

Linear Regression by Rohitash Chandra

Gradient descent

Gradient descent is a simple optimisation procedure that you can use with many machine learning algorithms. Gradient descent can only work for differentiable functions. The goal is to find the local minimum of a function using gradient descent where we take steps proportional to the **negative** of the gradient.

Consider the Rosenbrock function of two variables (dimensions):

$$f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2$$

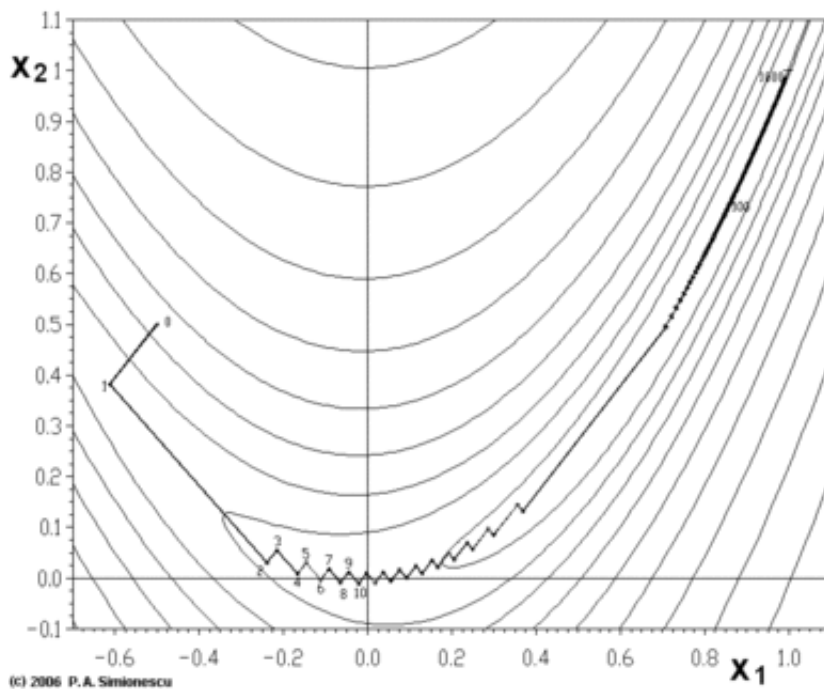


Figure: Plan view of Rosenbrock function. Adapted from "Rosenbrock function" by Wikipedia, 2020. Retrieved from: https://en.wikipedia.org/wiki/Rosenbrock_function.

The above figure shows the plan view of Rosenbrock function $f(x_1, x_2)$ vs $(x_1 \text{ and } x_2)$ and how gradient descent traverses with the help of gradients to iteratively find the lowest point.

Now consider another function:

$$\left(F(x, y) = \frac{1}{2}x^2 - \frac{1}{4}y^2 + 3 \right) \cos(2x + 1 - e^y)$$

