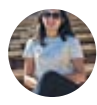


BACK TO BASICS IN ML

Implementing Linear Regression From Scratch


A walkthrough on implementing Simple and Multiple Linear Regression from Scratch in Python

 Priti Oli · Follow

10 min read · Jan 3, 2024

 135

 2







Linear regression is a widely used statistical technique for predicting a continuous outcome variable based on one or more predictor variables. While there are numerous libraries and tools available that offer ready-made implementations of linear regression, understanding how to implement it from scratch provides valuable insights into the underlying mathematics and concepts. In this article, we will walk through the process of implementing linear regression from scratch using Python.

Understanding Linear Regression

Linear regression aims to establish a linear relationship between the input variables (features) and the output variable (target). The goal is to find the best-fitting line that minimizes the difference between predicted and actual values. Mathematically, this line can be represented as:

$$y = wx + b \dots\dots\dots(eq\ 1)$$

where y represents the *dependent variable (output)*, x is the input or *independent variable*, w signifies the *weight* or *parameter vector* of the approximation function (i.e slope in a simple linear equation) and b denotes the *bias* of the approximation function(y-intercept in a simple approximation) and is also known as the *error term*.

Simple Linear Regression:

Simple Linear Regression specifically refers to the case where there is only one independent variable (x) predicting a single dependent variable (y). The relationship between x and y is modeled as a straight line, represented by the equation:

$$y=mx+b\dots\dots\dots eq(2)$$

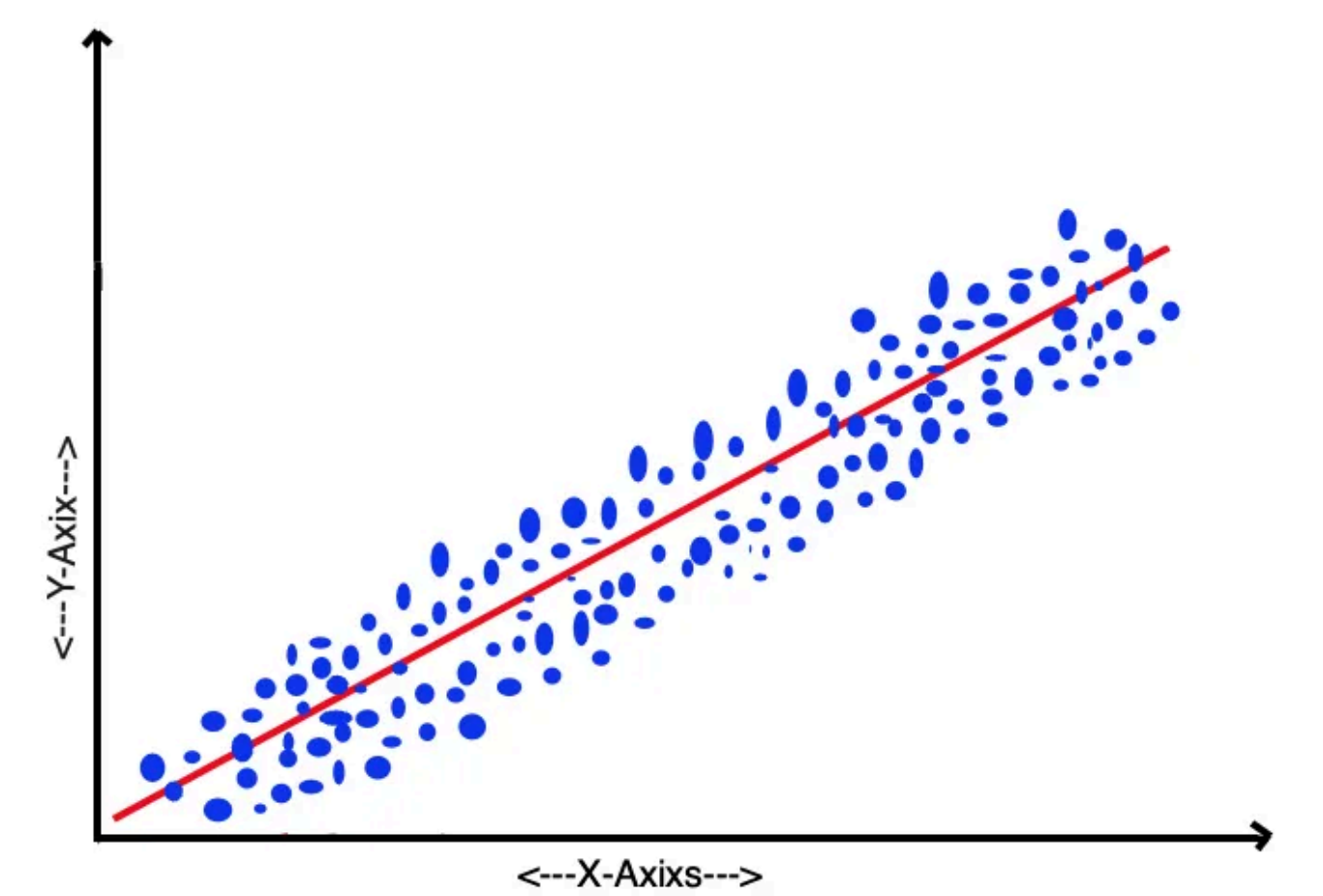


Fig: Simple Linear Regression where the blue dots represents the observed values and the red line indicates the predicted values after linear regression.

The linear regression model aims to find the optimal values for w and b that minimize the difference between the predicted values and the actual observations. The weight (w) governs the influence of the input variable,

determining the slope of the fitted line, while the bias (b) allows for an adjustment of the line's position on the y-axis. The interplay of these components forms the basis for constructing an effective linear regression model that accurately captures the underlying relationships within the data.

Some of the practical uses of using Simple Linear Regression include

- Predicting a students' exam score based on the number of hours they spent studying.
- Predicting temperature based on the number of hours of sunlight in a day.

Multiple Linear Regression

Multiple regression represents a broad category of regression analyses that include both linear and nonlinear regression models, incorporating multiple explanatory variables. **Multiple Linear Regression** is an extension of Simple Linear Regression, designed to model the relationship between a dependent variable (y) and multiple independent variables $X(x_1,x_2,...,x_1,x_2,...,x_n)$. In Multiple Linear Regression, the relationship is expressed through a linear equation that includes multiple predictors, allowing for a more realistic representation of complex real-world scenarios. The formula features multiple slopes (one for each variable x_i) and a shared y-intercept, that considers the impact of multiple variables on the relationship's slope. *Eq. 1* can be expanded as

$y = w_0 + w_1 * x_1 + w_2 * x_2 + ... + w_n * x_n + b.....eq(3)$

Multiple linear regression is used when several independent variables collectively influence a single dependent variable which allows for the incorporation of multiple factors that influence the dependent variable, providing a more realistic representation of complex relationships.

Some practical use cases of using Multiple Linear Regression include

- Predicting the price of a house based on multiple features such as square footage, number of bedrooms, and location.
- Predicting sales based on factors like advertising spending, store size, and promotional events.

Now that we understand Linear Regression, Simple Linear Regression, and Multiple Linear Regression let's dive deep into the implementation of the two. There are two ways to implement a Simple Linear Regression: **Ordinary Least Squares (OLS)** or **Gradient Descent** but they differ in their approaches to finding the optimal parameters for the model.

Ordinary Least Squares

OLS is a linear regression method that employs a closed-form solution, calculating optimal coefficients to analytically minimize the sum of squared differences between predicted and actual values. This method involves finding the coefficients using linear algebra operations and is well-suited for smaller datasets, where computational efficiency is less critical.

The prerequisites for Ordinary Least Squares (OLS) include an *approximately linear relationship, observations are independent, a normal distribution of residuals, no perfect multicollinearity among independent variables, and no correlation between independent variables and residuals*. Meeting these assumptions ensures the reliability of OLS estimates.

OLS is preferred in precise estimation and interpretability, evident in applications like housing price estimation and economic growth modeling, where closed-form solutions and the economic significance of coefficients are crucial.

