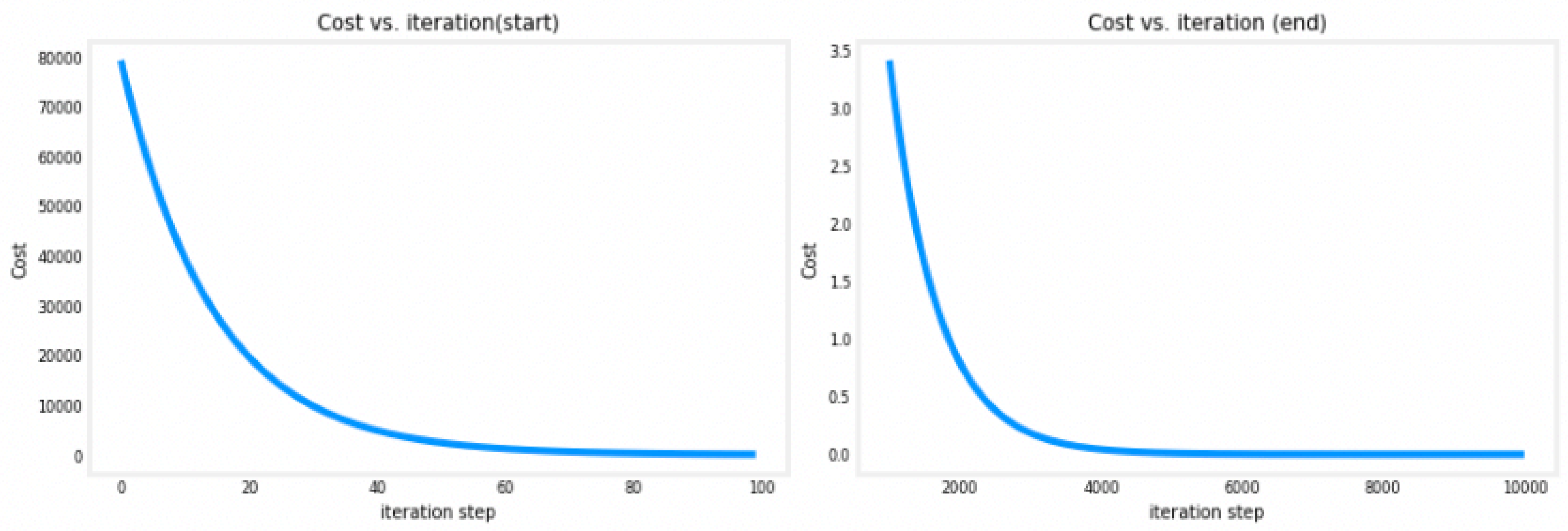
- Summation ranges are typically from 1 to m, while code will be from 0 to m-1.



```c
float compute_cost(float x[],float y[], float w, float b, int m){

    /*
    Computes the cost function for linear regression.

    Args:
      X (ndarray (m,1)): Data, m examples
      Y (ndarray (m,1)): target values
      w,b (scalar)     : model parameters
      m (scalar)       : number of examples, X

    Returns
        total_cost (float): The cost of using w,b as the parameters for linear regression
               to fit the data points in X and Y
    */

   float J_wb=0;
   float f_x;

   for (int i = 0; i < m; i++)
   {
    f_x = x[i]*w + b;
    J_wb += pow((f_x - y[i]), 2);
   }
   J_wb /= (2*m);

   return J_wb;
}
```

