Optimization Theory

Nishanth Nair

10 Nov,2019

Numerical Computation

Overflow and underflow

- Problem with performing continuous math on digital computers is the need to represent infinitely many real numbers with a finite number of bit patterns
- This means, we incur some approximation error when we represent the number in a computer. In many cases, this is **rounding error**.
- Rounding error is problematic when it compounds across many operations and can cause algorithms that work in theory to fail in practice.
- **Underflow**: Occurs when numbers near zero are rounded to zero.
- **Overflow**: Occurs when numbers with large magnitude are approximated to ∞ or $-\infty$
- Stabilizing against rounding errors: Example softmax function

$$softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

  - if all x_i are equal to a constant say 'c', theoretically, all outputs should equal $\frac{1}{n}$
  - if 'c' is negative with a very large magnitude, exp(c) will underflow making the denominator zero, and thus having the softmax as undefined.
  - One trick to solve this problem: evaluate softmax(z_i), where $z_i = x_i - max(x_j)$.
  - Softmax is unchanged by adding or subtracting a scalar to the input vector.
  - Subtracting $max(x_j)$ results in largest argument to exp being zero which rules out overflow.
  - Likewise, atleast 1 term in the denominator will have a value of 1 ruling out underflow in the denominator.
  - Underflow in the numerator can still cause expression to evaluate to zero.
  - Such issues need to be considered while implementing functions.
  - Many libraries provide stable implementations keeping these numerical issues in mind.

Conditioning

- Refers to how rapidly a function changes with respect to small changes in its input.
- Functions that change rapidly when their inputs are perturbed slightly can be problematic because rounding errors in the inputs result in large changes in the output.

- Consider a matrix $A \in R^{nxm}$ has an eigen decomposition. The **condition number** of A is

$$\max_{i,j} |\frac{\lambda_i}{\lambda_j}|$$

- Condition number is the ratio of the magnitude of the largest and smallest eigen values.
- When condition number is large, matrix inversion (eg. $f(x) = A^{-1}.x$) is sensitive to error in input.

Optimization

- Optimization refers to the task of minimizing or maximizing some function f(x) by altering x.

- The function we need to minimize or maximize is called the **objective function or criterion**
- When we are minimizing it, we call it the **cost function, loss function or error function** and when we maximize, we call it **utility or fitness function**.
- Maximization can be achieved via a minimization algorithm by minimizing -f(x).
- Constrained optimization:find min or max subject to constraints vs unconstrained optimization

Gradient Based Optimization

First Derivative:

- The derivative of a function, denoted by $f'(x) or \frac{dy}{dx}$, gives the slope of f(x) at x.
- The derivative specifies how to scale a small change in the input to get the corresponding change in output.
- $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$
- Points where $f'(x) = 0$ are called **critical or stationary points** and slope is zero at these points.
  - These could be local minimum or local maximum or in some cases neither where they are called **saddle points.**
- The **gradient** generalizes the notion of the derivative to the case where the derivative is with respect to a vector:
  - The gradient of f is the vector containing all the partial derivatives, denoted by $\nabla_x f(x)$
  - The partial derivative $\frac{\partial f}{\partial x}$, measures how f changes with respect to x.
  - in multiple dimensions, critical points are points where every element of the gradient equals zero.
- Sometimes we need to find the gradient of a function whose input and output are both vectors.The matrix containing all such partial derivatives function is called the **Jacobian Matrix**:

- If we have a function $f : IR^m \mapsto IR^n$, then the Jacobian Matrix, $J \in IR^{nxm}$ of f is defined by

$$J_{i,j} = \frac{\partial f(x)_i}{\partial x_j}$$

- In the special case where n=1, the Jacobian, J is a vector which is the same as the gradient, ∇_x.