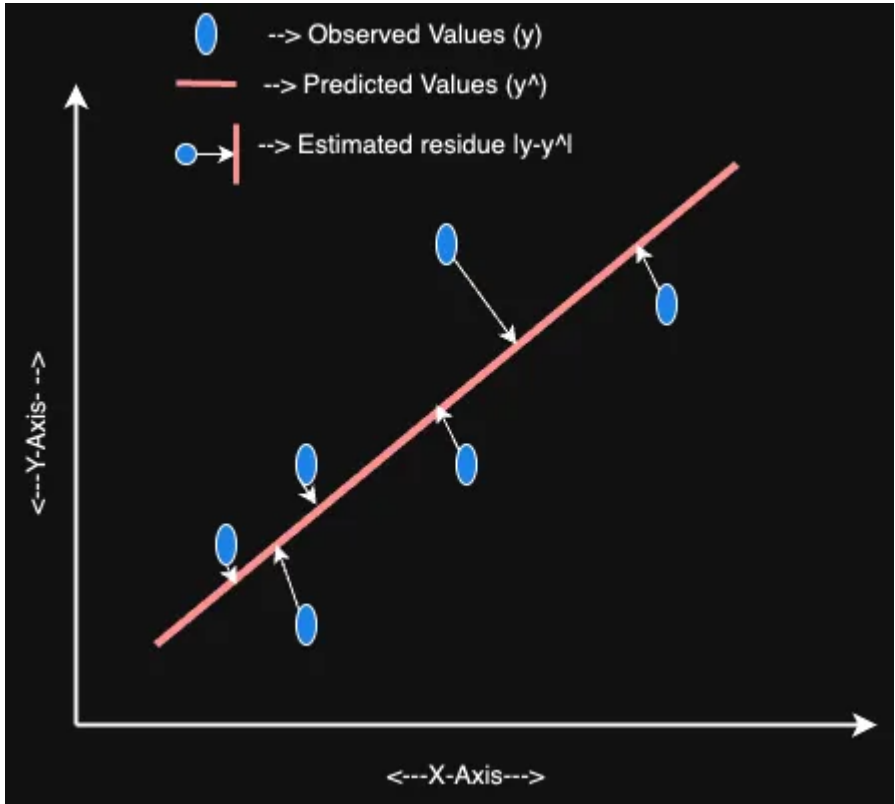


Fig : OLS is used to find the best-fitting line(y^{\wedge}) by minimizing the estimated residue or error ($y-y^{\wedge}$) i.e sum of squared differences between observed (y) and predicted values(y^{\wedge}) in linear regression

In linear regression using Ordinary Least Squares (OLS), the objective is to find the coefficients (slope and intercept) that minimize the sum of squared differences between the observed values (y) and the predicted values (y^{\wedge}). The linear regression model is represented as:

$$y_i^{\wedge} = w_0 + w_1 * x_i + \epsilon_i \dots \dots \dots eq(4)$$

where,

- x_i is the value of the independent variable for the i -th observation.
- w_0 is the y-intercept (constant term).
- w_1 is the slope coefficient.
- ϵ_i is the error term representing the difference between the observed and predicted values ($y_i - y_i^{\wedge}$) for the i -th observation.

The objective is to minimize the sum of squared errors: Minimize : $\sum (y_i - y_i^{\wedge})^2$

$$\begin{aligned} \text{Minimize : } & \sum_{i=1}^n \epsilon_i^2 \\ w_1 = \frac{\text{covariance}}{\text{variance}} &= \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} \\ w_0 &= \bar{y} - \hat{w}_1 \bar{x} \end{aligned}$$

where, \bar{x} is the mean of the independent variable, and \bar{y} is the mean of the dependent variable.