3.1 Cost versus iterations of gradient descent

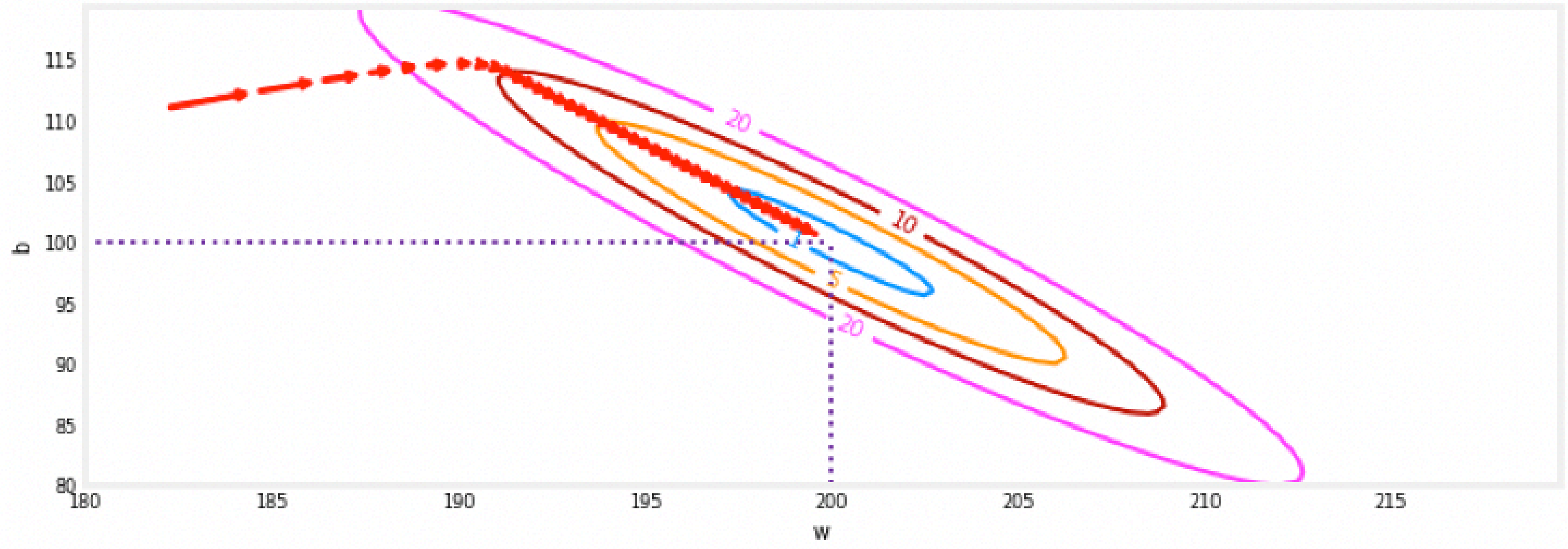
A plot of cost versus iterations is a useful measure of progress in gradient descent. Cost should always decrease in successful runs. The change in cost is so rapid initially, it is useful to plot the initial decent on a different scale than the final descent. In the plots below, note the scale of cost on the axes and the iteration step.



3.2 Plot of cost J(w,b) vs w,b with path of gradient descent

Plot shows the $cost(w,b)$ over a range of w and b. Cost levels are represented by the rings. Overlayed, using red arrows, is the path of gradient descent. Here are some things to note:

- The path makes steady (monotonic) progress toward its goal.