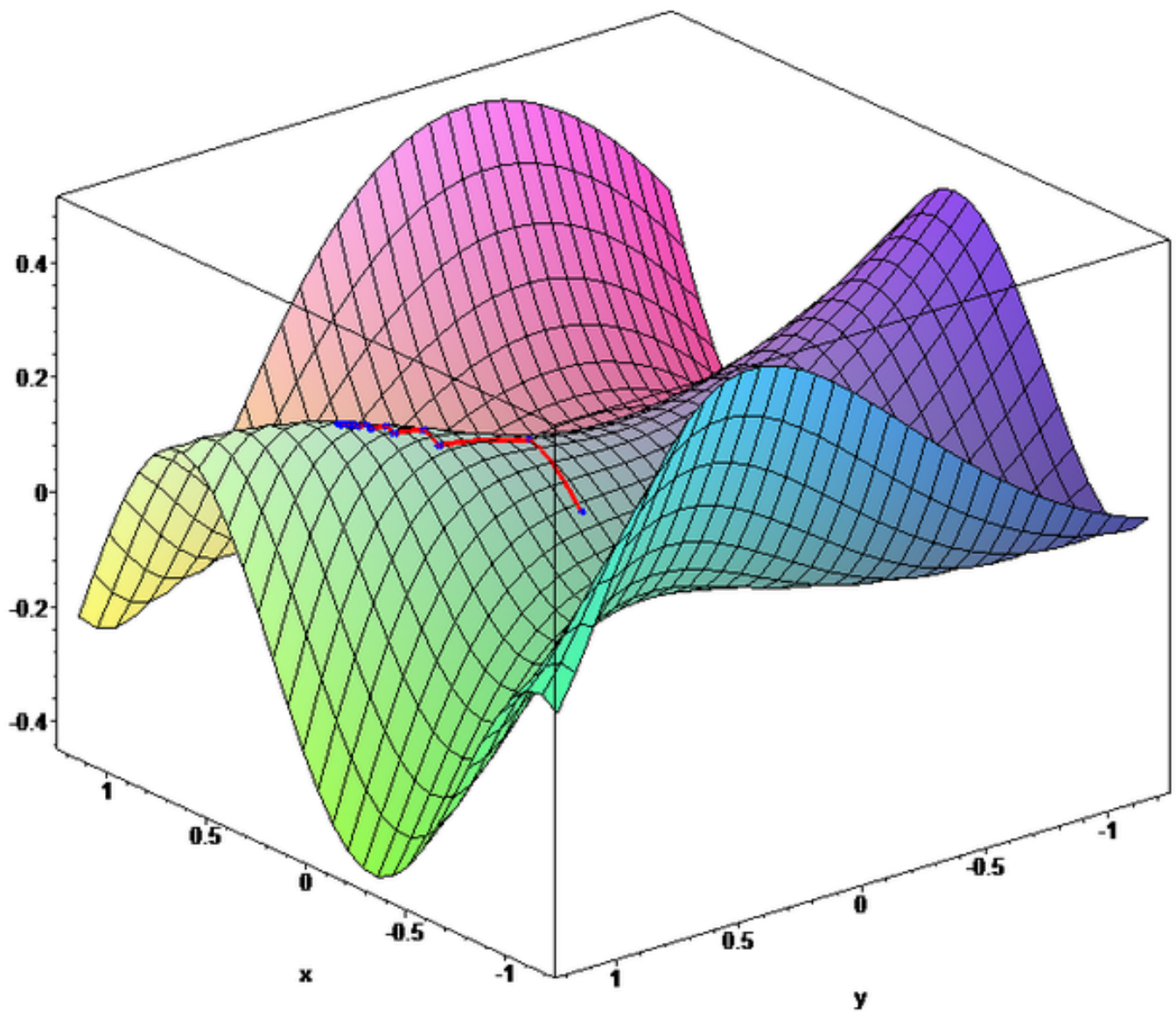


Figure: Three-dimensional view of $F(x, y)$ on the vertical axis vs parameters x and y axes. Adapted from "Rosenbrock function" by Wikipedia, 2020. Retrieved from https://en.wikipedia.org/wiki/Rosenbrock_function.

The above figure shows the three-dimensional view of $F(x, y)$ on the vertical axis vs parameters (x and y axes). Notice how gradient descent traverses with the help of gradients to iteratively find the lowest point.

The following code examples apply the gradient descent algorithm to find the minimum of the function

$$f(x) = x^4 - 3x^3 + 2$$

with derivative

$$f'(x) = 4x^3 - 9x^2$$

Solving for

$$4x^3 - 9x^2 = 0$$

and evaluation of the second derivative at the solutions shows the function has a plateau point at 0 and a global minimum at

$$x = \frac{9}{4}.$$

We note that in the above example, you can use the second derivative and solve it to find the minimum, but in more complex functions and models, that is not possible (such as Rosenbrock function and logistic regression model), hence gradient descent is used.



Practise gradient descent optimisation using Python. If necessary you may wish to go to Orientation week to refresh your Python skills.

Now that you have learned how gradient descent optimisation works, run the following codes by clicking on the 'Run' button.

▶ Run

PYTHON



```
1 #~source: https://en.wikipedia.org/wiki/Gradient_descent
2 # code source: https://en.wikipedia.org/w/index.php?title=Gradient_desce
3
4 next_x = 6# We start the search at x = 6
5 gamma = 0.01# Step size multiplier
6 precision = 0.00001# Desired precision of result
7 max_iters = 10000# Maximum number of iterations
8
9 # Derivative
10 #function
11 def df(x):
12     return 4 * x ** 3 - 9 * x ** 2
13
14 for i in range(max_iters):
```

The code above shows how gradient descent can be applied to a simple function using Python. Note the gradient of the function is computed after differentiating the function with respect to x (line 10) in the above code.



Practise gradient descent optimisation using R. If necessary you may wish to go to Orientation week to refresh your R skills.

Below, you can see the same activity completed using R. Notice the difference in the syntax of the two languages.

▶ Run

R



```
1 #~ source: https://en.wikipedia.org/wiki/Gradient\_descent
2 #https://en.wikipedia.org/w/index.php?title=Gradient\_descent&oldid=96627
3 # set up a stepsize multiplier
4 gamma = 0.003
5
6 # set up a number of iterations
7 iter = 500
8
9 # define the objective function  $f(x) = x^4 - 3x^3 + 2$ 
10 objFun = function(x) return( $x^4 - 3x^3 + 2$ )
11
12 # define the gradient of  $f(x) = x^4 - 3x^3 + 2$ 
13 gradient = function(x) return( $(4x^3) - (9x^2)$ )
14
```

Although gradient descent does not work very well for Rosenbrock function, we used this example to show that gradient descent is an optimisation method, which is extended for learning data-based models such as linear regression, logistic regression, and neural networks models.



For additional information, you may wish to watch the following video on how gradient descent works. The example in the video uses a different function example.

How gradient descent works

Bielinskas.V. (2019, August 09). *How Gradient Descent works?* [online video]. Retrieved from <https://www.youtube.com/watch?v=Gbz8RljxIH0>.

Linear regression

This lesson will help you understand the mathematics behind gradient descent and linear regression. In a linear regression model, our goal is to find the value of the coefficients or parameters of the linear model for given data such that the output of the linear model best fits the data.

The basics of the linear regression model can be summed up by the following diagram. Note that our linear model is $y = mx + b$ which in the diagram is given as $y = w_0 + w_1x$.