Subgradients

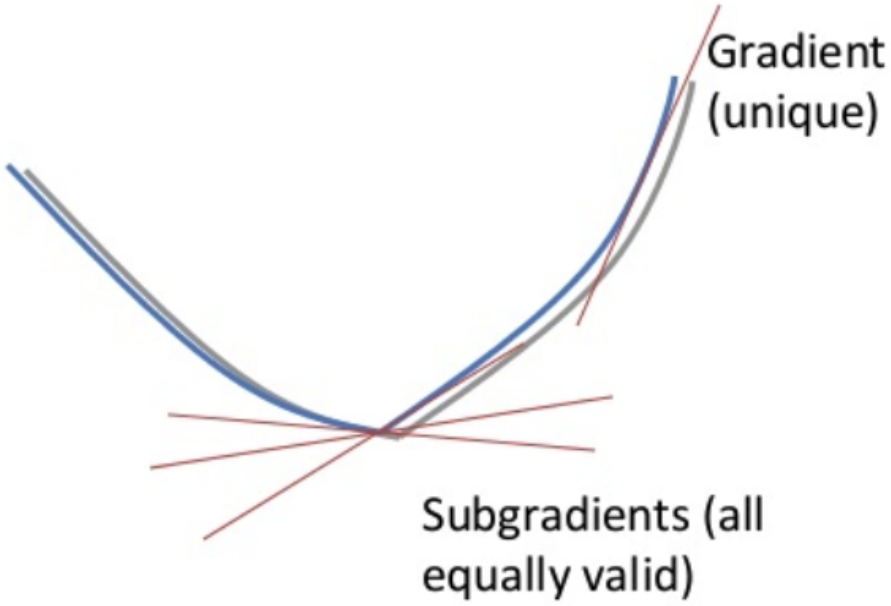

Figure 1: Example of subgradients

- Some functions are not differentiable everywhere.
- For example, the hinge function, $f(x) = \max(0, 1 - x)$ is differentiable at x > 1 and x < 1, but not at x=1
- Subgradients or subderivatives, denoted as ∂f, are a generalization of derivatives to non differentiable functions.
- The derivative of f at x is a the tangent line.
  - It is the line that touches f at x that is always below f(for convex functions).
- The subderivative, is the set of all such lines.
  - At differentiable positions, this set consists just of the actual derivative.
  - At non-differentiable positions, this contains all slopes that define lines that always lie under the function and make contact at the operating point
- Computing subgradient of hinge loss:

$$\text{Hinge Loss}, L_{hinge} = max(0, 1 - y_n.(w.x_n + b))$$

$$\implies \text{subgradient}, \partial_w L_{hinge} = \partial_w max(0, 1 - y_n.(w.x_n + b))$$

$$= \partial_w \begin{cases} 0, & \text{if } y_n.(w.x_n + b) > 1 \\ y_n.(w.x_n + b), & \text{otherwise} \end{cases}$$

$$= \begin{cases} \partial_w 0, & \text{if } y_n.(w.x_n + b) > 1 \\ \partial_w y_n.(w.x_n + b), & \text{otherwise} \end{cases}$$

$$= \begin{cases} 0, & \text{if } y_n.(w.x_n + b) > 1 \\ y_n.x_n, & \text{otherwise} \end{cases}$$

Second Derivative

- The derivative of a derivative is the second derivative. Denoted by $\frac{d^2 f}{dx^2}$ or $f''(x)$.
- This specifies how the gradient will change as we vary the input.
- Measures **curvature**. Functions with high curvature have gradients that change quickly.
- **Second order conditions**: At the point x^*, if $f'(x = x^*) = 0$
  - and $f''(x = x^*) > 0$, point is minimum.
  - and $f''(x = x^*) < 0$, point is maximum.
  - and $f''(x = x^*) = 0$, point is a saddle, no curvature.
- When the function has multiple input dimensions, the second derivatives can be collected into a matrix called the **Hessian Matrix**.
  - The Hessian matrix, $H(f(x))$ is defines as :

