IT496: Introduction to Data Mining

Lecture 13

Optimization - II

[Gradient Descent Algorithm]

Arpit Rana

12th September 2023

Components of Supervised Learning

Representation ✔

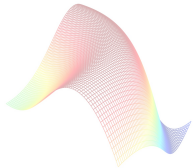
choosing the set of functions (*hypotheses space* or *the model class*) that can be learned.

$$h_\beta(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_m X_m = \sum_{i=1}^{m} \beta_i X_i$$

Evaluation ✔

An evaluation function (also called *objective function* or *scoring function*) is needed to distinguish good hypotheses from bad ones.

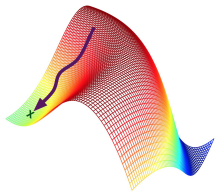
$$J(\beta) = \sum_{i=1}^{m} (h_\beta(X_i) - y_i)^2$$



Optimization

We need a method to search the hypothesis space for the highest-scoring one.

$$\arg min \; J(\beta)$$



Overall Objective

Let $\mathcal{E}$ be the set of all possible input–output examples that follow a prior probability distribution P(X, Y).

Then the expected *generalization loss* for a hypothesis **h** (with respect to loss function L) is–

$$GenLoss_L(h) = \sum_{(x,y) \in \xi} L(y, h(x)) P(x, y).$$

The best hypothesis $\mathbf{h}^*$, is the one with the minimum expected generalization loss:

$$h^* = \arg\min_{\mathsf{h} \in \mathcal{H}} GenLoss_L(h)$$

Overall Objective

Because *P*(X, Y) is not known in most cases, the learner can only estimate generalization loss with *empirical loss* on a set of examples *D* of size N.

$$EmpLoss_{L,D}(h) = \sum_{(x,y) \in D} L(y, h(x)) \frac{1}{N}.$$

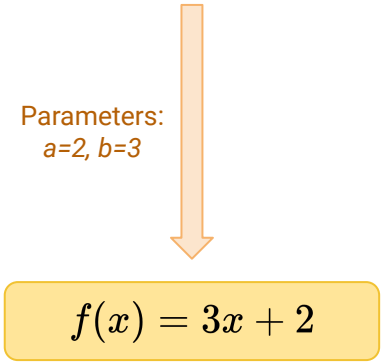
The estimated best hypothesis $\mathbf{h}^*$, is the one with the minimum expected empirical loss:

$$\hat{h} = \underset{\mathsf{h} \in \mathcal{H}}{\arg\min}\ EmpLoss_{L,D}(h)$$

Training

Training finds the best hypothesis within the hypothesis space.

$$y = f(x) = bx + a$$



One common way to find the final hypothesis is by minimizing a loss function using ***Gradient Descent*** algorithm.

Gradient Descent

Gradient Descent is a generic method to tweak parameters iteratively in order to minimize a cost (a.k.a. loss) function.

- It is a search in the model's **parameter space** for values of the parameters that minimize the *loss function*.

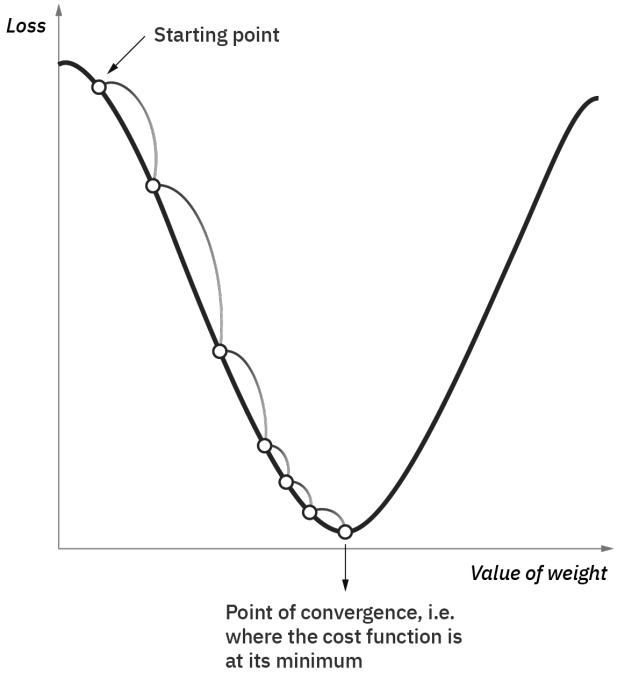
Gradient Descent

Conceptually:

- It starts with an initial guess for the values of the parameters (called *random initialization*).
- Then repeatedly:
  - It updates the parameter values — hopefully to reduce the loss.