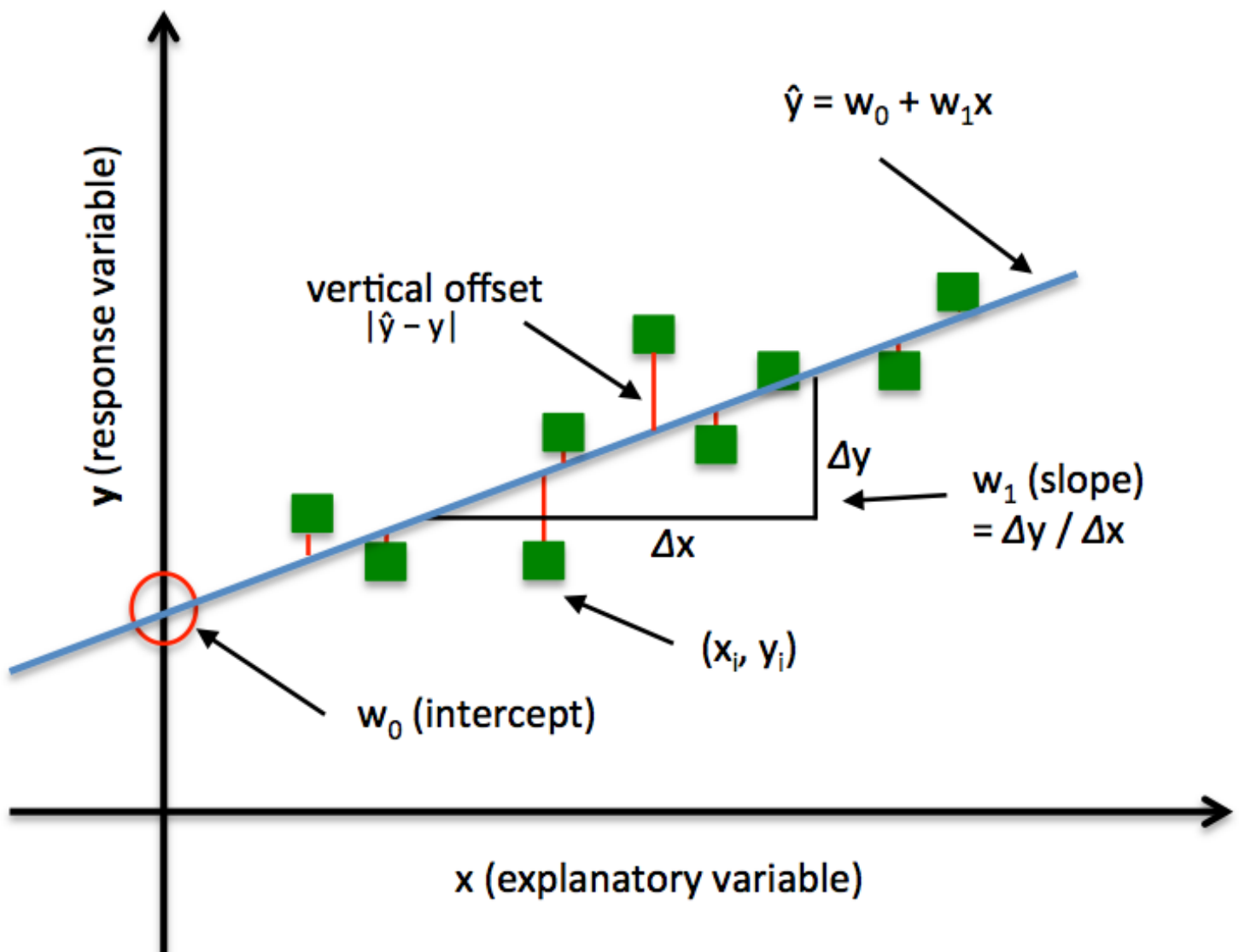


Figure: Linear regression. Adapted from "Linear regression" by Github, n.d.
(http://rasbt.github.io/mlxtend/user_guide/regressor/LinearRegression/)

Now that you have learned the mathematics behind linear regression and how to apply it, let's apply



Below are the code snippets showing how gradient descent is used to find parameters of a linear model given some data below with visualisation. First, we consider some data that gives x with relationship y as shown below.

PYTHON



```

1 x, y
2 32.502345269453031,31.70700584656992
3 53.426804033275019,68.77759598163891
4 61.530358025636438,62.562382297945803
5 47.475639634786098,71.546632233567777
6 59.813207869512318,87.230925133687393
7 55.142188413943821,78.211518270799232
8 52.211796692214001,79.64197304980874
9 39.299566694317065,59.171489321869508
10 48.10504169176825,75.331242297063056
11 52.550014442733818,71.300879886850353
12 45.419730144973755,55.165677145959123
13 54.351634881228918,82.478846757497919
14 44.164049496773352,62.008923245725825

```

Suppose we want to model a given set of points in data with a line (the linear regression model). We use the linear model ($y = mx + b$) equation, where m is the line's slope and b is the line's y -intercept. We need to find the best set of slope m and y -intercept b values that cover the data points. Note that this example considers only one feature of the input data (x), but linear models can take a vector or multiple input features.

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

The error function above considers the target labels t_d over data D given output of the lineal model O_d over set of parameters w that in our case is denoted by (m, b) , hence also given as:

$$\text{Error}_{(m,b)} = \frac{1}{N} \sum_{i=1}^N (y_i - (mx_i + b))^2$$

It takes in a (m, b) pair and returns an error value based on how well the line fits the data. Hence, we iterate through each (x, y) point in our data set and sum the square distances between each point's y value and the candidate line's y value (computed at $mx + b$) as shown in the code below.



```

1 # Source: https://github.com/mattnedrich/GradientDescentExample
2 # y = mx + b
3 # m is slope, b is y-intercept
4 def compute_error_for_line_given_points(b, m, points):
5     totalError = 0
6     for i in range(0, len(points)):
7         x = points[i, 0]
8         y = points[i, 1]
9         totalError += (y - (m * x + b)) ** 2
10    return totalError / float(len(points))

```

Next, we need to find a way to adjust the parameters (m, b) so that the line better fits the data over time which is done by gradient descent. Before we proceed with adjusting parameters, we need to calculate the gradient for (m, b) given as (new m , new b) for the entire set of points, with given x and y data features as shown in the equation below.

