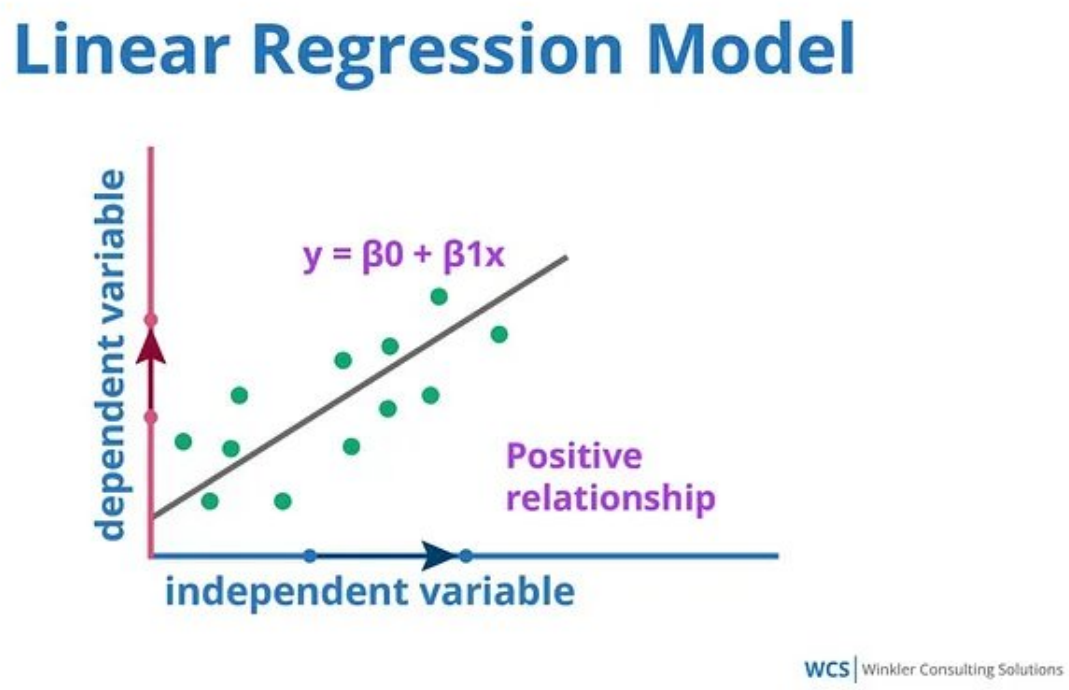
Linear Regression Made Simple: A First Step into ML

P santhoshini · Follow
8 min read · Just now



This blog is for anyone new to machine learning algorithms, taking their first step into this exciting field. I'll make this blog a little longer to ensure you clearly understand how everything is built. Since this is the foundation, it's important to make it solid and concrete.

There are many machine learning algorithms, each with a unique purpose. When I started learning, I had so many questions — some seemed simple, but understanding them was key to moving forward.

Through this blog, I'll address common doubts and make learning ML algorithms easier and more enjoyable for you!

Why should we learn ML algorithms?

Machine Learning (ML) algorithms are like tools designed to extract meaningful patterns from data. They determine how a model analyzes the data and learns from it. These algorithms enable machines to identify patterns, classify information, and understand the data, ultimately helping them predict outputs.

In essence, the algorithm acts as an **analyzer** that processes the data identifies relationships, and learns how to make decisions based on the given inputs. This is what allows machines to learn and adapt to various tasks.

Linear regression is a type of supervised learning.

B/W — What is supervised learning?

Supervised learning is a type of machine learning where we provide both input data (X) and corresponding output values (Y). The goal is to teach the model to learn the relationship between the inputs and the outputs.

For example, imagine you have data about 100 houses, each with a different size (input) and its corresponding price (output). In supervised learning, we train the model by showing it these pairs of input and output data, allowing it to understand the relationship between house size and price.

Once the model is trained, you can give it the size of a new house, and it will predict the price based on what it has learned from the previous data.

