Machine Learning to build Intelligent Systems

Manas Dasgupta

Understanding Gradient Descent

Structure of this Module

Understanding Gradient Descent

TOPICS
Batch Gradient Descent
Stochastic Gradient Descent
Mini-Batch Gradient Descent
Learning Curve
Learning Rate
Early Stopping
Optimising Logistic Regression Model

Linear Regression

We take a simple example to understand Linear Regression.

We are to **model a relationship between the Car Engine Size and Car Price**. The relationship is assumed to be Linear. We are to find the Linear Equation for the Relationship. We will look at the following aspects:

- Understand the **Closed Form Linear Equation** and find the Linear Equation.
- Use **Scikit Learn Linear Regression** modelling to find out the same and compare with the above.
- Look at how the **optimal Linear Equation is arrived at using Gradient Descent**.

Linear Equation

Linear Equation model:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Notations:

- *ŷ is the predicted value.*
- *n* is the number of features.
- x_i is the i-th feature value.
- θ_j is the j-th model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$).

Cost Function of a Linear Equation model (MSE):

$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

Notations:

- *X* is a matrix containing all the feature values of all instances in the dataset. There is one row per instance and the ith row is equal to the transpose of $x^{(i)}$, noted $(x^{(i)})^T$.
- *$x^{(i)}$ is a vector of all the feature values (excluding the label) of the ith instance in the dataset, and y(i) is its label (the desired output value for that instance).*
- *m* is the number of instances in the dataset you are measuring the RMSE on.

Linear Equation

Linear Equation model:

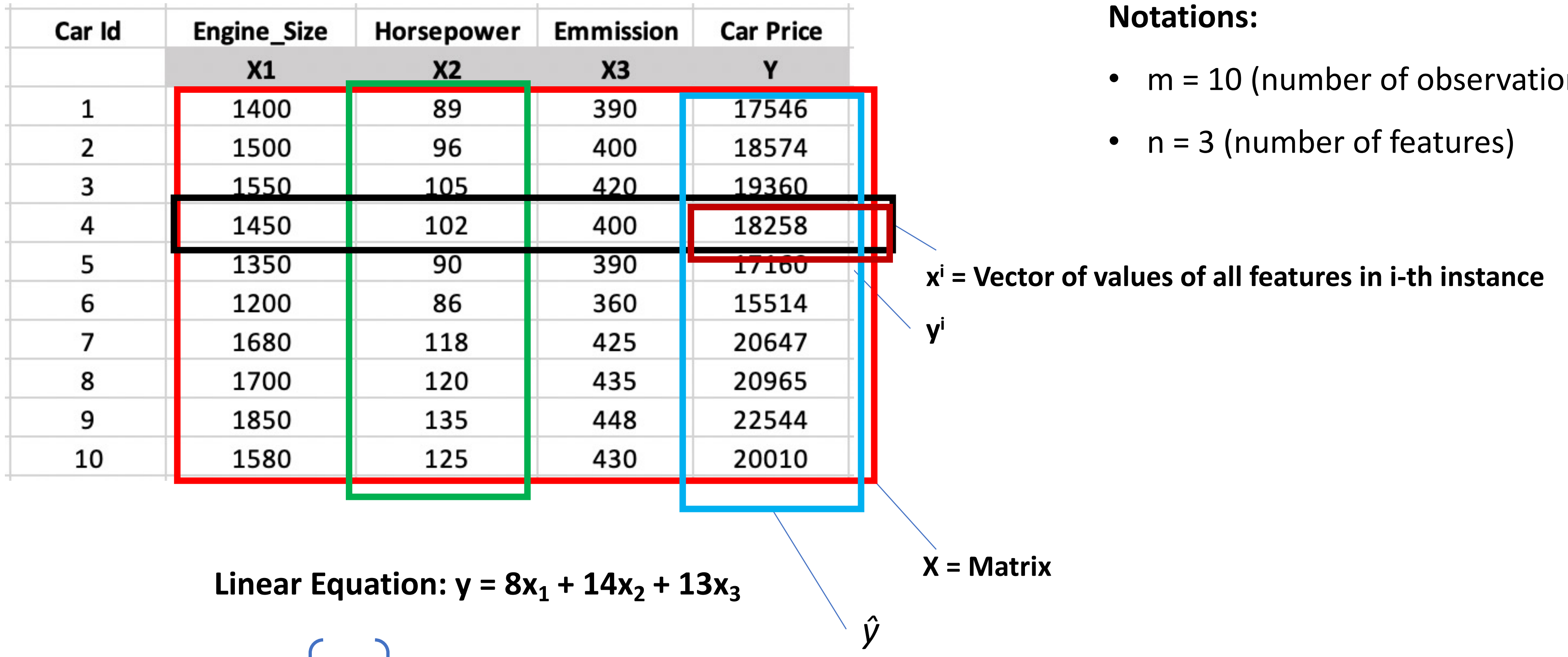
(Vectorized Form)