$$H_{i,j} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}$$

  - Hessian is the Jacobian of the gradient. i.e

$$H(f(x)) = J(\nabla f(x))^T$$

Convex functions

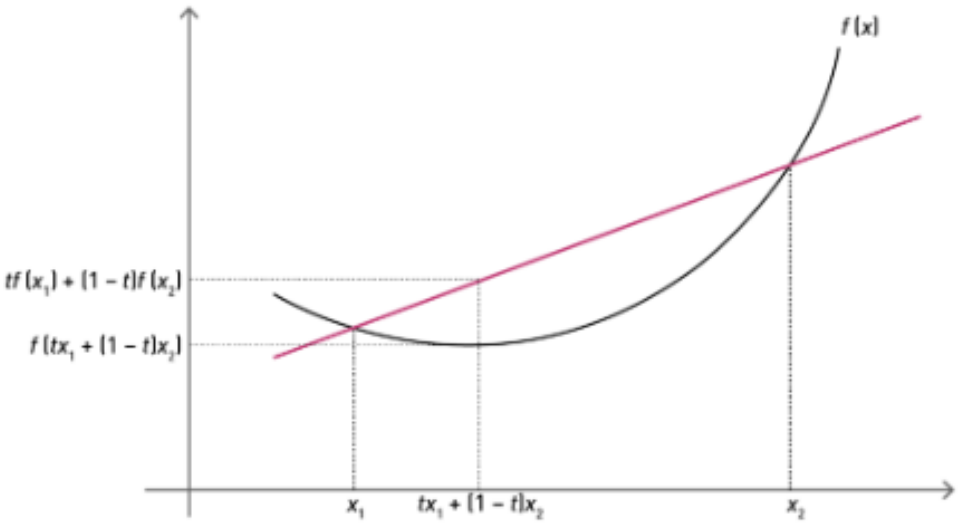

Figure 2: Example of convex function

Zero order conditions

Let X be a convex set in the real vector space and let $f : X \mapsto R$ be a function.

- f is **convex** if

$$\forall x_1, x_2 \in X, \forall t \in [0,1] : f(t.x_1 + (1-t).x_2) \leq t.f(x_1) + (1-t).f(x_2)$$

- f is **strictly convex** if

$$\forall x_1, x_2 \in X, \forall t \in (0,1) : f(t.x_1 + (1-t).x_2) < t.f(x_1) + (1-t).f(x_2)$$

- f is (strictly) concave if -f is (strictly) convex.

First Order Condition

- A real valued differentiable function is convex iff:

$$f(y) \geq f(x) + \nabla f(y)^T.(y-x), \forall x, y \in R$$

- The function is globally above the tangent at y.

Convex and non-convex optimization

- In Euclidean space, an object is convex if for every pair of points within the object, every point on the straight line segment that joins them is also within the object.
  - Example, a solid cube is convex, but anything that is hollow or has a dent in it, for example, a crescent shape, is not convex.
- **Definition**: An optimization problem is convex if its objective function is convex, the inequality constraints are convex and the equality contraints are affine.
- Benefits of convex optimization:
  - **Theorem**: If x is a local minimizer in a convex optimization problem,it is a global minimizer.
  - **Theorem**: $\nabla f(x) = 0$ if and only if x is a global minimizer of $f(x)$.
  - No need to look at second order conditions to determine max or min.
- Examples of covex optimization in ML
  - Linear regression/ Ridge regression, with Tikhonov regularisation
  - Sparse linear regression with L1 regularisation, such as lasso
  - support vector machines
  - parameter estimation in linear-Gaussian time series (Kalman filter)
- Examples of non-convex optimization in ML:
  - Neural Networks
  - Maximum likelihood mixtures of Gaussians

General approach to find Extremum of a function

- Well-behaved version spaces: Convex or concave function.
  - Algorithms seek a local extrema knowing that it will be global.
  - If function is a concave function, then local maximum is a global maximum.
  - If is a convex function, then local minimum is a global minimum.
  - Optimization algorithms to use: Newton-Raphson, gradient descent, conjugate gradient descent.
- Otherwise
  - Optimization approaches: Hill climibing, Simulated annealing

Convex Optimization Algorithms

To find the optimum, we find the roots of the gradient. Following are some algorithms used to do this:

- Closed form
- Newton-Raphson
- Gradient Descent
- Bisection method

Newton Raphson Method

Derivation: Solve a function of the form $f(x) = 0$

$$f(x) \approx f(x_0) + f'(x_0).(x - x_0)$$
$$\implies f(x_{i+1}) \approx f(x_i) + f'(x_i).(x_{i+1} - x_i)$$

To find the root, $f(x_{i+1}) = 0$, thus

$$f(x_i) + f'(x_i).(x_{i+1} - x_i) = 0$$
$$\implies f'(x_i).(x_{i+1} - x_i) = -f(x_i)$$
$$\implies x_{i+1} - x_i = -[f'(x_i)]^{-1}.f(x_i)$$
$$\implies x_{i+1} = x_i - [f'(x_i)]^{-1}.f(x_i)$$