- initial steps are much larger than the steps near the goal.



3.3 Convex Cost surface

The fact that the cost function squares the loss, $\sum_{i=0}^{m-1}(f_{w,b}(x^{(i)}) - y^{(i)})^2$ ensures that the 'error surface' is convex like a soup bowl. It will always have a minimum that can be reached by following the gradient in all dimensions.

4.0 Gradient Descent

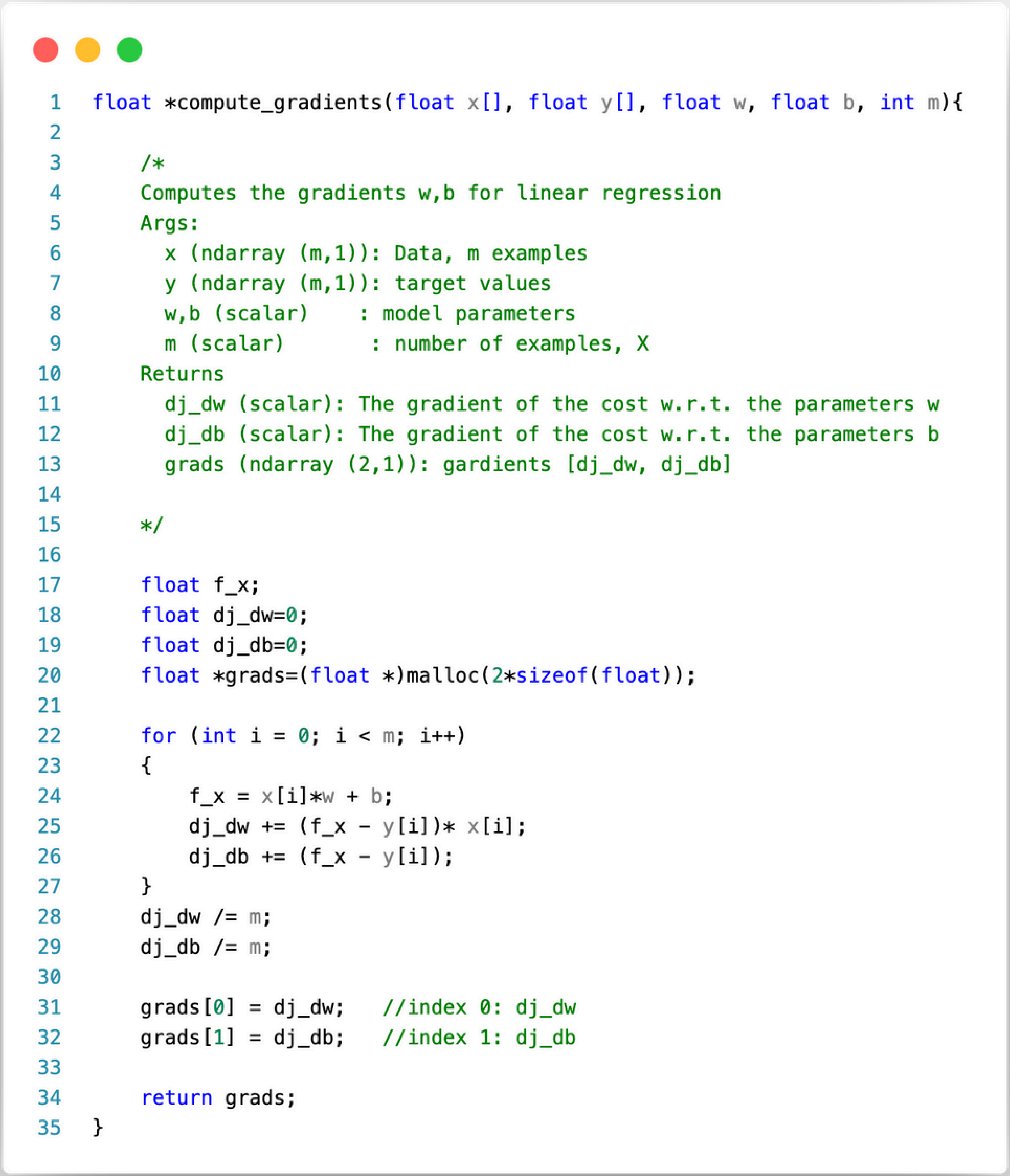
In linear regression, we utilize input training data to fit the parameters w,b by minimizing a measure of the error between our predictions $f_{w,b}(x^{(i)})$ and the actual data $y^{(i)}$. The measure is called the $cost$, $J(w,b)$. In training you measure the cost over all of our training samples $x^{(i)},y^{(i)}$