$$\frac{\partial}{\partial m} = \frac{2}{N} \sum_{i=1}^N -x_i (y_i - (mx_i + b))$$

$$\frac{\partial}{\partial b} = \frac{2}{N} \sum_{i=1}^N -(y_i - (mx_i + b))$$

The learning rate is a user-defined parameter for gradient descent which is used to determine the extent of the steps that need to be taken when adjusting the parameters during training (shown in Lines 11 and 12 of code below).




```

1 # Source: https://github.com/mattnedrich/GradientDescentExample
2 def step_gradient(b_current, m_current, points, learningRate):
3     b_gradient = 0
4     m_gradient = 0
5     N = float(len(points))
6     for i in range(0, len(points)):
7         x = points[i, 0]
8         y = points[i, 1]
9         b_gradient += -(2/N) * (y - ((m_current * x) + b_current))
10        m_gradient += -(2/N) * x * (y - ((m_current * x) + b_current))
11    new_b = b_current - (learningRate * b_gradient)
12    new_m = m_current - (learningRate * m_gradient)
13    return [new_b, new_m]



```

Now that we have the gradients, we use it to update the parameters given the defined number of

iterations set by the user as shown below.

```
PYTHON   
1 def gradient_descent_runner(points, starting_b, starting_m, learning_rate):  
2     b = starting_b  
3     m = starting_m  
4     for i in range(num_iterations):  
5         b, m = step_gradient(b, m, array(points), learning_rate)  
6     return [b, m]
```

Next, we place all the above code snippets together by calling them when in need. Below is the key function that sets up the problem by reading data and calling the previous functions that implement the linear regression model by training using gradient descent function. We need to set a value for initial b and m , and in this case its 0, but it can also be randomly assigned.

```
 Run PYTHON   
1 def run():  
2     points = genfromtxt("data_linearreg.csv", delimiter=",")  
3     learning_rate = 0.0001  
4     initial_b = 0 # initial y-intercept guess  
5     initial_m = 0 # initial slope guess  
6     num_iterations = 1000  
7     print ("Starting gradient descent at b = {0}, m = {1}, error = {2}".  
8     print ("Running...")  
9     [b, m] = gradient_descent_runner(points, initial_b, initial_m, learn  
10    print ("After {0} iterations b = {1}, m = {2}, error = {3}".format(n  
11
```

A demo of the code is shown below that highlights how the line (in red) iteratively fits into the data (in blue) by updating the slope m and the intercept b . Notice how the slope and the intercept change over time. Note that arbitrary values for the initial values of m and b need to be picked. The figure on the left shows how the values of the slope change over time (shown in green).