Goal: Find the roots of the gradient function $f'(x)$

STEPS:

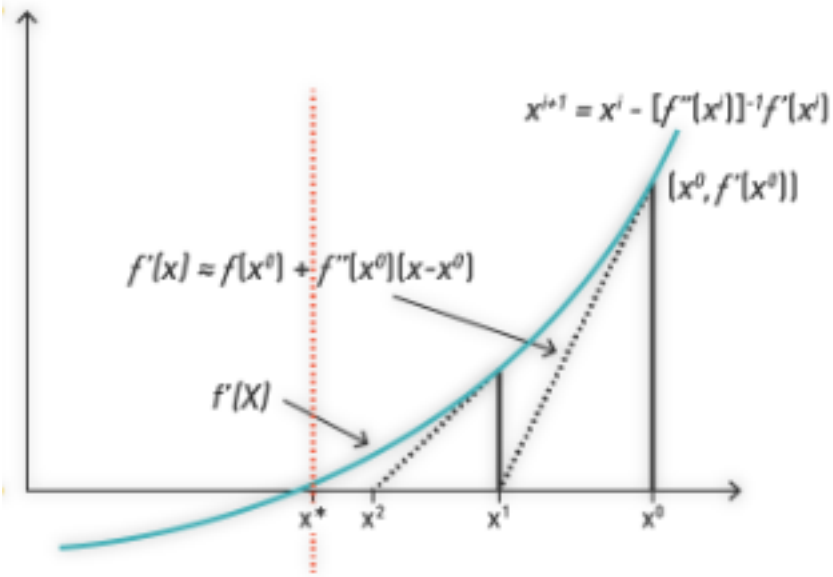

Figure 3: Newtons Method

- initialize x_0 to a random guess. For i=0, $x_{i+1} = x_0$
- Find the tangent at $(x_{i+1}, f'(x_{i+1}))$
- Find the point, x^* where $f_{tangent_{x_{i+1}}} = 0$. Based on the above, this is given by

$$x_{i+1} = x_i - [f''(x_i)]^{-1}.f'(x_i), \text{(Univariate case)}$$

$$x_{i+1} = x_i - [H(x_i)]^{-1}.J(x_i), \text{(Multivariate case, H- Hessian, J- Jacobian)}$$

- Repeat until x^* does not change.
- NOTE: Calculating the Hessian, H in multivariate case, and inverting it is complex so simpler algorithms have been developed such as gradient descent.

Gradient descent

Similar to Newton-Raphson, except, instead of taking inverse of Hessian, we use a hyper-parameter α called the learning rate.

Goal: Find the roots of the gradient function $f'(x)$

STEPS:

- initialize x_0 to a random guess and learning rate α to a small value. For i=0, $x_{i+1} = x_0$
- Find the point, x^* where $f_{tangent_{x_{i+1}}} = 0$. This is given by:

$$x_{i+1} = x_i - \alpha.f'(x_i), \text{(Univariate case)}$$

$$x_{i+1} = x_i - \alpha.J(x_i), \text{(Multivariate case, J- Jacobian)}$$

- Repeat until x^* does not change.

- Learning rate can be determined as follows:
  - Fixed
  - Decay over time
  - Line search

Linear Regression using MSE Example

MODEL: For input $X \in R^n$, predict a scalar $y \in R$ as output.

$$y = h(x) = \sum_{i=0}^{n} \theta_i.x_i = \theta^T.x, \theta \in R, n : \text{number of input variables}$$

Performance Measure: Mean Square Error, MSE $= \frac{1}{2}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$

GOAL: Minimize MSE. Thus, Cost Function, $J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$

Gradient descent algorithm:

We need to choose θ to minimize $J(\theta)$. To do so, start with an initial θ and repeatedly perform the update

$$\theta_j = \theta_j - \alpha.\frac{\partial J(\theta)}{\partial} \quad \text{(where j is the number of input variables or features)}$$