$$\begin{align*} \text{repeat}&\text{ until convergence:} \; \lbrace \newline
w &= w - \alpha \frac{\partial J(w,b)}{\partial w} \tag{3} \newline
b &= b - \alpha \frac{\partial J(w,b)}{\partial b} \newline \rbrace
\end{align*}$$

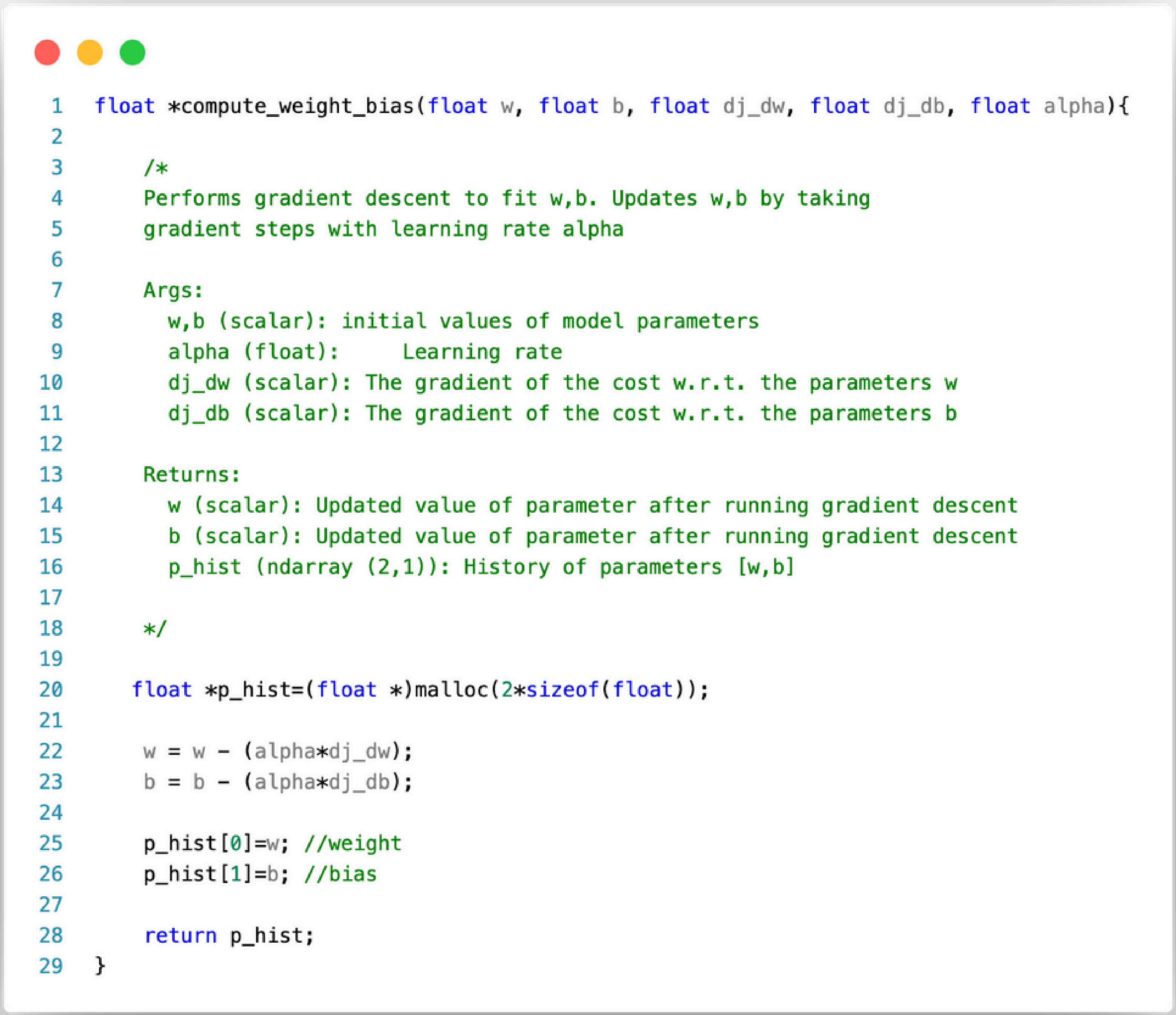
where, parameters w, b are updated simultaneously.
The gradient is defined as:

$$\begin{align}
\frac{\partial J(w,b)}{\partial w} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)}) * x^{(i)} \tag{4}\\
\frac{\partial J(w,b)}{\partial b} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)}) \tag{5}
\end{align}$$

Here *simultaniously* means that you calculate the partial derivatives for all the parameters before updating any of the parameters.



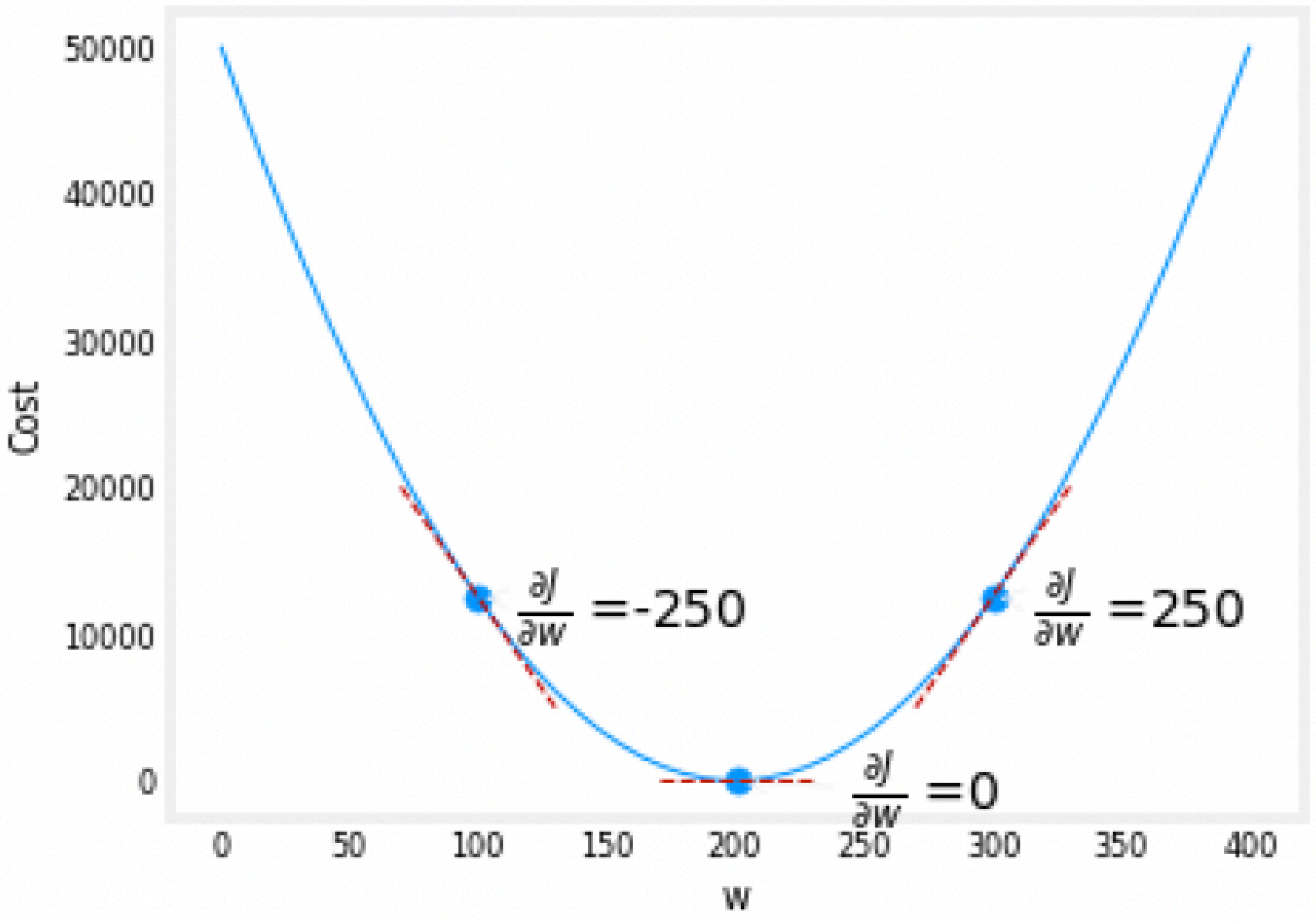
```c
float *compute_gradients(float x[], float y[], float w, float b, int m){

    /*
    Computes the gradients w,b for linear regression
    Args:
      x (ndarray (m,1)): Data, m examples
      y (ndarray (m,1)): target values
      w,b (scalar)    : model parameters
      m (scalar)       : number of examples, X
    Returns
      dj_dw (scalar): The gradient of the cost w.r.t. the parameters w
      dj_db (scalar): The gradient of the cost w.r.t. the parameters b
      grads (ndarray (2,1)): gardients [dj_dw, dj_db]

    */

    float f_x;
    float dj_dw=0;
    float dj_db=0;
    float *grads=(float *)malloc(2*sizeof(float));

    for (int i = 0; i < m; i++)
    {
        f_x = x[i]*w + b;
        dj_dw += (f_x - y[i])* x[i];
        dj_db += (f_x - y[i]);
    }
    dj_dw /= m;
    dj_db /= m;

    grads[0] = dj_dw;   //index 0: dj_dw
    grads[1] = dj_db;   //index 1: dj_db

    return grads;
}
```



```c
float *compute_weight_bias(float w, float b, float dj_dw, float dj_db, float alpha){

    /*
    Performs gradient descent to fit w,b. Updates w,b by taking
    gradient steps with learning rate alpha

    Args:
      w,b (scalar): initial values of model parameters
      alpha (float):     Learning rate
      dj_dw (scalar): The gradient of the cost w.r.t. the parameters w
      dj_db (scalar): The gradient of the cost w.r.t. the parameters b

    Returns:
      w (scalar): Updated value of parameter after running gradient descent
      b (scalar): Updated value of parameter after running gradient descent
      p_hist (ndarray (2,1)): History of parameters [w,b]

    */

   float *p_hist=(float *)malloc(2*sizeof(float));

    w = w - (alpha*dj_dw);
    b = b - (alpha*dj_db);

    p_hist[0]=w; //weight
    p_hist[1]=b; //bias

    return p_hist;
}
```