Ideally, it keeps doing this until **convergence** — changes to the parameter values do not result in lower loss.

The key to this algorithm is *how to update the parameter values*.



Gradient Descent: The Update Rule

To update the parameter values to reduce the loss:

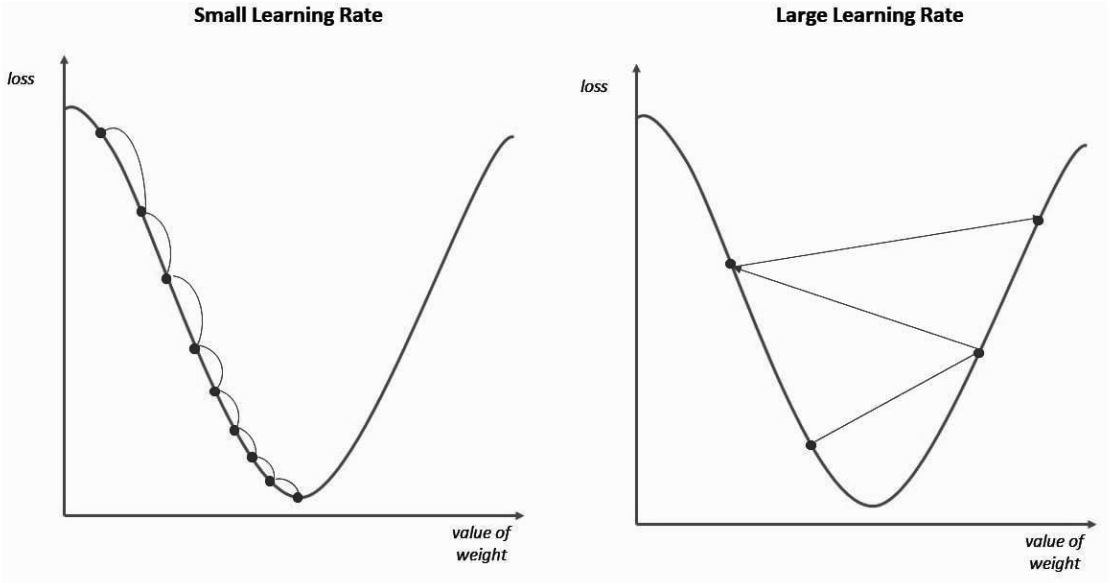
- Compute the *gradient vector*.
    - But this points 'uphill' and we want to go 'downhill'.
    - And we want to make 'baby steps' (see later), so we use a ***learning rate***, α which is between 0 and 1.
- So subtract α times the gradient vector from **w**

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$$

Gradient Descent: The Learning Rate

The size of the steps is determined by the *learning rate*.

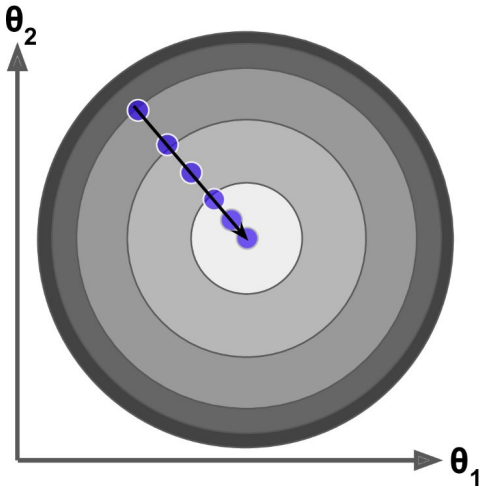
- If the learning rate is too small, it will take many updates until convergence:
- If the learning rate is too big, the algorithm might jump across the valley (overshoot) — it may even end up with higher loss than before, making the next step bigger.