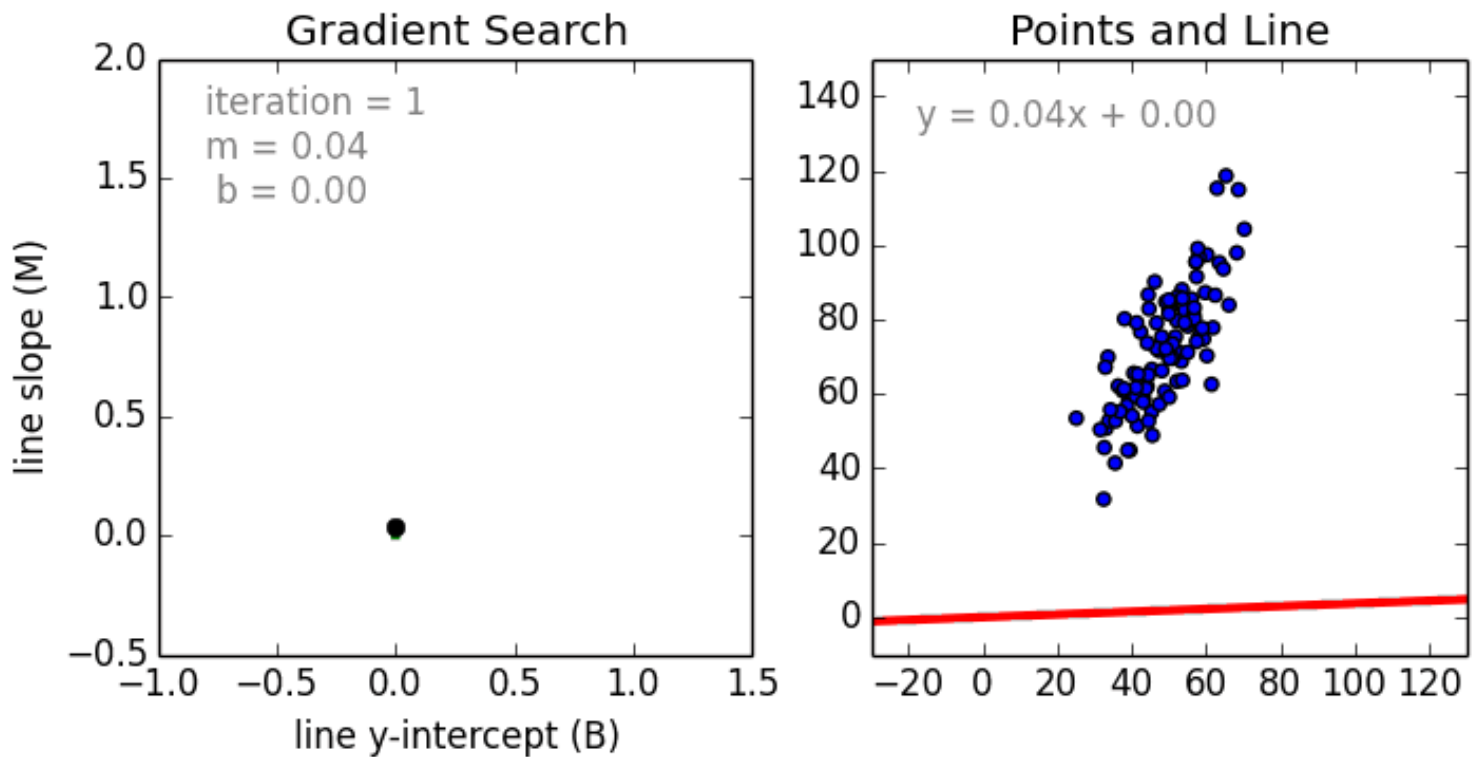


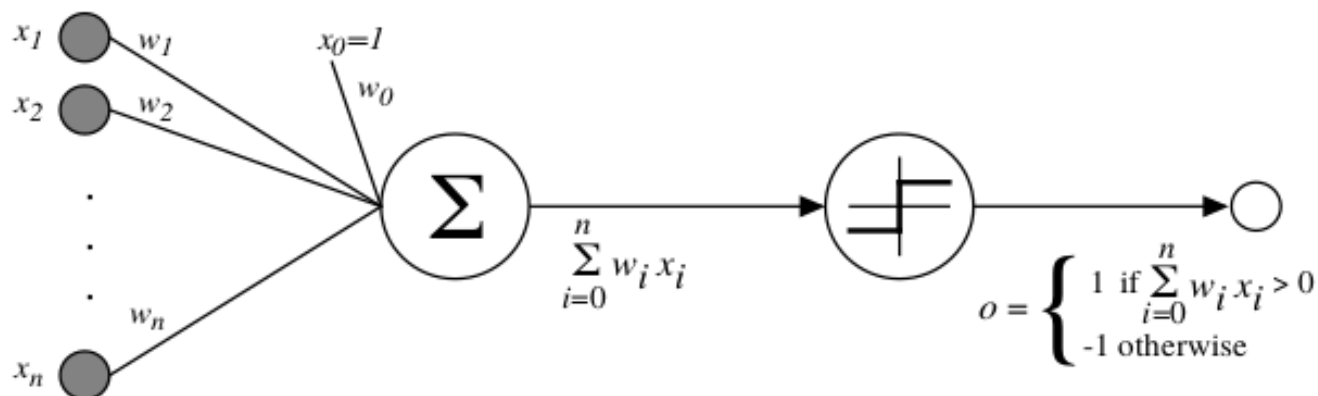
Figure: Gradient descent search on data points. Adapted from "Gradient Descent Example for Linear Regression" by Github, n.d. (<https://github.com/mattnedrich/GradientDescentExample>).

Note that all the above code is a preview. It is executable with data in the next lesson, Linear regression using Python.

The case of linear regression has a [closed form solution](#) that can be solved easily, hence gradient descent is not the best way to solve such a simple linear regression problem. However, gradient descent forms the basics of learning for machine learning and neural networks. Hence, this example was discussed so that it helps you to understand more complex models such as neural networks (week 3). In practice, gradient descent becomes more useful when you have hundreds of parameters instead of two parameters (m and b) considered in the above example. In deep learning models, you can expect millions of parameters for some large or deep neural network models.

Let's consider a case where there is not just one feature in x , but x represents a vector of features, e.g., your data set is about presence/absence of heart disease given a set of features by x_1, x_2, x_3 ; such as age, weight, and body-mass-index.

Below you can see an example of a model where the sum of the incoming links ($w_1, w_2, w_3..w_N$) over the inputs ($x_1, x_2, ...x_N$) is computed and then the output (Z) goes through an activation function. A linear activation function is simply a linear regression model which is also known as the perceptron in the neural network literature.



$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Sometimes we'll use simpler vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Figure: Activation function. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.



Now let's examine how gradient descent can be used as a tool for finding the best parameters for a given linear model.

We need a way to capture how well the data fits the linear model, hence we use an *error or cost function* known as the sum-squared-error (E). Our goal is to adjust the parameters of the linear model $(w_0, w_1, w_2, \dots, w_N)$ for the given input data (x_1, x_2, \dots, x_N) from the set of training examples denoted by D .

To understand, consider a simpler linear unit where

$$o = w_0 + w_1 x_1 + \dots + w_n x_n$$

and w_i is a parameter or weight that minimises the squared error.

The error or cost function can be written as

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

where D is a set of training examples.