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

Notations:

- *$\hat{y}$ is the predicted value.*
- θ_j is the j-th model parameter (including the bias term θ_0 and the feature weights θ_1, θ_2, ⋯, θ_n).
- θ is the model's *parameter vector*, containing the bias term $\boldsymbol{\theta_0}$ and the feature weights $\boldsymbol{\theta_1}$ to $\boldsymbol{\theta_n}$.
- **x** is the instance's *feature vector*, containing $\boldsymbol{x_0}$ to $\boldsymbol{x_n}$, with $\boldsymbol{x_0}$ always equal to 1.
- $\boldsymbol{\theta} \cdot \mathbf{x}$ is the dot product of the vectors **θ** and **x**, which is of course equal to $\boldsymbol{\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n}$.
- $h_{\boldsymbol{\theta}}$ is the hypothesis function, using the model parameters **θ**.

Linear Equation Example



Car Id	Engine_Size	Horsepower	Emmission	Car Price
	X1	X2	X3	Y
1	1400	89	390	17546
2	1500	96	400	18574
3	1550	105	420	19360
4	1450	102	400	18258
5	1350	90	390	17160
6	1200	86	360	15514
7	1680	118	425	20647
8	1700	120	435	20965
9	1850	135	448	22544
10	1580	125	430	20010

x^i = Vector of values of all features in i-th instance
y^i
X = Matrix
$\hat{y}$

Notations:

- m = 10 (number of observations/instances)
- n = 3 (number of features)

Linear Equation: $y = 8x_1 + 14x_2 + 13x_3$

$$\theta = \begin{bmatrix} 8 \\ 14 \\ 13 \end{bmatrix}$$

Linear Equation

Python Demo

- **Linear Regression using Closed Form**
- **Linear Regression using Scikit Learn**

Gradient Descent

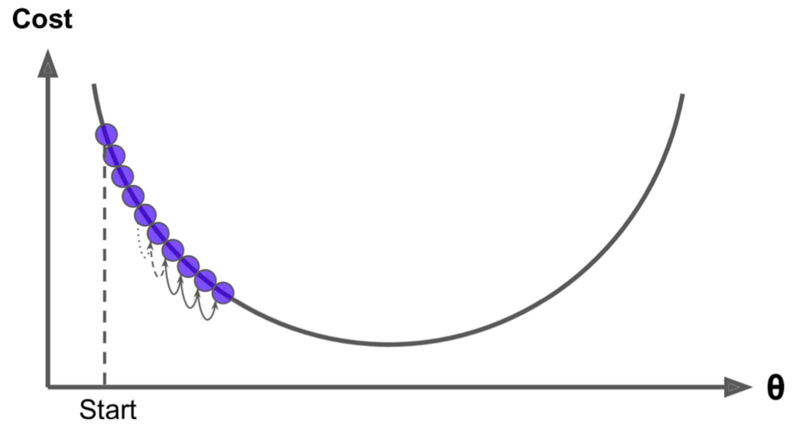
Gradient Descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to **tweak parameters iteratively** in order to **minimize a cost function.**

Suppose you are lost in the mountains in a dense fog; you can only feel the slope of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope. This is exactly what Gradient Descent does: it measures the local gradient of the error function with regards to the parameter vector θ, and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

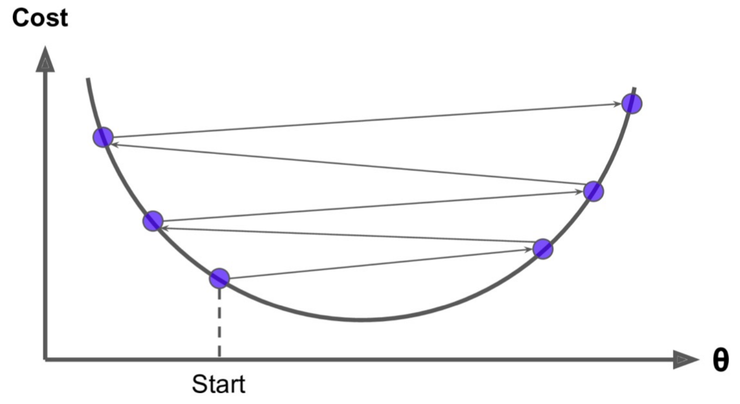
Concretely, you start by filling θ with random values (this is called random initialization), and then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum.