Implementing Linear Regression using Ordinary Least Squares

- Write a function to calculate RMSE (Root Mean Square Error)

```
from math import sqrt
import numpy as np

def rmse_metric(actual, predicted):
    sum_error = 0.0
    for y, y_hat in zip(actual,predicted) :
        prediction_error = y - y_hat
        sum_error += (prediction_error ** 2)
    mean_error = sum_error / float(len(actual))
    return sqrt(mean_error)
```

- Write functions to compute the coefficients of OLS (slope and intercept) by computing the covariance(numerator) and variance(denominator)

```
def compute_coefficient(x, y):
    n = len(x)
    x_mean = np.mean(x)
    y_mean = np.mean(y)

    numerator = 0
    denominator = 0

    for i in range(n):
        numerator += (x[i] - x_mean) * (y[i] - y_mean)
        denominator += (x[i] - x_mean) ** 2

    # Calculate coefficients
    slope = numerator / denominator
```

```
intercept = y_mean - slope * x_mean

return slope, intercept
```

- Write a function to predict the value of y

```
def predict(x, w1, w0):
    return w1 * x + w0
```

Let’s create a toy dataset where x spans from 1 to 50. The relationship between x and y is defined with a slope of 3 and an intercept of 5. To introduce subtle variations, random values within the range of -4 to 4 are added to random indexes in y.

```
x = np.arange(1, 51)
y = x*3+5

# Add some random error to the array
y[np.random.randint(0, len(y), size=10)] += np.random.randint(-5, 5)
```

Now, let us predict the value of y by calling the function to compute the coefficients and then the function to predict the value of y. We observe that our slope and intercept are predicted well by inspecting the value of the coefficients.

```
w1, w0 = compute_coefficient(x, y)
y_hat = predict(x,w1,w0)
# display the value of predicted coefficients
print(w1,w0)
```

Let’s evaluate our OLS algorithm using the Root Mean Square Error

```
def evaluate_ols(y,y_hat):
    mse = np.mean((y - y_hat) ** 2)
    return mse,np.sqrt(mse)
print(evaluate_ols(y,y_hat))
```

Let’s plot the observed value and predicted value using matplotlib

```
import matplotlib.pyplot as plt

plt.scatter(x, y, label='Observed Value')
plt.plot(x, y_hat, label='Predicted Value', color='red')
plt.xlabel('<--X-Axis-->')
plt.ylabel('<--Y-Axis-->')
plt.legend()
plt.show()
```

Gradient Descent

Gradient Descent is an iterative optimization algorithm used to find the minimum of a function by adjusting parameters in the direction of the steepest descent of the gradient, aiming to minimize the cost or error associated with a model’s predictions. Gradient descent is used in fields such as web analytics, online advertising, and large-scale machine-learning applications.

Gradient Descent is well-suited for dynamic and large-scale environments, such as online retail sales prediction, social media engagement prediction, and weather forecasting, where continuous model adjustments are essential for accuracy. Additionally, Gradient descent is also adaptable for nonlinear

regression in complex relationships, finding utility in fields such as biology, chemistry, and physics.

