Convex Optimization

1 Introduction

Machine Learning is function approximation at its core. Can we find a function that fits our data well? In general, we need to select what type of function we think fits our data well. What assumptions did we make for our learning methods so far: Decision Trees, KNN, Naïve Bayes? For KNN, we made the assumption that data points that are "close" to each other in feature space, will have the same label. **Thus, data that is not closesly related will not fit a KNN well.** For decision trees, we assumed that the data could be meaningfully split in a generalizable way. (counter-example: MNIST or image classification from pixels **- classifying individual pixels as digits in relation to an entire image with thousands of pixels will not mean much, and won't generalize well**). For Naïve-Bayes we made the naïve assumption that our features were conditionally independent from each other. So, we always make some model hypothesis about our data. For a linear regression or best fit line, the assumption is that our data is a linear combination of input features. For a polynomial regression, the output is a polynomial of the input variables. Even neural networks and deep learning models **- which are also representations of functions -** are making specific hypotheses about the structure of data.

The methods we've discussed so far have been very particular ML methods, as in they each have a unique way of fitting data and making predictions. The next series of ML methods we will look at are *Parametric Learning Models*. Parametric learning models use a finite set of parameters to make predictions.

The key to generating a good linear regression or polynomial model is continuous optimization. In the past, we looked at discrete optimization (i.e. decision trees or binary classification, which makes discrete choices).

We're now going to take a step back and look at generalizing the process of ML with formal math. This week will review/introduce fundamental techniques: Continuous Optimization, Linear Algebra (data comes in a matrix, labels come in a vector, etc.), and statistics and probability distributions.

2 Continuous Optimization

For some given $f : \mathbb{R}^D \to \mathbb{R}$, we seek an optimal solution x^* and associated solution value $f(x^*)$ such that:

$$f(x^*) = \min_x f(x)$$

In general, we may have additional constraints on the values that x may take on, such as $x \geq 0$, but for most machine learning applications, our parameters will be unconstrained.

Method 1: Critical Points

Example: find the global minimum of:

$$f(x) = x^4 - 4x^3 + 8x + 1$$

You should already be familiar with one method of continuous optimization from Calculus. Namely, finding the critical points of a function. The critical points of a function are the points of a function can be found by finding where the derivative **(or the slope of the tangent line)** is 0 or undefined.

$$f'(x) = 4x^3 - 12x^2 + 8 \tag{1}$$
$$0 = 4x^3 - 12x^2 + 8 \tag{2}$$
$$x = 1 + \sqrt{3}, 1 - \sqrt{3}, 1 \tag{3}$$
$$\tag{4}$$

We can plug the critical points we found into $f(x)$ and find that the function is minimized at $f(1+\sqrt{3}) = -3, f(1-\sqrt{3}) = -3$. **We can also take the second derivative to determine if a critical point is a local maxima (negative) or local minima (positive) - more on this later.**

When the model is simple enough, we can find optimal solutions through critical points. When we can write out a way to find the optimal value explicitly (e.g., solving $0 = 4x^3 - 12x^2 + 8$), we call this a closed-form solution. A closed-form solution requires solving an equation to find an optimal value. However, closed-form solutions are not always possible when our models are particularly complicated (neural networks), **our equations are not differentiable everywhere**, or require too many computational resources to solve (e.g., require inverting a very large matrix). In such a case, we can use the next method, *gradient descent.*

Method 2: Gradient Descent

First, we should briefly cover what a gradient is. For a function $f(\vec{x}) : \mathbb{R}^n \to \mathbb{R}$ where $\vec{x} \in \mathbb{R}^n$, the gradient is the vector

$$\nabla f(\vec{x}) = \begin{bmatrix} \frac{\partial f(\vec{x})}{\partial x_1} \\ \frac{\partial f(\vec{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\vec{x})}{\partial x_n} \end{bmatrix}$$