To implement the algorithm, we first need to find the partial derivative (or Jacobian) of the cost function:

$$\nabla_\theta J(\theta) = \frac{\partial J(\theta)}{\partial \theta_j}$$

$$= \frac{\partial}{\partial \theta_j}.(\frac{1}{2}(\hat{y}_j - y)^2)$$

$$= \frac{1}{2}.2.(\hat{y}_j - y).\frac{\partial}{\partial \theta_j}(\hat{y}_j - y)$$

$$= (\hat{y}_j - y).\frac{\partial}{\partial \theta_j}(\sum_{i=0}^{n} \theta_i.x_i - y)$$

$$= (\hat{y}_j - y).x_j$$

thus, for a single training example, the update rule is:

$$\boxed{\theta_j = \theta_j + \alpha.(y^i - \hat{y}_j^i).x_j^i}$$

, where j is the number of input variables or features and i is the ith example

- The rule is called the **LMS update rule** (LMS stands for "least mean squares"), and is also known as the **Widrow-Hoff learning rule**.
- **Batch Gradient Descent**: Looks at every example in the training set on every step. The update rule is given by:

Repeat until Convergence {

$$\theta_j = \theta_j + \alpha.\sum_{i=1}^{m}(y^i - \hat{y}_j^i).x_j^i, \text{ (For every j)}$$

}

- **Stochastic Gradient Descent (or Incremental Gradient descent)**: In this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that single training example only.The update rule is given by:

Repeat until Convergence {

for i = 1 to m: {

$\theta_j = \theta_j + \alpha.(y^i - \hat{y}^i_j).x^i_j$, (For every j)

}

}

- Stochastic gradient descent can start making progress right away and continues to make progress with each example it looks at.
- Often, stochastic gradient descent gets θ "close" to the minimum much faster than batch gradient descent.
  - Note, however, that it may never "converge" to the minimum, and the parameters θ will keep oscillating around the minimum of J(θ) - but in practice most of the values near the minimum will be reasonably good approximations to the true minimum.
- For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

Logistic Regression using gradient descent

- Linear regression has an analytic solution.So gradient descent is not often used.
- However, logistic regression does not have the same analytic solution and gradient descent is commonly used in this case.
- Models linear relationship between one or more independent variables.
- Dependent variable is binary (0 or 1) instead of continuous arbitrary value (as in linear regression).

Note on Sigmoid function

- Logistic (sigmoid) function is given by : $g(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}$
- In Logistic regression, $z = \alpha + \beta.X + ..$
- It transforms $[-\infty, +\infty] \rightarrow [0, 1]$
- Constrains output of our model between 0 and 1

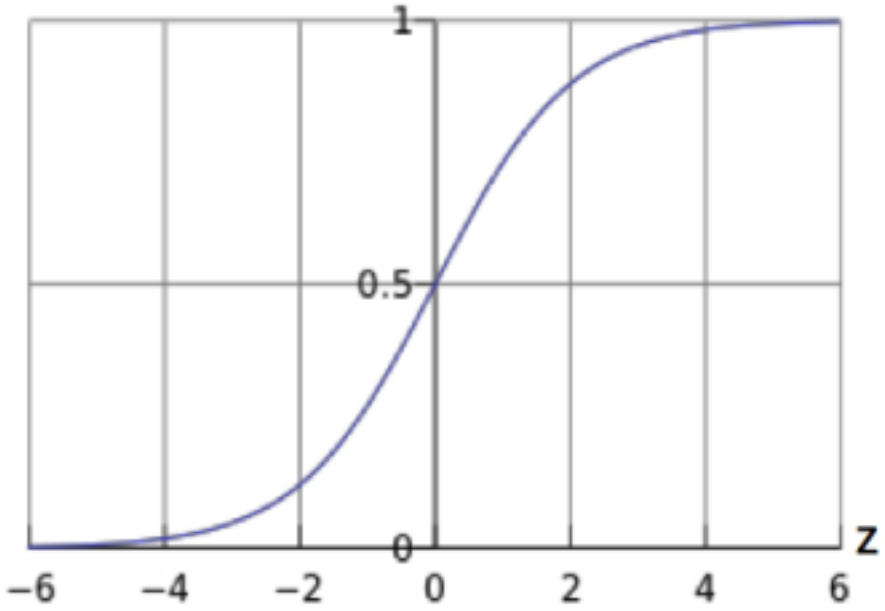

Figure 4: Sigmoid Function

- Squared Error loss of a sigmoid function:

$$J(\alpha, \beta) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - \frac{1}{1 + e^{-(\alpha+\beta.x)}})$$

- **This function is not convex**. Thus cannot be used as the cost function for Logistic Regression.