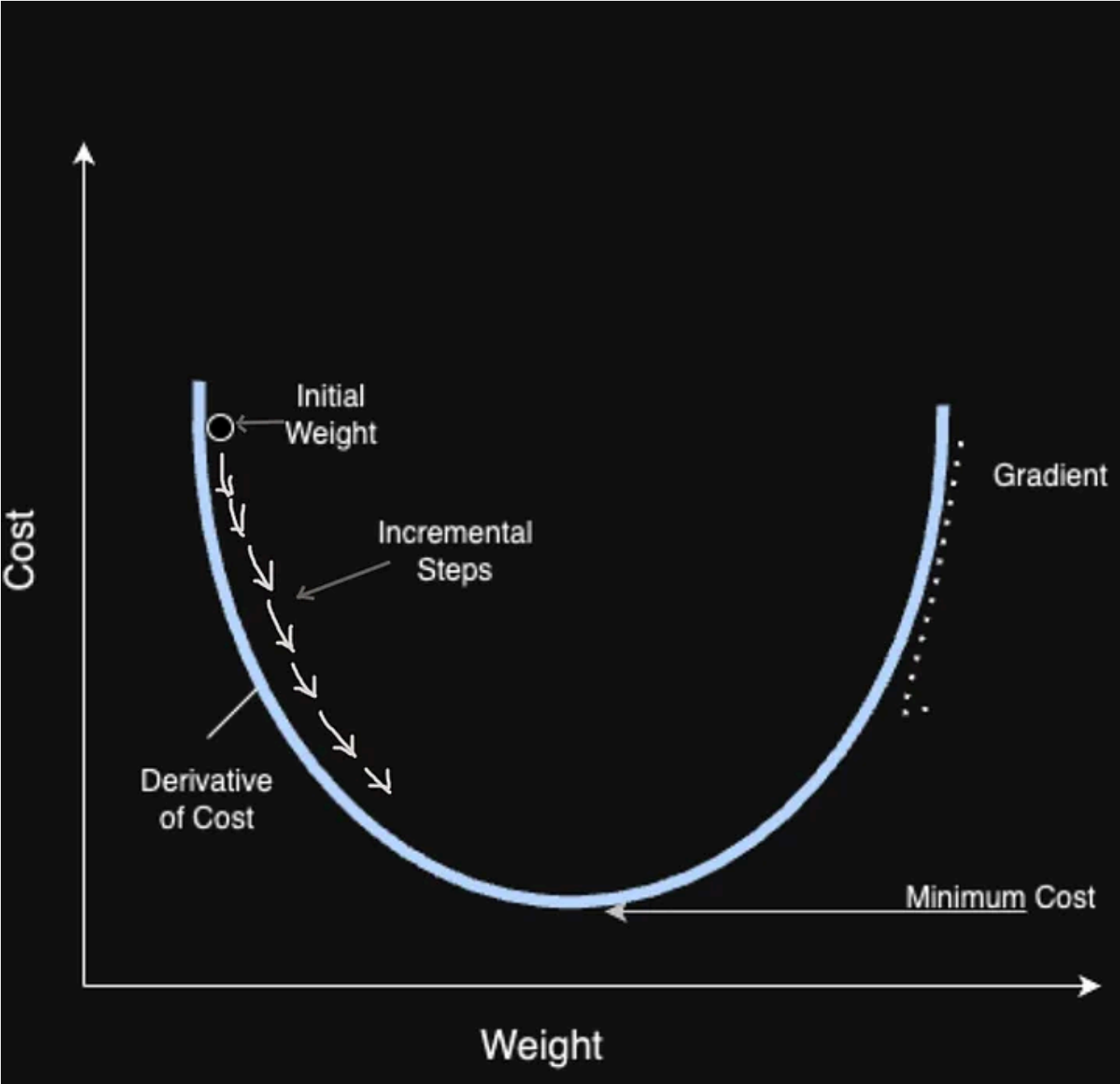


Fig: Gradient Descent using iterative optimization method, adjusting model parameters (weight in the fig.) in the direction of decreasing gradient to find the steepest descent i.e to minimize the cost function.

Linear regression using the gradient descent algorithm involves minimizing the cost function by iteratively updating the model parameters slope(w_1) and y-intercept (w_0). The cost function (also known as the loss function) measures the difference between predicted and actual values. Here’s a step-by-step explanation using mathematical expressions:

Linear Regression Model:

The linear regression model is represented as:

$y_i^{\wedge} = h(x) = w_0 + w_1 * x_i,.....eq(5)$

where,

- y_i^{\wedge} or $h(x)$ is the predicted value.
- w_0 is the y-intercept.
- w_1 is the slope coefficient.

Cost Function: The cost function is defined as:

$$J(w_0, w_1) = \frac{1}{2m} \sum_{i=1}^m (h(x^{(i)}) - y^{(i)})^2$$

where, $J(w_0, w_1)$ is the cost function, m is the number of observations and $x^{(i)}$ and $y^{(i)}$ are the independent and dependent variables for the i-th observation.

...

Gradient Descent Update Rule: The gradient descent algorithm minimizes the cost function by iteratively updating the parameters using the following update rule.

$$w_0 = w_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h(x^{(i)}) - y^{(i)})$$

$$w_1 = w_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

where, α is the learning rate, controlling the step size in the update.