For the equation $f(\vec{x}) = {x_1}^2 + 2{x_2}^4$, the gradient would be $\nabla f(\vec{x}) = \begin{bmatrix} 2x_1 \\ 8{x_2}^3 \end{bmatrix}$. **We got $2x_1$ by taking the partial derivative of $f(\vec{x})$ with respect to x_1, and $8{x_2}^3$ by taking the partial derivative with repsect to x_2.** For a given point, $x = 3, y = -1$, the gradient vector would be $\begin{bmatrix} 6 \\ -8 \end{bmatrix}$. **So, if we were to plot (3,-1) on a graph and travel in the direction of the gradient vector, the value of our objective function will increase.**

The gradient vector points in the direction of fastest **increase** in value for the function. More often than not, we are seeking to *decrease* our function values. If we begin moving in the opposite direction of the gradient, we will begin to decrease our function value. Remember, derivatives **and gradients** provide local information about the behavior of functions **and only hold for small regions**, so we'll have to take a small step to guarantee that the function value decreases.

Gradient descent is an iterative algorithm, much like local search from earlier in this class. We begin at some initial solution and take small steps in the direction of improvement until we reach a solution where we can no longer improve **(i.e. a local/global minima where our derivative is 0)**.

Note that x is generally used to represent features in the machine learning world. So, to represent the weights of our parametric models we instead use $\vec{w}$. Our goal is to find a good set of weights to optimize our model. If we have n paremeters in our model, $\vec{w}$ will be an

Algorithm 1 Gradient Descent Pseudocode

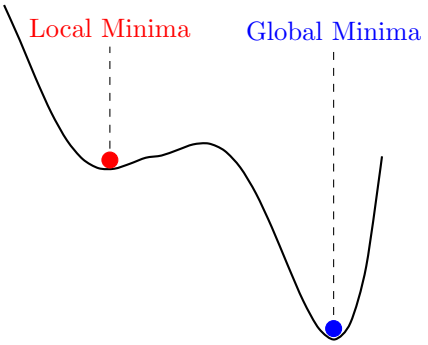
$\vec{w} \leftarrow$ initial point in param space
while convergence not reached **do**
  for w_i in w_f **do**
    $w_i \leftarrow w_i - \alpha * \frac{\partial f}{\partial w_i}$
  end for
end while

n-length vector with initial weights - these can all start out at 0. The general algorithm is as follows:

The alpha represents the **learning rate** - these values are generally very low - like 0.1. This changes on a case-by-case basis.

Example: $f(w_1) = {w_1}^2$
$\alpha = 0.1$
$w_1 = 2$

We'd like our solutions to act like a ball rolling down a hill. If the function slopes downward, our solution moves in the direction of steepest descent. However, this approach doesn't always work - our ball could find a **local** minima and not the global minima.



3 Convex Optimization

When are we guaranteed to find an optimal solution? Intuitively, if we can roll a ball down a function and it always funnels to a single point (like a quadratic), then we know we can always find the optimal solution. Such functions are called *convex*. Formally, for all $0 \leq \lambda \leq 1$ and all possible x_1, x_2, a convex function is a function that satisfies:

$$f(\lambda x_1 + (1 - \lambda)x_2) \leq \lambda f(x_1) + (1 - \lambda)f(x_2)$$

The left hand side corresponds to all values the function can take on in between x_1 and x_2. The right hand side corresponds to the line between $f(x_1)$ and $f(x_2)$. Together, it says that the value the function takes on in between any x_1 and x_2 must be less than or equal to the value of the line connecting x_1 and x_2

4 Non-Convex Optimization

Not all functions we want to optimize are convex, some in fact are highly-non-convex with many possible local minima, saddle points, and many other things that can get in the way of GD's convergence. How can we deal with these issues? In a very similar way to how we solved the issue in local search, with the addition of "random actions".

One possible solution: Add some (gaussian) noise to the gradient vector before taking a step of gradient descent.

A better alternative: Don't compute the full gradient vector, just compute a small portion. This method is called Stochastic Gradient Descent and is the reason why Deep Learning is so successful.