$h\theta(x) = \theta 0 + \theta 1\, x1$

Here h(x) is called the hypothesis, to learn a function h: $X \rightarrow Y$ so that h(x) is a "good" predictor for the corresponding value of y.

if only one feature such as (the size of the house) then only x1 is used, If there are more features like no of bedrooms and garden length, then there will be x1,x2, and so on. We consider x0 as 1 for conciseness. Since we are summing the whole thing our updated equation would be :

$$h(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x,$$

The doubt I had was during the summation: While I know that (x) represents the features, I didn't understand the value of θ at first. They told me that θ is called a parameter, but what exactly is a parameter? Let's take an example. Imagine you're predicting the price of a house based on its size. The **features (x)** are the house sizes in square feet, like x=[1000,1500,2000]x = [1000, 1500, 2000]x=[1000,1500,2000].

The **parameters (θ)** determine how much each feature affects the price. Initially, θ might start as something like θ=[0.5], which means the model starts with the assumption that for each square foot of house size, the price increases by 0.5. The **goal** is to adjust these parameters θ(e.g., θ=[0.5] to something like θ=[1.5] so that the model's predictions closely match the actual house prices. During training, **gradient descent** helps update θ\thetaθ step by step, reducing the error between predicted and real prices, until it finds the best parameters that make the model accurate.

Why θT * x?

In vector notation, **θT * x** represents the **dot product** between two vectors:

- **θ = [θ1, θ2, ..., θn]**: The vector of weights (or parameters) of the model.
- **x = [x1, x2, ..., xn]**: The vector of input features for a single data point.

> *Point to be noted:*

If you have **n features(size, bedrooms,...)**in your data, the corresponding space is **n-dimensional**. However, if you include a bias(x0) term, it effectively adds an extra dimension, making it **n+1 dimensions**.

- With **2 features**, the data is in a **2D space**, but adding the bias term makes it **3D** (this is **2 + 1**).
- With **3 features**, the data is in a **3D space**, and adding the bias term makes it **4D** (this is **3 + 1**).

to get a perfectly trained model, our hypothesis at least should be nearer to the output Y. To formalize this, we will define a function

that measures, for each value of the θ's, how close the h(x)'s are to the corresponding y 's. We define the cost function:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2.$$

- **J(θ)**: This is the **cost function**, representing the "error" or "loss" of the model. We want to minimize this function to make our model better.
- **m**: The total number of data points in the training set.
- $\mathbf{h_x(x_i)}$: This is the hypothesis function (or model), which predicts the output for the given input data.
- $\mathbf{y_i}$: This is the actual value (the true output)
- The term $\mathbf{(h_x(x_i) - y_i)}$ represents the **error** or difference between the predicted value and the actual value for the i-th data point.
- The squared error $\mathbf{(h_x(x_i) - y_i)^2}$ is used to ensure that both positive and negative errors contribute equally to the cost (i.e., we avoid canceling out errors that are positive and negative).
- **1/2**: Makes the math simpler. The factor of **1/2** is included for convenience when differentiating the cost function later in the gradient descent process (it cancels out the 2 when differentiating).

Here our final aim is to reduce the cost function, make the model learn to predict things and get an efficient output.

When learning about linear regression, you might wonder: if linear regression is already a great way to fit a model to data, why do we need something like the **Least Mean Squares (LMS)** algorithm?

Here's the answer: While **linear regression** is excellent for working with a fixed dataset (where you can compute everything all at once), it has a limitation. What if your data is constantly changing or too large to process all at once? This is where the **LMS algorithm** comes in.

Think of **linear regression** as a one-time process where you gather all your data, plug it into an equation, and find the best-fit line. It's perfect for when the data is ready, and you're not expecting any new data soon.

LMS is designed for situations where data is continuously arriving or if the dataset is too large to handle all at once. Instead of redoing all the calculations from scratch every time new data comes in, **LMS** updates the model **step by step** with each new data point. It's like making small adjustments as you go, which allows the model to adapt in real-time.