The learning rate(α) is a hyperparameter in gradient descent that determines the step size taken during each iteration to update model parameters and influence the convergence speed and stability of the algorithm. *Choosing an appropriate learning rate is crucial.* If the learning rate is too small, the algorithm may take a long time to converge, or it may get stuck in local minima. On the other hand, if the learning rate is too large, the algorithm may overshoot the minimum, potentially failing to converge or oscillating around the optimal values. The learning rate is often set empirically through experimentation and hyperparameter tuning, balancing the trade-off between convergence speed and stability. Common choices for learning rates include 0.1, 0.01, and 0.001, but the optimal value can vary depending on the specific problem and dataset.

Iterative Process:

Repeat until convergence:

1. Update $w1$ and $w0$ using the gradient descent update rule.
2. Calculate the cost function $J(w0,w1)$.
3. Check for convergence or a predefined number of iterations.

Implementing Linear Regression using Gradient Descent

- Randomly initialize the weights($w1$) and biases($w0$). The weights and biases will be updated during each iteration of the gradient descent to minimize the loss.

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

def initialize(dim):
    w1 = np.random.rand(dim)
    w0 = np.random.rand()
    return w1, w0
```

- Compute the cost i.e the difference between the predicted and observed value of y

```
def compute_cost(X,Y, y_hat):
    m = len(Y)
    cost = (1/(2*m)) * np.sum(np.square(y_hat - Y))
    return cost
```

- Predict the value of y using weights and biases for the input X

```
def predict_y(X,w1,w0):
    if len(w1)==1:
        w1 = w1[0]
    return X*w1+w0
return np.dot(X,w1)+w0
```

- Update the parameters ($w1,w0$)

```
def update_parameters(X,Y,y_hat,cost,w0,w1,learning_rate):
    m = len(Y)
    db=(np.sum(y_hat-Y))/m
    dw=np.dot(y_hat-Y,X)/m

    w0_new=w0-learning_rate*db
    w1_new=w1-learning_rate*dw
    return w0_new,w1_new
```

- Let’s implement the gradient descent function by iteratively updating the weights and biases to minimize the cost function. We’ll set a maximum threshold for the number of iterations or terminate the iteration when the specified early stopping criterion is met. In our case, the stopping criterion is defined by the difference between the previous and current

values of the cost function falling below a specified threshold. Throughout the iterations, we'll update the cost at each step, allowing us to visualize the trend and assess any issues related to vanishing or exploding gradients.

```
def run_gradient_descent(X,Y,alpha,max_iterations,stopping_threshold = 1e-6):
    dims = 1
    if len(X.shape)>1:
        dims = X.shape[1]
    w1,w0=initialize(dims)
    previous_cost = None
    cost_history = np.zeros(max_iterations)
    for itr in range(max_iterations):
        y_hat=predict_y(X,w1,w0)
        cost=compute_cost(X,Y,y_hat)
        # early stopping criteria
        if previous_cost and abs(previous_cost-cost)<=stopping_threshold:
            break
        cost_history[itr]=cost
        previous_cost = cost
        old_w1=w1
        old_w0=w0
        w0,w1=update_parameters(X,Y,y_hat,cost,old_w0,old_w1,alpha)

    return w0,w1,cost_history
```

- Write a function to run gradient descent

```
def run_gradient_descent(X,Y,alpha,num_iterations):
    w0,w1=initialize(X.shape[1])
    num_iterations=0
    gd_iterations_df=10
    for itr in range(num_iterations):
        Y_hat=predict_y(w0,w1,X)
        cost=compute_cost(Y,Y_hat)
        old_w0=b
        old_w0=theta
        w0,w1=update_theta(X,Y,Y_hat,old_w0,old_w0,alpha)

    return gd_iterations_df,w0,w1
```

Now let’s put it all together and run the algorithm on a toy dataset as shown below.

```
X = np.array([32.50234527, 53.42680403, 61.53035803, 47.47563963, 59.81320787,
55.14218841, 52.21179669, 39.29956669, 48.10504169, 52.55001444,
45.41973014, 54.35163488, 44.1640495 , 58.16847072, 56.72720806,
48.95588857, 44.68719623, 60.29732685, 45.61864377, 38.81681754])
Y = np.array([31.70700585, 68.77759598, 62.5623823 , 71.54663223, 87.23092513,
78.21151827, 79.64197305, 59.17148932, 75.3312423 , 71.30087989,
55.16567715, 82.47884676, 62.00892325, 75.39287043, 81.43619216,
60.72360244, 82.89250373, 97.37989686, 48.84715332, 56.87721319])
```

Define the hyper-parameters like learning rate and maximum number of iterations and call the gradient descent function.