Gradient Descent: Sensitive to Scaling of Features

For Gradient Descent, we do need to scale the features.

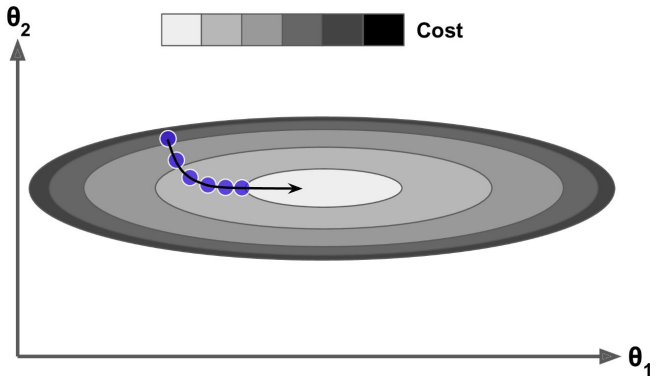
- If features have different ranges, it affects the shape of the 'bowl'.
    - E.g. features 1 and 2 have similar ranges of values — a 'bowl':
- The algorithm goes straight towards the minimum.



Gradient Descent: Sensitive to Scaling of Features

E.g. feature 1 has smaller values than feature 2 — an elongated 'bowl':

- Since feature 1 has smaller values, it takes a larger change in θ_1 to affect the loss function, which is why it is elongated.
- It takes more steps to get to the minimum — steeply down but not really towards the goal, followed by a long march down a nearly flat valley.
- It makes it more difficult to choose a value for the learning rate that avoids *divergence*: a value that suits one feature may not suit another.



Gradient Descent Algorithm Pseudocode

$\mathbf{w} \leftarrow$ any point in the parameter space
while not converged **do**
    for each w_i **in w do**

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$$

Gradient Descent in Action

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$$

For *univariate regression*, the *squared-error loss* is quadratic, so the partial derivative will be linear.

$$\begin{aligned} \frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(x))^2 = 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(x)) \\ &= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i}(y - (w_1 x + w_0)) . \end{aligned}$$

Gradient Descent in Action

Applying this to both w_0 and w_1 we get:

$$\frac{\partial}{\partial w_0} Loss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)); \qquad \frac{\partial}{\partial w_1} Loss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)) \times x.$$

$$w_0 \leftarrow w_0 + \alpha(y - h_{\mathbf{w}}(x)); \quad w_1 \leftarrow w_1 + \alpha(y - h_{\mathbf{w}}(x)) \times x.$$

Here, loss covers only one training example.

Batch Gradient Descent

For N training examples, we want to minimize the sum of the individual losses for each example.

- The derivative of a sum is the sum of the derivatives, so we have:

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)); \quad w_1 \leftarrow w_1 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)) \times x_j .$$

- We have to sum over all N training examples for every step, and there may be many steps.
- A step that covers all the training examples is called an *epoch*.

These updates constitute the *batch gradient descent learning rule for univariate linear regression.*

Gradient Descent: Multivariate Linear Regression

We can easily extend to multivariable linear regression problems, in which each example x_j is an n-element vector

$$h_{\mathbf{w}}(\mathbf{x}_j) = w_0 + w_1 x_{j,1} + \cdots + w_n x_{j,n} = w_0 + \sum_i w_i x_{j,i} .$$

Vectorized form -

$$h_{\mathbf{w}}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^\top \mathbf{x}_j = \sum_i w_i x_{j,i} .$$

The best vector of weights, w^*, minimizes squared-error loss over the examples:

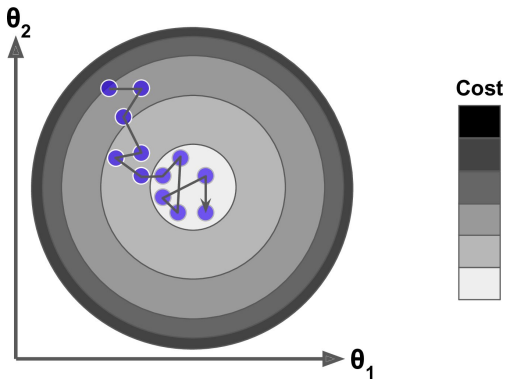
$$\mathbf{w}^* = \operatorname*{argmin}_{\mathbf{w}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j) .$$

$$w_i \leftarrow w_i + \alpha \sum_j (y_j - h_{\mathbf{w}}(\mathbf{x}_j)) \times x_{j,i} .$$

Stochastic Gradient Descent

As we saw, in each iteration, Batch Gradient Descent does a calculation on the entire training set, which, for large training sets, may be slow.

- Stochastic Gradient Descent (SGD), on each iteration, picks *just one training example x_j at random* and computes the gradients on just that one example.
- This gives huge speed-up.
    - It enables us to train on huge training sets since only one example needs to be in memory in each iteration.
    - But, because it is stochastic (the randomness), the loss will not necessarily decrease on each iteration:
- On average, the loss decreases, but in any one iteration, loss may go up or down.
- Eventually, it will get close to the minimum, but not necessarily optimal.



Simulated Annealing

- As we discussed, SGD does not settle at the minimum.
- One solution is to gradually reduce the learning rate:
  - Updates start out 'large' so you make progress.
  - But, over time, updates get smaller, allowing SGD to settle at or near the global minimum.
- The function that determines how to reduce the learning rate is called the learning schedule.
  - Reduce it too quickly and you may not converge on or near to the global minimum.
  - Reduce it too slowly and you may still bounce around a lot and, if stopped after too few iterations, may end up with a suboptimal solution.

Mini-Batch Gradient Descent

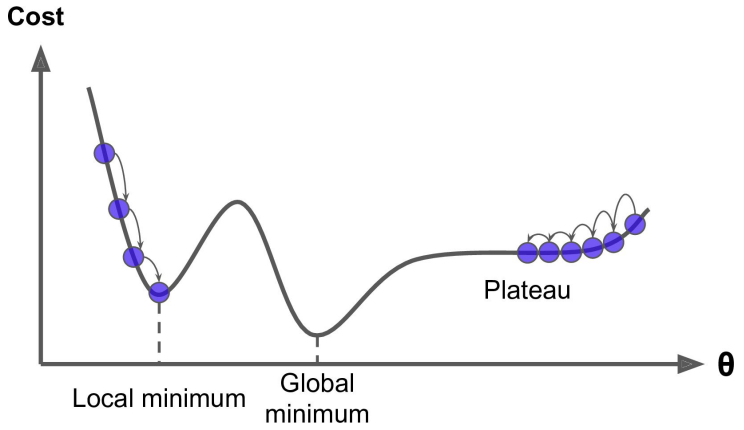
Batch Gradient Descent computes gradients from the full training set and Stochastic Gradient Descent computes gradients from just one example.

- Mini-Batch Gradient Descent lies between the two:
  - It computes gradients from a small randomly-selected subset of the training set, called a mini-batch.
- Since it lies between the two:
  - It may bounce less and get closer to the global minimum than SGD…
    - …although both of them can reach the global minimum with a good learning schedule.
  - Its time and memory costs lie between the two.

Non-Convex Loss Functions

Gradient Descent is a generic method: you can use it to find the minima of other loss functions.

- Not all loss functions are convex, which can cause problems for Gradient Descent:
- The algorithm might converge to a local minimum, instead of the global minimum.
- It may take a long time to cross a plateau.



What do we do about this?

Non-Convex Loss Functions

What do we do about this?

- One thing is to prefer Stochastic Gradient Descent (or Mini-Batch Gradient Descent): because of the way they 'bounce around', they might even escape a local minimum, and might even get to the global minimum.

- In this context, simulated annealing is also useful: updates start out 'large' allowing these algorithms to make progress and even escape local minima; but, over time, updates get smaller, allowing these algorithms to settle at or near the global minimum.

- But, if using simulated annealing, if you reduce the learning rate too quickly, you may still get stuck in a local minimum.

Next lecture

Linear Regression

14th September 2023