Introduction to Machine Learning

Linear Regression

Varun Chandola

February 24, 2020

Outline

Contents

1 Linear Regression **1**
1.1 Problem Formulation 1
1.2 Matrix Calculus Basics 2
1.3 Learning Parameters 2
1.4 Machine Learning as Optimization 3
1.5 Convex Optimization 5
1.6 Gradient Descent 6
1.7 Issues with Gradient Descent 8
1.8 Stochastic Gradient Descent 9

1 Linear Regression

1.1 Problem Formulation

- There is one scalar **target** variable y
- There is one vector **input** variable x
- Inductive bias:

$$y = \mathbf{w}^\top \mathbf{x}$$

Linear Regression Learning Task
Learn $\mathbf{w}$ given training examples, $\langle \mathbf{X}, \mathbf{y} \rangle$.

The training data is denoted as $\langle \mathbf{X}, \mathbf{y} \rangle$, where $\mathbf{X}$ is a $N \times D$ data matrix consisting of N data examples such that each data example is a D dimensional vector. $\mathbf{y}$ is a $N \times 1$ vector consisting of corresponding target values for the examples in $\mathbf{X}$.

- Fitting a straight line to d dimensional data

$$y = \mathbf{w}^\top \mathbf{x}$$

$$y = \mathbf{w}^\top \mathbf{x} = w_1 x_1 + w_2 x_2 + \ldots + w_d x_d$$

- Will pass through origin
- Add intercept

$$y = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_d x_d$$

- Equivalent to adding another column in $\mathbf{X}$ of 1s.

1.2 Matrix Calculus Basics

$$\frac{\partial \mathbf{a}^\top \mathbf{b}}{\partial \mathbf{a}} = \frac{\partial \mathbf{b}^\top \mathbf{a}}{\partial \mathbf{a}} = \mathbf{b}$$

$$\frac{\partial \mathbf{a}^\top \mathbf{M} \mathbf{a}}{\partial \mathbf{a}} = 2\mathbf{M}\mathbf{a}$$

where $\mathbf{M}$ is a symmetric matrix.

1.3 Learning Parameters

- Minimize *squared loss*

$$J(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{N} (y_i - \mathbf{w}^\top \mathbf{x}_i)^2$$

- Make prediction ($\mathbf{w}^\top \mathbf{x}_i$) as close to the target (y_i) as possible

- Least squares estimate

$$\widehat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}$$

The derivation of the least squares estimate can be done by first converting the expression of the squared loss into matrix notation, i.e.,

$$\begin{aligned} J(\mathbf{w}) &= \frac{1}{2}\sum_{i=1}^{N}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2 \\ &= \frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w}) \end{aligned}$$

To minimize the error, we first compute its derivative with respect to $\mathbf{w}$:

$$\frac{\partial LL(\mathbf{w})}{\partial \mathbf{w}} = \frac{1}{2}\frac{\partial}{\partial \mathbf{w}}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w})$$

Note that, we use the fact that $(\mathbf{X}\mathbf{w})^\top \mathbf{y} = \mathbf{y}^\top \mathbf{X}\mathbf{w}$, since both quantities are scalars and the transpose of a scalar is equal to itself. Continuing with the derivative:

$$\frac{\partial LL(\mathbf{w})}{\partial \mathbf{w}} = \frac{1}{2}(2\mathbf{w}^\top \mathbf{X}^\top \mathbf{X} - 2\mathbf{y}^\top \mathbf{X})$$

Setting the derivative to 0, we get:

$$\begin{aligned} 2\mathbf{w}^\top \mathbf{X}^\top \mathbf{X} - 2\mathbf{y}^\top \mathbf{X} &= 0 \\ \mathbf{w}^\top \mathbf{X}^\top \mathbf{X} &= \mathbf{y}^\top \mathbf{X} \\ (\mathbf{X}^\top \mathbf{X})^\top \mathbf{w} &= \mathbf{X}^\top \mathbf{y} \text{ (Taking transpose both sides)} \\ (\mathbf{X}^\top \mathbf{X})\mathbf{w} &= \mathbf{X}^\top \mathbf{y} \\ \mathbf{w} &= (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y} \end{aligned}$$

1.4 Machine Learning as Optimization

At this point, we move away from the situation where a perfect solution exists, and the learning task it to reach the perfect solution. Instead, we focus on finding the *best possible* solution which optimizes certain criterion.