4.1 Cost vs w, with gradients, b set to 100.

The plot shows $\frac{\partial J(w,b)}{\partial w}$ or the slope of the cost curve relative to w at three points. The derivative is negative. Due to the 'bowl shape', the derivatives will always lead gradient descent toward the bottom where the gradient is zero.

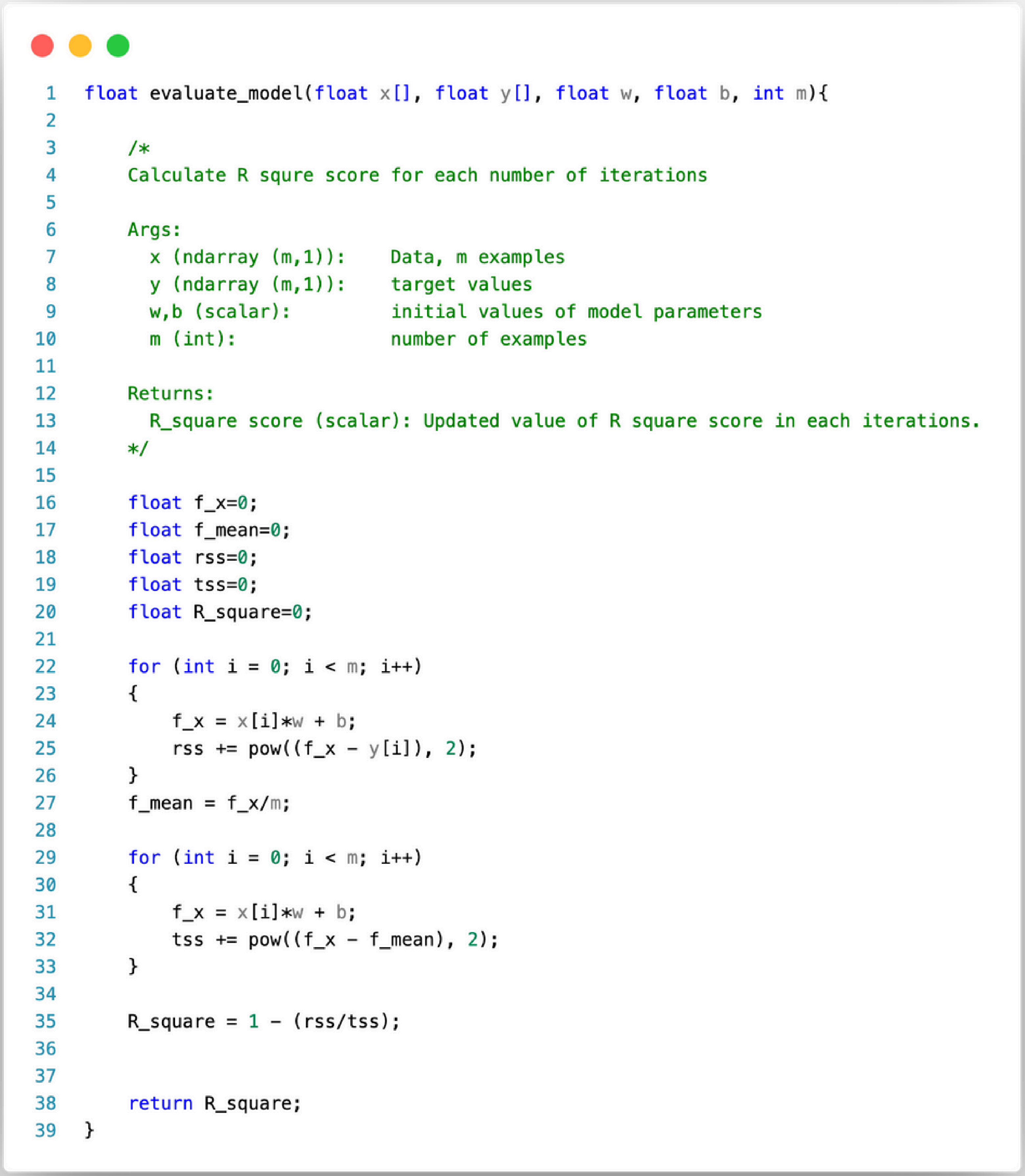


5.0 Evaluating our model

To evaluate the estimation model, we use coefficient of determination which is given by the following formula:

$$R^2 = 1 - \frac{Residual\ Square\ Sum}{Total\ Square\ Sum}$$

$$R^2 = 1 - \frac{\sum_{i=0}^{(m-1)} (f_{w,b}(x^{(i)}) - y^i)^2}{\sum_{i=0}^{(m-1)} (f_{w,b}(x^{(i)}) - f_{w,b}(x^{(i)})_{mean})^2}$$



```c
float evaluate_model(float x[], float y[], float w, float b, int m){

    /*
    Calculate R squre score for each number of iterations

    Args:
      x (ndarray (m,1)):    Data, m examples
      y (ndarray (m,1)):    target values
      w,b (scalar):         initial values of model parameters
      m (int):              number of examples

    Returns:
      R_square score (scalar): Updated value of R square score in each iterations.
    */

    float f_x=0;
    float f_mean=0;
    float rss=0;
    float tss=0;
    float R_square=0;

    for (int i = 0; i < m; i++)
    {
        f_x = x[i]*w + b;
        rss += pow((f_x - y[i]), 2);
    }
    f_mean = f_x/m;

    for (int i = 0; i < m; i++)
    {
        f_x = x[i]*w + b;
        tss += pow((f_x - f_mean), 2);
    }

    R_square = 1 - (rss/tss);


    return R_square;
}
```

6.0 Learning parameters using batch gradient descent

You will now find the optimal parameters of a linear regression model by using batch gradient descent. Recall batch refers to running all the examples in one iteration.

- You don't need to implement anything for this part. Simply run the cells below.
- A good way to verify that gradient descent is working correctly is to look at the value of $J(w, b)$ and check that it is decreasing with each step.
- Assuming you have implemented the gradient and computed the cost correctly and you have an appropriate value for the learning rate alpha, $J(w, b)$ should never increase and should converge to a steady value by the end of the algorithm.