Logistic regression

MODEL: For input $X \in R^n$, predict a binary output y.

$$P(Y_i = 1|x : \theta) = h(x) = \frac{1}{1 + e^{-(\theta^T.x)}}, \theta \in R, n : \text{number of input variables}$$

Performance Measure or goal: Minimize Logistic Loss,

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N}[y_i.log(\hat{y}_i) + (1 - y_i).log(1 - \hat{y}_i)]$$

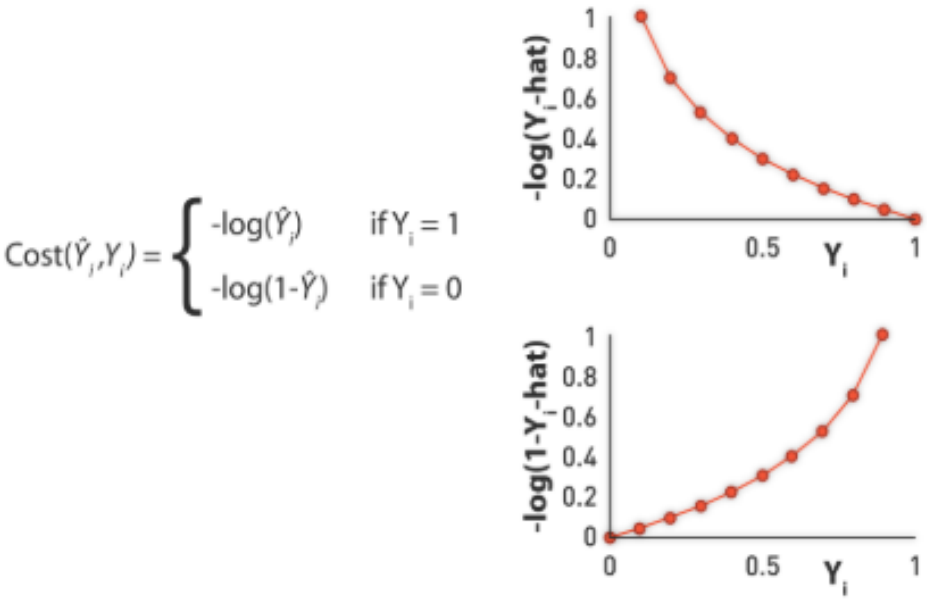

Figure 5: Cost Function

Gradient descent algorithm:

We need to choose θ to minimize $J(\theta)$. To do so, start with an initial θ and repeatedly perform the update

$$\theta_j = \theta_j - \alpha.\frac{\partial J(\theta)}{\partial \theta} \quad \text{(where j is the number of input variables or features)}$$

To implement the algorithm, we first need to find the partial derivative (or Jacobian) of the cost function:

$$\nabla_\theta J(\theta) = \frac{\partial J(\theta)}{\partial \theta_i} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i).x_i$$

thus, for a single step, the update rule is:

$$\begin{aligned}\theta_j &= \theta_j + \alpha.\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i).x_i \\ &= \theta_j + \alpha.\frac{1}{N}\sum_{i=1}^{N}(y_i - \frac{1}{1+e^{-\theta^T.x_i}}).x_i\end{aligned}$$

Feature Scaling

With gradient descent, the direction of steepest descent can depend on the units in which variables are measured

- Inefficient when features or axes are on different scales
- More efficient when different features are on the same scale
- Force features to be roughly between -1 and 1
- Options for feature scaling:
  - $x_i = \frac{x_i}{max(x)}$
  - Mean Normalization: $x_i = \frac{x_i = \bar{x}}{s}$, s is the standard deviation or range of x.