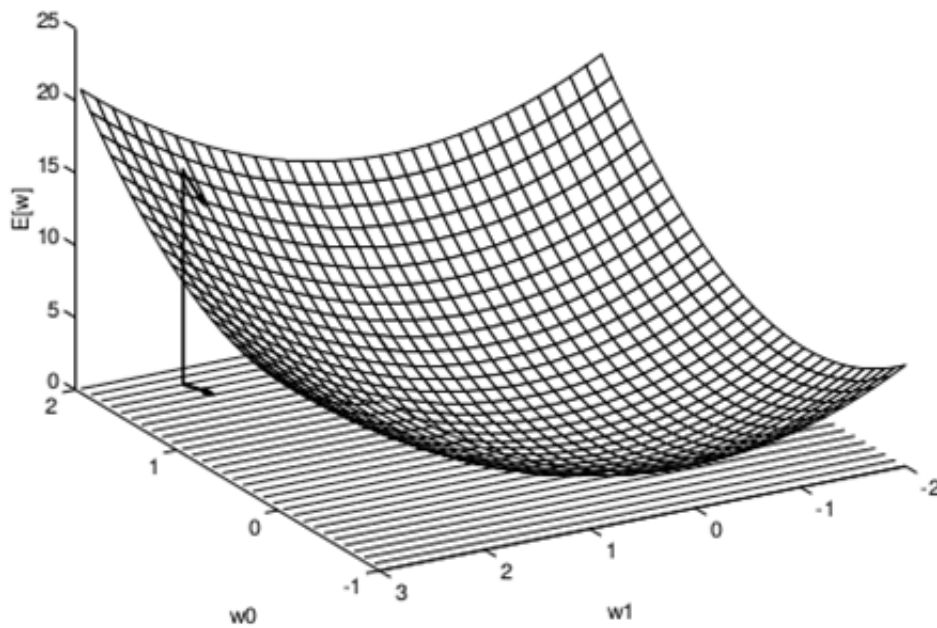


Figure: Error or cost function. Adapted from *Machine Learning* by T. Mitchell, 1997, Maidenhead; U.K: McGraw Hill.

Let's understand the derivation of the gradient descent rule. What we need to calculate is the derivative of E .

If necessary, revise partial derivatives and chain rule by clicking on the links below:

1. [Tutorial 1](#) Partial derivatives (MathisFun, n.d.)
2. [Tutorial 2](#) Partial derivatives (Khan Academy, n.d.)
3. [Tutorial 3](#) Chain rule (Khan Academy, n.d.)

Gradient of E which is a vector derivative can be written as

$$[\nabla E[\vec{w}]] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$$

The training rule for gradient decent is

$$[\Delta \vec{w} = -\eta \nabla E[\vec{w}]]$$

i.e.,

$$[\Delta w_i = -\eta \frac{\partial E}{\partial w_i}]$$

As you can see, E is differentiated with respect to the weights or parameters $w(w_1, w_1, \dots w_N)$ to find a gradient to compute the parameter update, that is delta Δw . Note that the term weights are used as this linear model is one of the building blocks of simple neural networks which will be covered later.

Now let's look at the derivation for the weight update by differentiating the error function E with respect to weights w , where the model is the linear model. Below are the equations:

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\ &= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\ \frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d) (-x_{i,d}) \end{aligned}$$

Now let's learn about the training examples. Each training example is a pair of the form $\langle \vec{x}, t \rangle$ where \vec{x} is the vector of input values, and t is the target output value. η is the learning rate(e.g., 0.5).

Initialise each w_i to some small random value. Until the termination condition is met, do the following:

- Initialise each Δw_i to zero.
- For each pair in training examples, input the vector of input values to the unit and compute the output o . For each linear unit weight w_i ,

$$\Delta w_i \leftarrow \Delta w_i + \eta (t - o) x_i$$

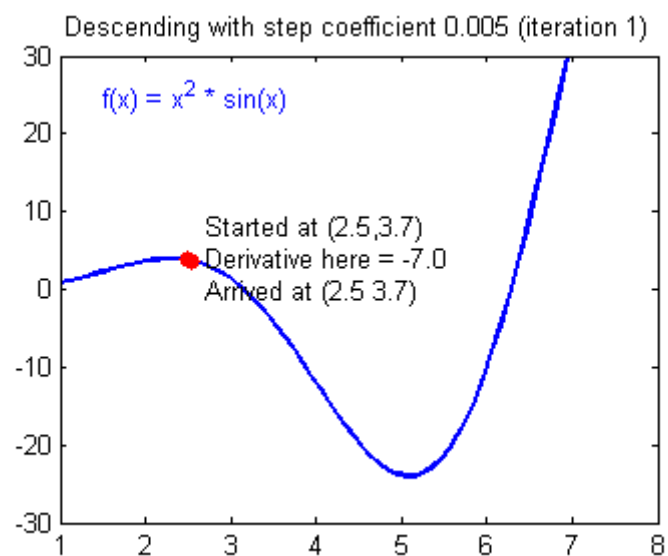
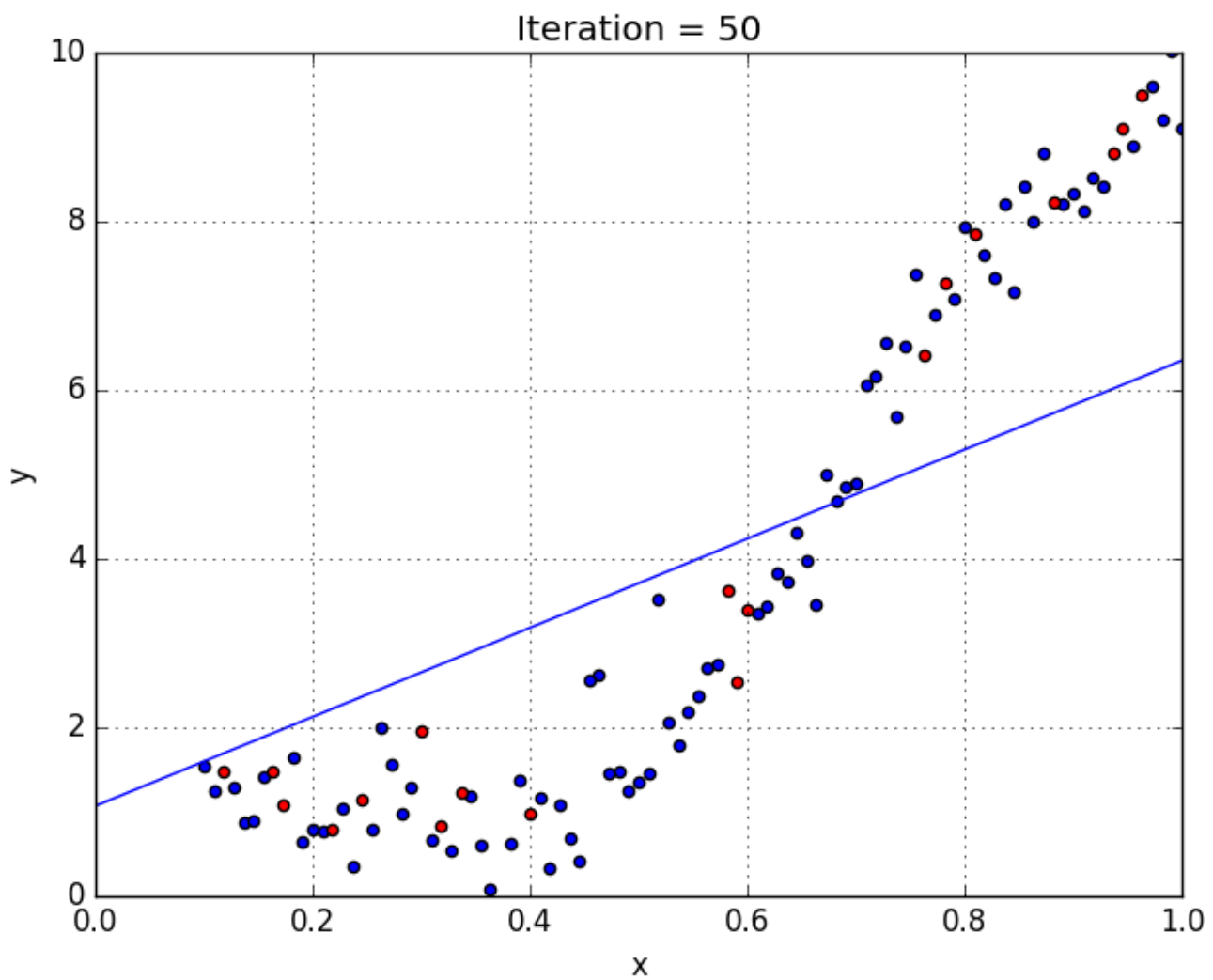
For each linear unit weight w_i ,