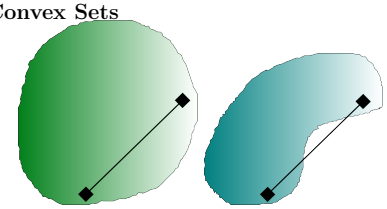
- Learning is optimization

- Faster optimization methods for faster learning
- Let $w \in \mathbb{R}^d$ and $S \subset \mathbb{R}^d$ and $f_0(w), f_1(w), \ldots, f_m(w)$ be real-valued functions.
- Standard optimization formulation is:

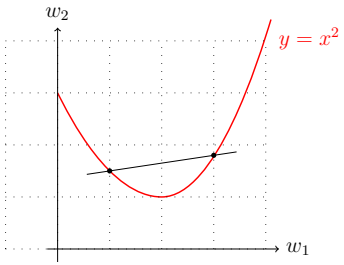
$$\begin{aligned} \underset{w}{\text{minimize}} \quad & f_0(w) \\ \text{subject to} \quad & f_i(w) \leq 0,\ i = 1, \ldots, m. \end{aligned}$$

- Methods for **general optimization problems**
  - Simulated annealing, genetic algorithms
- Exploiting *structure* in the optimization problem
  - **Convexity**, Lipschitz continuity, smoothness

Convex Sets



Convex Functions



[1] Adapted from `http://ttic.uchicago.edu/~gregory/courses/ml2012/lectures/tutorial_optimization.pdf`. Also see, `http://www.stanford.edu/~boyd/cvxbook/` and `http://scipy-lectures.github.io/advanced/mathematical_optimization/`.

Convexity is a property of certain functions which can be exploited by optimization algorithms. The idea of convexity can be understood by first considering *convex sets*. A convex set is a set of points in a coordinate space such that every point on the line segment joining any two points in the set are also within the set. Mathematically, this can be written as:

$$w_1, w_2 \in S \Rightarrow \lambda w_1 + (1-\lambda) w_2 \in S$$

where $\lambda \in [0, 1]$. A *convex function* is defined as follows:

- $f : \mathbb{R}^d \to \mathbb{R}$ is a convex function if the domain of f is a convex set and for all $\lambda \in [0, 1]$:

$$f(\lambda w_1 + (1-\lambda) w_2) \leq \lambda f(w_1) + (1-\lambda) f(w_2)$$

Some examples of convex functions are:

- Affine functions: $w^\top x + b$
- $||w||_p$ for $p \geq 1$
- Logistic loss: $\log(1 + e^{-yw^\top x})$

1.5 Convex Optimization

- Optimality Criterion

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & f_0(w) \\ \text{subject to} \quad & f_i(w) \leq 0,\ i = 1, \ldots, m. \end{aligned}$$

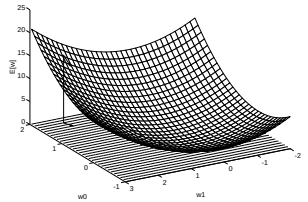
  where all $f_i(w)$ are **convex functions**.

- w_0 is feasible if $w_0 \in Dom\ f_0$ and all constraints are satisfied
- A feasible w^* is optimal if $f_0(w^*) \leq f_0(w)$ for all w satisfying the constraints

1.6 Gradient Descent

- Denotes the direction of steepest ascent

$$\nabla E(\mathbf{w}) = \begin{bmatrix} \frac{\partial E}{\partial w_0} \\ \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_d} \end{bmatrix}$$



A small step in the weight space from $\mathbf{w}$ to $\mathbf{w} + \delta\mathbf{w}$ changes the objective (or error) function. This change is maximum if $\delta\mathbf{w}$ is along the direction of the gradient at $\mathbf{w}$ and is given by:

$$\delta E \simeq \delta \mathbf{w}^\top \nabla E(\mathbf{w})$$

Since $E(\mathbf{w})$ is a smooth continuous function of $\mathbf{w}$, the extreme values of E will occur at the location in the input space ($\mathbf{w}$) where the gradient of the error function vanishes, such that:

$$\nabla E(\mathbf{w}) = 0$$

The vanishing points can be further analyzed to identify them as saddle, minima, or maxima points.