LMS algorithm- least mean squares

We want to choose θ so as to minimize J(θ). To do so, let's use a search algorithm that starts with some "initial guess" for θ, and that repeatedly changes θ to make J(θ) smaller, until hopefully, we converge to a value of θ that minimizes J(θ). Specifically, let's consider the gradient descent algorithm, which starts with some initial θ, and repeatedly performs the update:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta).$$

In order to implement this algorithm, we have to work out what is the partial derivative term on the right hand side. Let's first work it out for the case of if we have only one training example (x, y), so that we can neglect the sum in the definition of J. We have:

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} \left(h_\theta(x) - y\right)^2 \\
&= 2 \cdot \frac{1}{2} \left(h_\theta(x) - y\right) \cdot \frac{\partial}{\partial \theta_j} \left(h_\theta(x) - y\right) \\
&= \left(h_\theta(x) - y\right) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^{n} \theta_i x_i - y \right)
\end{aligned}$$

$$= \quad (h_\theta(x) - y)\, x_j$$

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}.$$

The rule is called the LMS update rule (LMS stands for "least mean squares"), and is also known as the Widrow-Hoff learning rule. We'd derived the LMS rule for when there was only a single training
example. There are two ways to modify this method for a training set of more than one example. The first is to replace it with the following algorithm:

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}$$

The reader can easily verify that the quantity in the summation in the update rule above is just ∂J(θ)/∂θj (for the original definition of J). So, this is simply gradient descent on the original cost function J.(Observe the equations)

> *Point to be noted*

Imagine you're walking down a hill to find the lowest point. The **learning rate (alpha, α)** controls how big each step is:

- **α = 0.001** (Too small): You take tiny steps, so it's **really slow** to reach the bottom.
- **α = 1** (Too large): You take big steps, but you might **overshoot** and miss the lowest point.
- **α = 0.1** (Just right): You take moderate steps, reaching the bottom **efficiently** without missing it.

when you're learning machine learning, you'll often hear about **gradient descent**, a method used to improve models by minimizing errors. There are two main types: **batch gradient descent** and **stochastic gradient descent (SGD)**. Let's break them

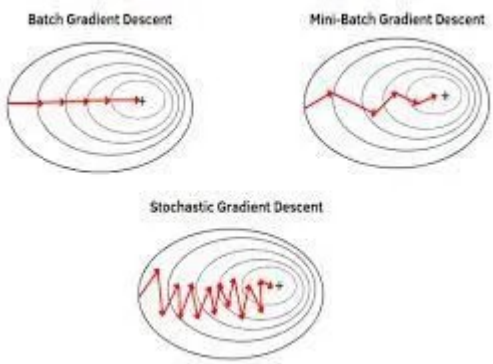
down.

Gradient Descent: The Basics

Gradient descent is about tweaking the model's parameters step by step to reduce the error. The goal is to make these small changes until the error is as low as possible.

Batch Gradient Descent: All at Once

In **batch gradient descent**, the algorithm uses the **entire dataset** to calculate gradients and update the model. It's precise but can be slow, especially with large datasets. It's best for smaller datasets that fit in memory. (refer to the image for better understanding)



Stochastic Gradient Descent (SGD): One Step at a Time

SGD updates the model after looking at just **one data point at a time**. This makes it much faster and better for large datasets, but the updates are noisier and less stable. It refers to the nearest compatible point and goes that way.

Which One to Use?

- Use **batch gradient descent** for small datasets when you need precision.
- Use **SGD** for larger datasets or when you need faster updates.

In short, both methods help you train your model, but the choice depends on the dataset size and the speed you need.

In summary, **Linear Regression** uses the **Least Squares Method (LSM)** to minimize prediction errors. We optimize it with **gradient descent**, adjusting model parameters step by step. You can use **Batch Gradient Descent** for precise but slower updates using the entire dataset, or **Stochastic Gradient Descent (SGD)** for faster, noisier updates with individual data points. The choice depends on your dataset and needs

for speed or accuracy. A well-tuned **learning rate** ensures efficient convergence. These concepts together make linear regression a powerful tool for predictions.

A special shoutout to Andrew Ng for explaining these concepts crystal clear! His teachings have truly simplified complex topics and math.

The math here is a bit trickier than expected... Feel free to reach out to the ***NOTES*** for further clarifications provided in my GitHub account.

Machine Learning

Follow

Written by santhoshini

0 Followers · 11 Following

No responses yet