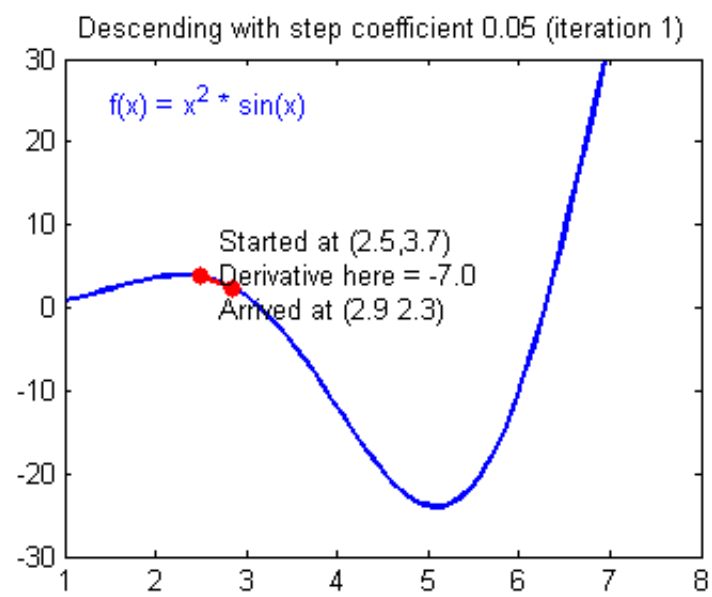
$$w_i \leftarrow w_i + \Delta w_i$$

Source: Mitchell, T. (1997). *Machine Learning*. Maidenhead; U.K. McGraw Hill.

Further information: <https://folk.idi.ntnu.no/keithd/classes/advai/lectures/backprop.pdf>

More visualization:





Source: https://www.cs.toronto.edu/~frossard/post/linear_regression/

Correlation coefficient - R Score

The correlation coefficient provides a statistical measure (given by R score) of the strength of the relationship between two variables where the R score values range between -1.0 and 1.0. The R score with a number greater than 1.0 or less than -1.0 indicates an error in the measurement. The R score of -1.0 indicates a perfect negative correlation and 1 a perfect position creation, while the R score of 0 would show no linear relationship between the two variables.

Further information about interpreting the R score is given below:

1. **Exactly -1.** A perfect downhill (negative) linear relationship
 2. **-0.70.** A strong downhill (negative) linear relationship
 3. **-0.50.** A moderate downhill (negative) relationship
 4. **-0.30.** A weak downhill (negative) linear relationship
- 0.** No linear relationship
1. **+0.30.** A weak uphill (positive) linear relationship
 2. **+0.50.** A moderate uphill (positive) relationship
 3. **+0.70.** A strong uphill (positive) linear relationship
 4. **Exactly +1.** A perfect uphill (positive) linear relationship

Source: Rumsey, D. (n.d). How to Interpret a Correlation Coefficient r. Retrieved from <https://www.dummies.com/education/math/statistics/how-to-interpret-a-correlation-coefficient-r/>.

Correlation & regression: Concepts with illustrative examples

Learn and apply. (2017, August 26). *Correlation & Regression: Concepts with Illustrative examples* [online video]. Retrieved from <https://www.youtube.com/watch?v=xTpHD5WLuoA>.