An important parameter in Gradient Descent is the size of the steps, determined by the **learning rate hyperparameter**. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time.

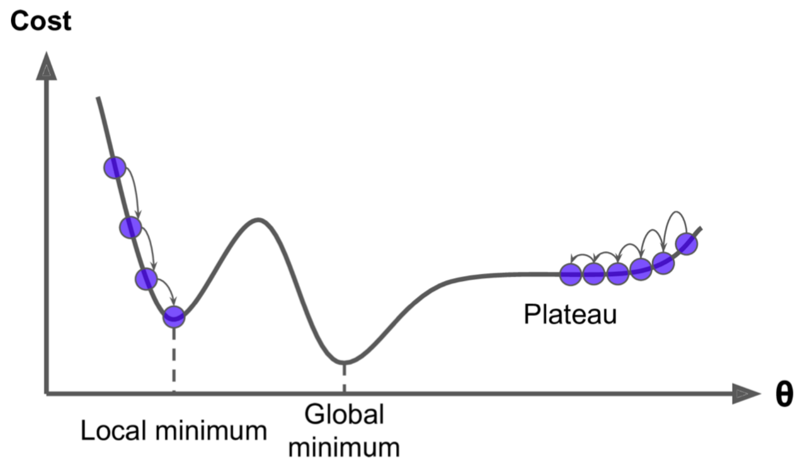
Learning Rate too small



Learning Rate too big



Gradient Descent



Not all cost functions look like nice regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum very difficult.

Figure on the left shows the two main challenges with Gradient Descent:

- If the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*.
- If it starts on the right, then it will take a very long time to cross the plateau, and if you stop too early you will never reach the global minimum.

Fortunately, the MSE cost function for a Linear Regression model happens to be a *convex function*, which means that if you pick any two points on the curve, the line segment joining them never crosses the curve. This implies that **there are no local minima, just one global minimum**. It is also a continuous function with a slope that never changes abruptly. These two facts have a great consequence: Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

When using Gradient Descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's StandardScaler class), or else it will take much longer to converge.

Batch Gradient Descent

Partial Derivative of the Cost Function:

$$\frac{\partial}{\partial \theta_j} \mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

Gradient Vector of the Cost Function:

$$\nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

To implement Gradient Descent, you need to compute the gradient of the cost function with regards to each model parameter θ*j*. I.e., you need to calculate how much the cost function (MSE) will change if you change θ*j* just a little bit. This is called a *partial derivative*.

Our motive is to minimise the Cost Function achieving the global minima.

In Batch Gradient Descent, the partial derivative of all the observations in the Input (Training) Data Set is computed at one go, producing the Gradient Vector.

The gradient vector, contains all the partial derivatives of the cost function (one for each model parameter).

Once you have the gradient vector, which points uphill, we need to go to the opposite direction to go downhill (minimise MSE).

This means **subtracting ∇θMSE(θ) from θ**.

Gradient Descent Step:

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta})$$

Stochastic Gradient Descent

The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large.

Stochastic Gradient Descent just picks a **random instance in the training set** at every step and computes the gradients based only on that single instance. Obviously this makes the algorithm much faster since it has very little data to manipulate at every iteration.

It also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration (SGD can be implemented as an out-of-core algorithm

Due to its stochastic (random) nature, this algorithm is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average.

Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down).

So once the algorithm stops, the final parameter values are good, but not optimal.

When the cost function is very irregular, this can actually help the algorithm jump out of local minima, so Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.

The randomness also means that the algorithm **can never settle at the minimum**. One solution to this dilemma is to gradually reduce the learning rate. The steps start out large, then get smaller and smaller, allowing the algorithm to settle at the global minimum. The function that determines the learning rate at each iteration is called the ***learning schedule***.

If the learning rate is reduced too quickly, you may get stuck in a local minimum, or even end up frozen halfway to the minimum. If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution if you halt training too early.

Mini-Batch Gradient Descent

At each step, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Minibatch GD computes the gradients on **small random sets of instances called *mini-batches***. The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.

The algorithm’s progress in parameter space is less erratic than with SGD, especially with fairly large mini-batches. As a result, Mini-batch GD will end up walking around a bit closer to the minimum than SGD. But, on the other hand, it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike Linear Regression as we saw earlier).

Gradient Descent

Python Demo

- **Batch Gradient Descent**
- **Stochastic Gradient Descent**

Hope you have liked this Video.
Please help us by providing your Ratings and Comments for this Course!

Thank You!!
Manas Dasgupta

Happy Learning!!