6.1 Expected Output

Optimal w, b found by gradient descent

w	**b**
1.492054	-3.216610

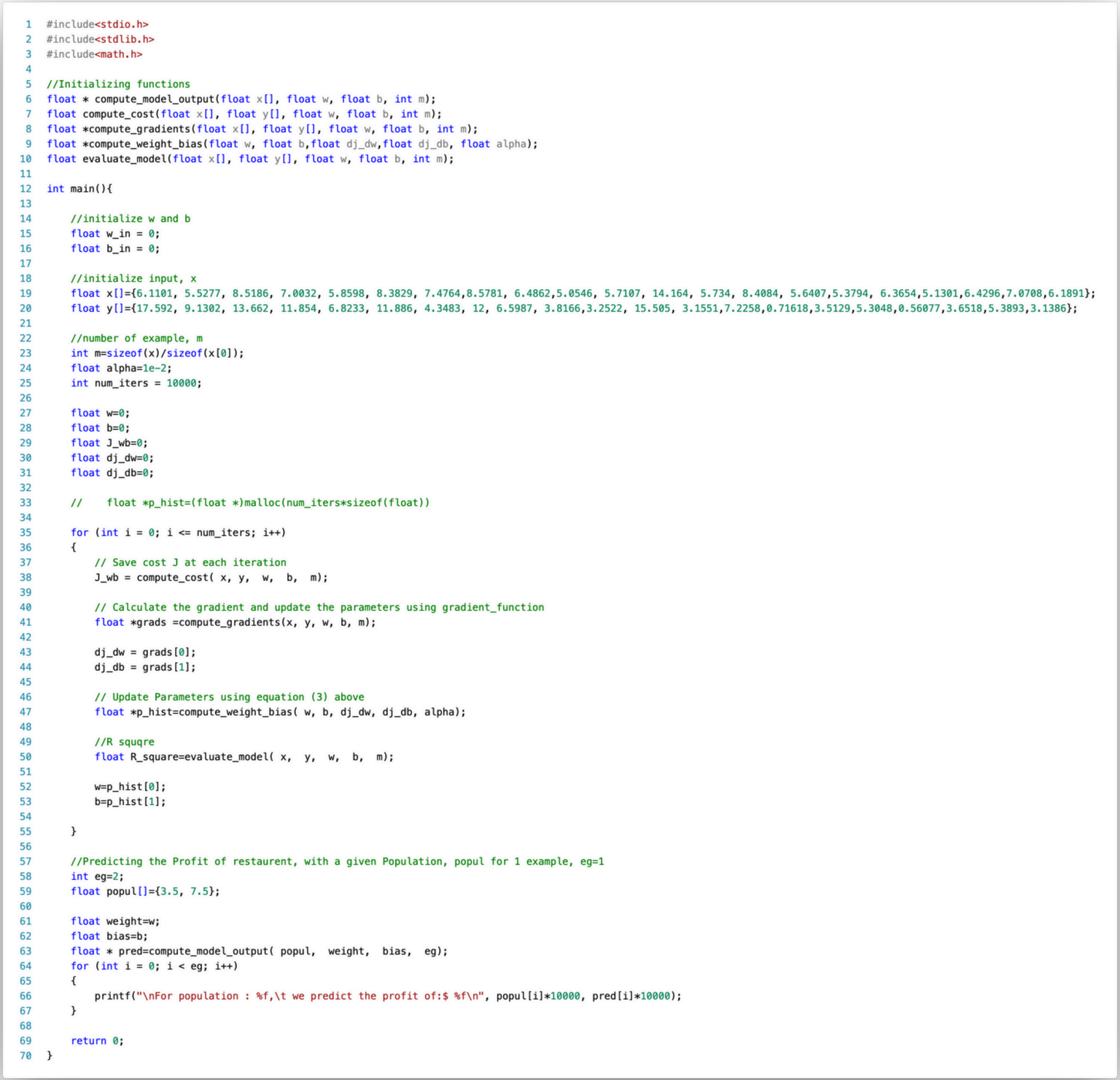
We will now use our final parameters w, b to find our prediction for single example.

recall:

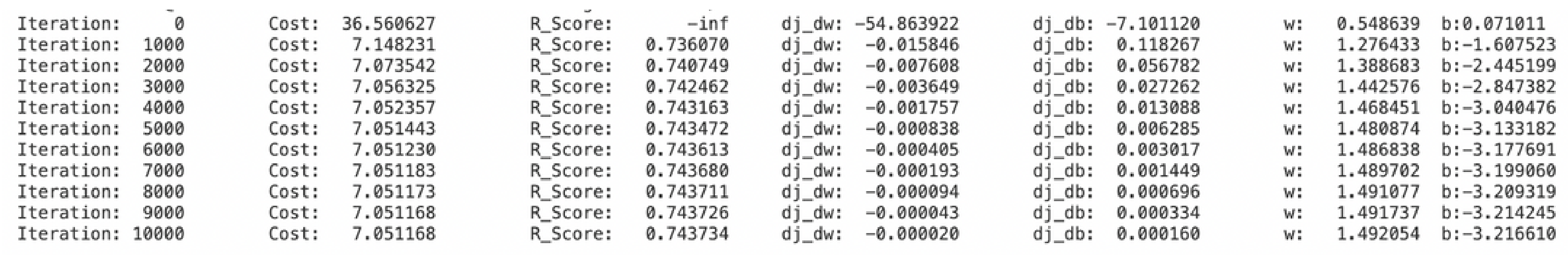
$$f_{w,b}(x^{(i)}) = w * x^{i} + b$$

Let's predict what profit will be for the ares of 35,000 and 70,000 people

- The model takes in population of a city in 10,000s as input.
- Therefore, 35,000 people can be translated into an input to the model as `input[] = {3.5}`
- Similarly, 70,000 people can be translated into an input to the model as `input[] = {7.5}`