Closed Form (Linear regression)

MODEL:

For input $X \in R^n$, predict a scalar $y \in R$ as output.

$$y = \theta^T.x, \theta \in R$$

Performance Measure:

Mean Square Error, MSE = L^2 Norm = $\sum_i (\hat{y}_i - y_i)^2$

- NOTE: L^P Norm of x,$||x||_p = (\sum_i |x_i|^p)^{\frac{1}{p}}$

GOAL: Minimize MSE

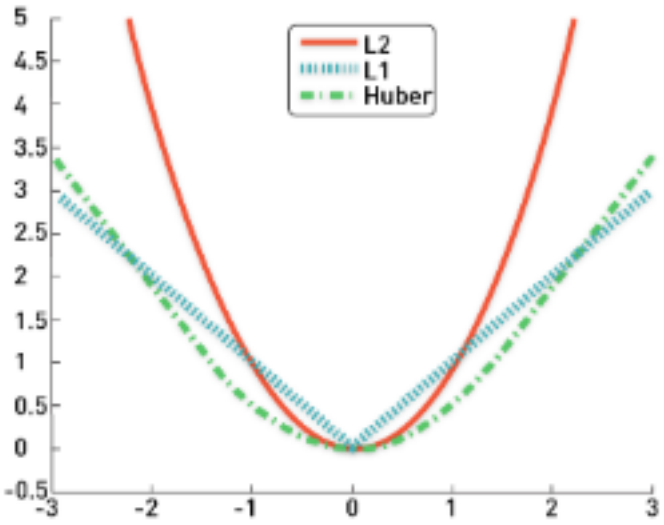

Figure 6: convex functions

- L^2 Norm is a convex function.
- When we combine all training data, sum of all convex functions is convex.
- Based on FOC for convex optimization, $\nabla f(x) = 0$ at the global minimum.

Thus, $\nabla_\theta MSE^{train} = 0$

Closed form Derivation:

$\nabla_\theta MSE^{train} = 0$

$\implies \nabla_\theta \sum_i (x_{train}.\theta - y_{train})^2 = 0$

$\implies \nabla_\theta (x_{train}.\theta - y_{train})^T.(x_{train}.\theta - y_{train}) = 0$, since , $\sum_i x_i^2 = x^T.x$

$\implies \nabla_\theta (\theta^T.x_{train}^T - y_{train}^T).(x_{train}.\theta - y_{train}) = 0$

$\implies \nabla_\theta (\theta^T.x_{train}^T.x_{train}.\theta - y_{train}^T.x_{train}.\theta - y_{train}.\theta^T.x_{train}^T + y_{train}^T.y_{train}) = 0$

$\implies \nabla_\theta (\theta^T.x_{train}^T.x_{train}.\theta - \theta^T.x_{train}^T.y_{train} - y_{train}.\theta^T.x_{train}^T + y_{train}^T.y_{train}) = 0$, since, $x^T.y = y^T.x$

$\implies \nabla_\theta (\theta^T.x_{train}^T.x_{train}.\theta - 2.\theta^T.x_{train}^T.y_{train} + y_{train}^T.y_{train}) = 0$

$\implies 2.x_{train}^T.x_{train}.\theta - 2.x_{train}^T.y_{train} + 0 = 0$, since, $\nabla_\theta \theta^T.\theta = 2.\theta$ and $\nabla_\theta \theta^T = 1$

$\implies \boxed{\theta = (x_{train}^T.x_{train})^{-1}(x_{train}^T.y_{train})}$, Closed Form Equation

Gradient Descent vs Closed Form

Closed Form:

```
- gives an exact solution, modulo numerical innacuracies with computing matrix inverses.
- Assuming N examples with F features each,
    - Constructing $X^T.X$ takes O($NF^2$)
    - inversion takes O($F^3$)
    - Multipliccation takes O(NK)
    - Overall runtime is: O($F^3 + F^2.N$)
        - If N > D, runtime $\approx O(F^2.N)$
- For low to medium dimensional  problems (F < 100), closed form is probably faster.
```

Gradient Descent:

```
- will give you progressively better solutions and will eventually converge to the optim
- Assuming N examples with F features each,
    - To run gradient descent for one step will take O(NF) time,with a relatively small
    - To run K iterations, yields an overall runtime of O(KNF).
```

- For high dimensional problems (F > 10,000), gradient descent is probably faster.