We compute the r score with equation below

$$r = \frac{\sum (x - m_x)(y - m_y)}{\sqrt{\sum (x - m_x)^2 \sum (y - m_y)^2}}$$

where x and y are two vectors and m_x and m_y represent their means, respectively. More info: <http://www.sthda.com/english/wiki/correlation-test-between-two-variables-in-r>

```
1 #source: https://stackoverflow.com/questions/3949226/calculating-pearson
2
3 import numpy as np
4 def pcc(X, Y):
5     ''' Compute Pearson Correlation Coefficient. '''
6     # Normalise X and Y
7     X -= X.mean(0)
8     Y -= Y.mean(0)
9     # Standardise X and Y
10    X /= X.std(0)
11    Y /= Y.std(0)
12    # Compute mean product
13    return np.mean(X*Y)
14
```

The above code shows a python implementation of R score. Another code example in R is given below.

R-Squared Score

The R-squared score is another way to measure the strength of the correlation between two variables, which has more advantages for larger or more complex linear models. R score is appropriate for simple linear regression models, where one x and one y variable are concerned, but for the case when the model becomes more complex with more than one x variable, then R squared becomes more appropriate.

R-squared is the percentage of the response variable variation that is explained by a linear model.

R-squared = Explained variation / Total variation

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

where, y and \hat{y} are the actual and prediction vectors and \bar{y} is the mean of y .

More information

R-squared is always between 0 and 100% and:

1. 0% indicates that the model explains none of the variability of the response data around its

mean.

2. 100% indicates that the model explains all the variability of the response data around its mean.


In general, the higher the R-squared, the better the model fits your data.

Minitab. (2013). Regression Analysis: How Do I Interpret R-squared and Assess the Goodness-of-Fit. Retrieved from <https://blog.minitab.com/blog/adventures-in-statistics-2/regression-analysis-how-do-i-interpret-r-squared-and-assess-the-goodness-of-fit>.

More information is in the video below:

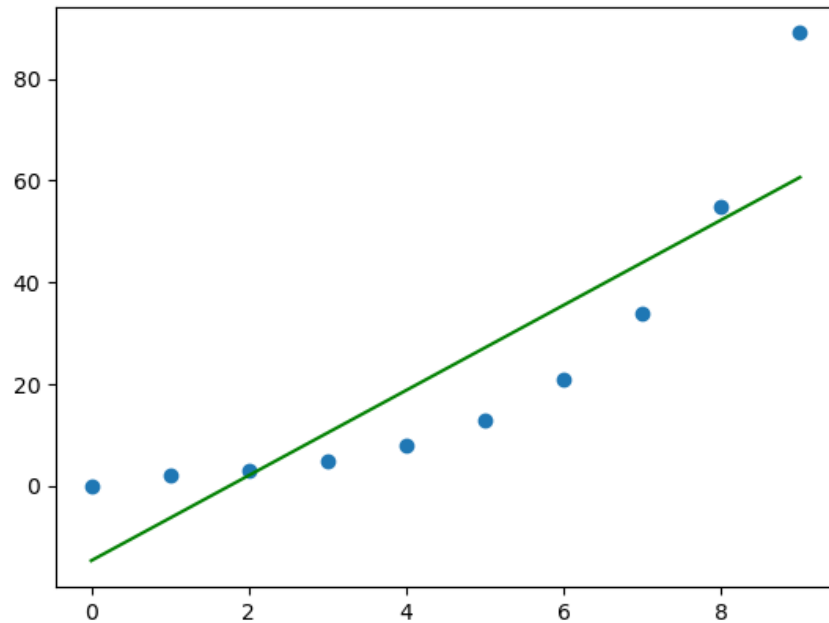
StatQuest. (2015, February 03). *StatQuest: R-squared explained* [online video]. Retrieved from <https://www.youtube.com/watch?v=2AQKmw14mHM>.

Example code is given below to demonstrate the concept using sklearn library.

```
▶ Run PYTHON 
```

```
1
2 #source for this code: https://towardsdatascience.com/r-squared-recipe-5
3 #importing matplotlib inline
4
5 import numpy as np
6 from sklearn.metrics import r2_score
7 import matplotlib.pyplot as plt
8 from scipy import stats
9
10 #creating data
11 x = np.array([0,1,2,3,4,5,6,7,8,9])
12 y = np.array([0,2,3,5,8,13,21,34,55,89])
13
14 #creating OLS regression
```

The output of code above is given below.



▶ Run

PYTHON



```
1 #source https://stackoverflow.com/questions/893657/how-do-i-calculate-r-
2
3 from __future__ import print_function, division
4 import sklearn.metrics
5 import numpy as np
6
7 def compute_r2_weighted(y_true, y_pred, weight):
8     sse = (weight * (y_true - y_pred) ** 2).sum(axis=0, dtype=np.float64)
9     tse = (weight * (y_true - np.average(
10         y_true, axis=0, weights=weight)) ** 2).sum(axis=0, dtype=np.floa
11     r2_score = 1 - (sse / tse)
12     return r2_score, sse, tse
13
14 def compute_r2(y_true, y_predicted):
```

The code above gives further information about implementation from scratch using Python and code below gives implementation in R from scratch.

```
1 #Source for code: https://stackoverflow.com/questions/40901445/function-
2
3 preds <- c(1, 2, 3)
4 actual <- c(2, 2, 4)
5
6 rss <- sum((preds - actual) ^ 2) ## residual sum of squares
7 print(rss)
8 tss <- sum((actual - mean(actual)) ^ 2) ## total sum of squares
9 print(tss)
10 rsq <- 1 - rss/tss
11
12 print(rsq)
13
14
```