```
learning_rate = 0.0001
iterations = 10
print(X.shape,Y.shape)

w0,w1,cost_history = run_gradient_descent(X,Y,learning_rate,iterations)
print(w0,w1)
```

Let’s plot the cost function to visualize the loss during each iteration.

```
import numpy as np
import matplotlib.pyplot as plt
# Plot the cost history
plt.plot(range(1, iterations + 1), cost_history, color='blue')
plt.rcParams["figure.figsize"] = (10, 6)
plt.grid()
plt.xlabel('Number of iterations')
plt.ylabel('Cost (J)')
plt.title('Convergence of gradient descent')
plt.show()
```

Let’s explore another example for running the linear regression implementation using gradient descent, this time with a two-dimensional feature set (x). In this case, each row of x contains two columns of data, making it a dataset with two features for each sample

```
# Set hyperparameters
learning_rate = 0.0001
iterations = 1000

# Set random seed for reproducibility
np.random.seed(42)

# Number of samples
m = 200

# generate random features X with two dimensions
X = 10 * np.random.rand(m, 2) # Scaling by 10 for variety


# generate target variable y with some constant weights and bias
true_weights = np.array([3, 4])
bias = 2
Y = X.dot(true_weights) + bias + np.random.randn(m) # adding some random noise

w0,w1,cost_history = run_gradient_descent(X,Y,learning_rate,iterations)
print(w0,w1)
```

Let’s plot the cost function to visualize the loss during each iteration