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

//Initializing functions
float * compute_model_output(float x[], float w, float b, int m);
float compute_cost(float x[], float y[], float w, float b, int m);
float *compute_gradients(float x[], float y[], float w, float b, int m);
float *compute_weight_bias(float w, float b,float dj_dw,float dj_db, float alpha);
float evaluate_model(float x[], float y[], float w, float b, int m);

int main(){

    //initialize w and b
    float w_in = 0;
    float b_in = 0;

    //initialize input, x
    float x[]={6.1101, 5.5277, 8.5186, 7.0032, 5.8598, 8.3829, 7.4764,8.5781, 6.4862,5.0546, 5.7107, 14.164, 5.734, 8.4084, 5.6407,5.3794, 6.3654,5.1301,6.4296,7.0708,6.1891};
    float y[]={17.592, 9.1302, 13.662, 11.854, 6.8233, 11.886, 4.3483, 12, 6.5987, 3.8166,3.2522, 15.505, 3.1551,7.2258,0.71618,3.5129,5.3048,0.56077,3.6518,5.3893,3.1386};

    //number of example, m
    int m=sizeof(x)/sizeof(x[0]);
    float alpha=1e-2;
    int num_iters = 10000;

    float w=0;
    float b=0;
    float J_wb=0;
    float dj_dw=0;
    float dj_db=0;

    //    float *p_hist=(float *)malloc(num_iters*sizeof(float))

    for (int i = 0; i <= num_iters; i++)
    {
        // Save cost J at each iteration
        J_wb = compute_cost( x, y,  w,  b,  m);

        // Calculate the gradient and update the parameters using gradient_function
        float *grads =compute_gradients(x, y, w, b, m);

        dj_dw = grads[0];
        dj_db = grads[1];

        // Update Parameters using equation (3) above
        float *p_hist=compute_weight_bias( w, b, dj_dw, dj_db, alpha);

        //R squqre
        float R_square=evaluate_model( x,  y,  w,  b,  m);

        w=p_hist[0];
        b=p_hist[1];

    }

    //Predicting the Profit of restaurent, with a given Population, popul for 1 example, eg=1
    int eg=2;
    float popul[]={3.5, 7.5};

    float weight=w;
    float bias=b;
    float * pred=compute_model_output( popul,  weight,  bias,  eg);
    for (int i = 0; i < eg; i++)
    {
        printf("\nFor population : %f,\t we predict the profit of:$ %f\n", popul[i]*10000, pred[i]*10000);
    }

    return 0;
}
```



```
Iteration:     0        Cost:  36.560627        R_Score:      -inf     dj_dw: -54.863922        dj_db: -7.101120        w:   0.548639  b:0.071011
Iteration:  1000        Cost:   7.148231        R_Score:  0.736070     dj_dw:  -0.015846        dj_db:  0.118267        w:   1.276433  b:-1.607523
Iteration:  2000        Cost:   7.073542        R_Score:  0.740749     dj_dw:  -0.007608        dj_db:  0.056782        w:   1.388683  b:-2.445199
Iteration:  3000        Cost:   7.056325        R_Score:  0.742462     dj_dw:  -0.003649        dj_db:  0.027262        w:   1.442576  b:-2.847382
Iteration:  4000        Cost:   7.052357        R_Score:  0.743163     dj_dw:  -0.001757        dj_db:  0.013088        w:   1.468451  b:-3.040476
Iteration:  5000        Cost:   7.051443        R_Score:  0.743472     dj_dw:  -0.000838        dj_db:  0.006285        w:   1.480874  b:-3.133182
Iteration:  6000        Cost:   7.051230        R_Score:  0.743613     dj_dw:  -0.000405        dj_db:  0.003017        w:   1.486838  b:-3.177691
Iteration:  7000        Cost:   7.051183        R_Score:  0.743680     dj_dw:  -0.000193        dj_db:  0.001449        w:   1.489702  b:-3.199060
Iteration:  8000        Cost:   7.051173        R_Score:  0.743711     dj_dw:  -0.000094        dj_db:  0.000696        w:   1.491077  b:-3.209319
Iteration:  9000        Cost:   7.051168        R_Score:  0.743726     dj_dw:  -0.000043        dj_db:  0.000334        w:   1.491737  b:-3.214245
Iteration: 10000        Cost:   7.051168        R_Score:  0.743734     dj_dw:  -0.000020        dj_db:  0.000160        w:   1.492054  b:-3.216610
```

```
For population : 35000.000000,   we predict the profit of:$ 20055.775391

For population : 75000.000000,   we predict the profit of:$ 79737.921875
```