One can also derive the local approximations done by first order and second order methods using the Taylor expansion of $E(\mathbf{w})$ around some point $\mathbf{w}'$ in the weight space.

$$E(\mathbf{w}') \simeq E(\mathbf{w}) + (\mathbf{w}' - \mathbf{w})^\top \nabla + \frac{1}{2}(\mathbf{w}' - \mathbf{w})^\top \mathbf{H}(\mathbf{w}' - \mathbf{w})$$

For first order optimization methods, we ignore the second order derivative (denoted by $\mathbf{H}$ or the *Hessian*). It is easy to see that for $\mathbf{w}$ to be the local minimum, $E(\mathbf{w}) - E(\mathbf{w}') \leq 0$, $\forall \mathbf{w}'$ in the vicinity of $\mathbf{w}$. Since we can choose any arbitrary $\mathbf{w}'$, it means that every component of the gradient ∇ must be zero.

- Set derivative to 0
- Second derivative for minima or maxima

1. Start from any point in variable space
2. Move along the direction of the steepest descent (or ascent)
   - By how much?
   - A learning rate (η)
   - What is the direction of steepest descent?
     - Gradient of E at $\mathbf{w}$

Gradient descent is a first-order optimization method for convex optimization problems. It is analogous to "hill-climbing" where the gradient indicates the direction of steepest ascent in the local sense.

Training Rule for Gradient Descent

$$\mathbf{w} = \mathbf{w} - \eta \nabla E(\mathbf{w})$$

For each weight component:

$$w_j = w_j - \eta \frac{\partial E}{\partial w_j}$$

The key operation in the above update step is the calculation of each partial derivative. This can be computed for perceptron error function as

follows:

$$\begin{aligned}
\frac{\partial E}{\partial w_j} &= \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 \\
&= \frac{1}{2}\sum_i \frac{\partial}{\partial w_j}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2 \\
&= \frac{1}{2}\sum_i 2(y_i - \mathbf{w}^\top \mathbf{x}_i)\frac{\partial}{\partial w_j}(y_i - \mathbf{w}^\top \mathbf{x}_i) \\
&= \sum_i (y_i - \mathbf{w}^\top \mathbf{x}_i)(-x_{ij})
\end{aligned}$$

where x_{ij} denotes the j^{th} attribute value for the i^{th} training example. The final weight update rule becomes:

$$w_j = w_j + \eta \sum_i (y_i - \mathbf{w}^\top \mathbf{x}_i) x_{ij}$$

- Error surface contains only one global minimum
- Algorithm *will* converge
  - Examples need not be linearly separable
- η should be *small enough*
- Impact of too large η?
- Too small η?

If the learning rate is set very large, the algorithm runs the risk of overshooting the target minima. For very small values, the algorithm will converge very slowly. Often, η is set to a moderately high value and reduced after each iteration.

1.7 Issues with Gradient Descent

- Slow convergence
- Stuck in local minima

One should note that the second issue will not arise in the case of Perceptron training as the error surface has only one global minima. But for general setting, including multi-layer perceptrons, this is a typical issue.

More efficient algorithms exist for batch optimization, including *Conjugate Gradient Descent* and other *quasi*-Newton methods. Another approach is to consider training examples in an online or incremental fashion, resulting in an online algorithm called **Stochastic Gradient Descent** [1], which will be discussed next.

1.8 Stochastic Gradient Descent

- **Update weights after every training example.**
- For sufficiently small η, closely approximates Gradient Descent.

Gradient Descent	**Stochastic Gradient Descent**
Weights updated after summing error over all examples	Weights updated after examining each example
More computations per weight update step	Significantly lesser computations
Risk of local minima	Avoids local minima

- Minimize the squared loss using *Gradient Descent*

$$J(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2$$

- Why?

The analytical approach discussed earlier involves a matrix inversion ($(\mathbf{X}^\top\mathbf{X})^{-1}$) which is a $(D+1)\times(D+1)$ matrix. Alternatively, one could solve a system of equations. When D is large, this inversion can be computationally expensive ($O * D^3$) for standard matrix inversion. Moreover, often, the linear system might have singularities and inversion or solving the system of equations might yield numerically unstable results.

To compute the gradient update rule one can differentiate the error with respect to each entry of $\mathbf{w}$.

$$\begin{aligned}\frac{\partial J(\mathbf{w})}{\partial w_j} &= \frac{1}{2}\frac{\partial}{\partial w_j}\sum_{i=1}^{N}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2 \\ &= \sum_{i=1}^{N}(\mathbf{w}^\top \mathbf{x}_i - y_i)x_{ij}\end{aligned}$$

Using the above result, one can perform repeated updates of the weights:

$$w_j := w_j - \eta\frac{\partial J(\mathbf{w})}{\partial w_j}$$

References

References

[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, Dec. 1989.