```
plt.plot(range(1, iterations + 1), cost_history, color='blue')
plt.rcParams["figure.figsize"] = (10, 6)
plt.grid()
plt.xlabel('Number of iterations')
plt.ylabel('Cost (J)')
plt.title('Convergence of gradient descent')
plt.show()
```

- Towards Data Science
- Machine Learning
- Python
- Linear Regression
- Linear Regression Python




Written by Priti Oli

120 Followers · 70 Following

Computer Science Ph.D. graduate exploring the frontiers of Artificial Intelligence and Human Computer Interaction


Follow

Responses (2)



What are your thoughts?

Respond




Sagar Pathak


Jan 3, 2024

...

Nice article

 4

[Reply](#)





Joe Vitor Silva

Jan 3, 2024

...

Wonderful article on LR! Simple to read and understand! Congratulations! Happy New Year of 2024, from Brazil!

 1

 1 reply

[Reply](#)

https://medium.com/@pritioli/implementing-linear-regression-from-scratch-747343634494

8/10

More from Priti Oli



Priti Oli

20 Basic Operations using NumPy
A Complete Refresher on NumPy Libraries for ML Enthusiasts

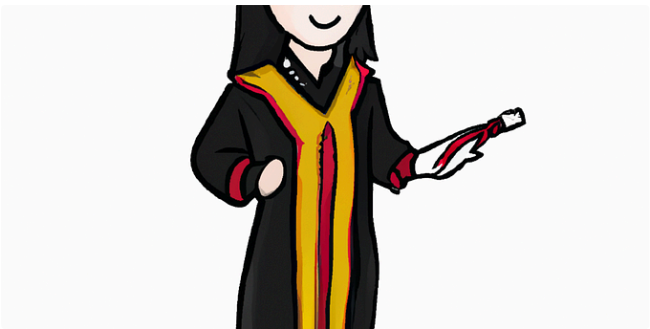
Dec 26, 2023 56



Priti Oli

Essential NumPy Data Types: A Must-Know Guide
A Complete Refresher on Numpy DataType for Data Science Enthusiasts

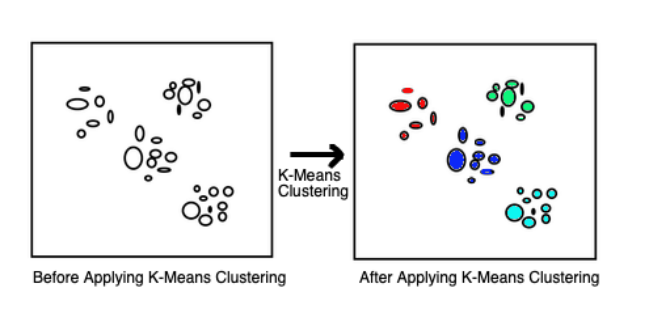
Dec 22, 2023 81



Priti Oli

Beyond Academia: 10 Profound Life Lessons I Learned Through M...
Embarking on the journey of a Ph.D. isn't just an academic endeavor; it's a transformative...

Dec 29, 2023 72 2



Priti Oli

Implementing K-Means Clustering From Scratch
A walkthrough on implementing K-Means Clustering from Scratch in Python

Dec 31, 2023 3

See all from Priti Oli

Recommended from Medium



In All About Algorithms by Dr. Robert Kübler

Finding Closest Points Faster Than $O(n^2)$
An interesting application of the divide-and-conquer paradigm

Jan 5, 2024 462 6

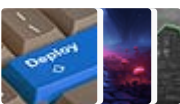


Piyush Kashyap

Understanding SGD with Momentum in Deep Learning: A...
Hello, deep learning enthusiasts! In this article, we'll dive into the optimization...

Nov 2, 2024 2

Lists



Predictive Modeling w/ Python
20 stories · 1836 saves



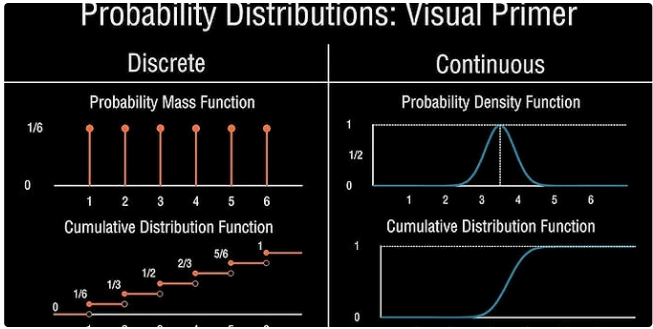
Coding & Development
11 stories · 1014 saves



Practical Guides to Machine Learning
10 stories · 2212 saves



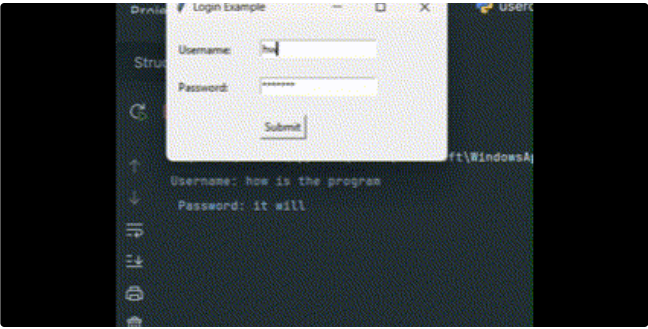
Natural Language Processing
1955 stories · 1602 saves



CodexRushi

Statistics Checklist Before Going for a Data Science Interview
Data science interviews often test your knowledge of statistics, as it's the foundatio...

Feb 11 54 2



Shovit Kafle

Python Tkinter Module—GUI
A short and worth-it documentation on Tkinter to clear all our doubts. Contains som...

Oct 30, 2024 851



Ai In AI Made Simple by Dr. Roi Yehoshua

Understanding Polynomial Regression
In the previous article we discussed linear regression and its variants. In many problem...

Apr 28, 2023 121 5

Capacity = 2	Capacity = 3	Capacity = 4	Capacity = 5	Capacity = 6	Capacity = 7	Capacity = 8	Capacity
0	0	0	0	0	0	0	0

Rohollah

An Introduction to Dynamic Programming: Step-by-Step Guide
One of the most famous problems in the Divide and Conquer approach is the...

Sep 21, 2024 2

See more recommendations