Linear and Logistic Regression

Linear Regression

To perform supervised learning, we must decide how we're going to represent functions/hypotheses h in a computer. As for Linear regression, we decide to approximate y as a linear function of x:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

where θ_i's are the **parameters** (also called **weights**) parameterizing the space of linear functions mapping from X to Y. To simplify our notation, we also introduce the convention of letting $x_0 = 1$ (this is the **intercept term**), so that

$$h(x) = \sum_{i=0}^{d} \theta_i x_1 = \theta^T x,$$

on RHS we are viewing θ and x both as vectors (column matrices), and here d is the number of input variables (not counting x_0). To learn, the parameters θ, we will define a function that measures, for each value of the θ's, how close the $h(x^{(i)})$'s are to the corresponding $y^{(i)}$'s and then we try to minimize it by minimizing the **cost function** defined as:

$$J(\theta) = \frac{1}{2} \sum_{i=0}^{n} (h_\theta(x^{(i)}) - y^{(i)})^2.$$

LMS Algorithms

We want to choose θ so as to minimize $J(\theta)$, one natural and intuitive method to do so is, start with some "*initial guess*" for θ, and that repeatedly changes θ to make $J(\theta)$ smaller, until hopefully we converge to a value of θ that minimizes $J(\theta)$. Specifically, let's consider the **gradient descent algorithm**, which starts with some initial θ, and repeatedly performs the update:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_j).$$

(This update is simultaneously performed for all values of j = 0, . . . , d.) Here, α is called the **learning rate**. This is a very natural algorithm that repeatedly takes a step in the direction of **steepest decrease** of J.

Formal definition

Gradient Descent is an iterative optimization algorithm used to minimize a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$. Given an initial point $x_0 \in \mathbb{R}^n$, gradient descent generates a sequence of points $\{x_k\}_{k=0}^{\infty}$ by the recursive formula:

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

Batch Gradient Descent

In order to implement this algorithm, we have to work out what is the partial derivative term on the right hand side. Let's first work it out for the case of if we have only one training example (x, y). In this case

$$\begin{aligned}\frac{\partial}{\partial\theta_j}J(\theta) &= \frac{\partial}{\partial\theta_j}\frac{1}{2}(h_\theta(x)-y)^2 \\ &= 2\cdot\frac{1}{2}(h_\theta(x)-y)\cdot\frac{\partial}{\partial\theta_j}(h_\theta(x)-y) \\ &= (h_\theta(x)-y)\cdot\frac{\partial}{\partial\theta_j}\left(\sum_{i=0}^{d}\theta_i x_i - y\right) \\ &= (h_\theta(x)-y)x_j\end{aligned}$$

Therefore, the update rule is

$$\theta_j := \theta_j + \alpha(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}.$$

This rule is called the **LMS** update rule (LMS stands for "**least mean squares**"), and is also known as the **Widrow-Hoff learning** rule. Here the the magnitude of the update is proportional to the error term. Now, for a training set of more than one datapoint:
Repeat until it converge {

$$\theta_j = \theta_j + \alpha\sum_{i=1}^{n}(y - (h_\theta(x^{(i)}))x_j^{(i)},\ \ \forall\ j$$

}
By grouping the updates of the coordinates into an update of the vector θ, we can rewrite update as:

$$\theta := \theta + \alpha\sum_{i=1}^{n}(y - (h_\theta(x^{(i)}))x^{(i)}$$

This method looks at every training example in the entire training set on every step and hence it is known as **batch gradient descent**. As it is processing the entire data as a batch to make an update, it is disadvantageous when the dataset is large. As it will be scanning through the entire dataset multiple times to converge, which is quite expensive.

Stochastic Gradient Descent

Stochastic gradient descent is an alternate method for batch gradient descent, when the data set is large. Here we loop through each training example and and make the updates to θ w.r.t each training example.
Loop {
for $i = 1$ to n, {

$\theta := \theta + \alpha(y^{(i)} - h(x^{(i)}))x^{(i)}$
}
}
This is also known as **incremental gradient descent**. Stochastic gradient descent gets θ "close" to the minimum much faster than batch gradient descent. However, that it may never "converge" to the minimum, and the parameters θ will keep oscillating around the minimum of $J(\theta)$; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum.

Normal Equation

Instead of doing gradient descent, there is a way to solve for the optimal parameters θ without needing to use an iterative algorithm. This works only for linear regression.

Given a training set, define the **design matrix** X to be the n by d matrix (actually n by $d+1$, if we include the intercept term) that contains the training examples input values in its rows:

$$X = \begin{bmatrix} -(x^{(1)})^T- \\ -(x^{(2)})^T- \\ \vdots \\ -(x^{(n)})^T- \end{bmatrix}.$$

Also, let $\vec{y}$ be the n-dimensional vector containing all the target values from the training set:

$$\vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

Now, using the fact that for a vector z, we have that $z^T z = \sum z_i^2$:

$$\begin{aligned} \frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) &= \frac{1}{2}\sum_{i=0}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2 \\ &= J(\theta) \end{aligned}$$

Finally, to minimize J, let's find its derivatives with respect to θ.

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) \\ &= \frac{1}{2}\nabla_\theta((X\theta)^T X\theta - (X\theta)^T\vec{y} - \vec{y}^T(X\theta) + \vec{y}^T\vec{y}) \\ &= \frac{1}{2}\nabla_\theta(\theta^T(X^TX)\theta - \vec{y}^T(X\theta) - \vec{y}^T(X\theta)), \quad \text{as} \;\; a^Tb = b^Ta \\ &= \frac{1}{2}\nabla_\theta(\theta^T(X^TX)\theta - 2(X^T\vec{y})^T\theta) \\ &= \frac{1}{2}(2X^TX\theta - 2X^T\vec{y}) \\ &= X^TX\theta - X^T\vec{y} \end{aligned}$$

In the fifth step we have used the fact that $\nabla_x b^T x = b$ and $\nabla_x x^T A x = 2Ax$ for symmetric matrix A. ”). To minimize J, we set its derivatives to zero, and obtain the **normal equation**:

$$X^TX\theta = X^T\vec{y}.$$

Thus, the value of θ that minimizes $J(\theta)$ is given by the equation

$$\theta = (X^TX)^{-1}X^T\vec{y}.$$

Note that X need not be square matrix so we cannot write $(X^TX)^{-1} = X^{-1}(X^T)^{-1}$. And if X^TX is not invertible then it implies that features are linearly dependent, in that case we can either reduce the number of features by cutting down the redundant features or we can take the pseudo inverse.

Probabilistic interpretation

In this section we will explain, why the least-squares cost function J, be a reasonable choice? Let us assume that the target variables and the inputs are related via the equation:

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)},$$

where $\epsilon^{(i)}$ is an error term that captures either unmodeled effects or random noise. Let us further assume that the $\epsilon^{(i)}$ are distributed IID (independently and identically distributed) and we are also gonna assume that, $\epsilon^{(i)}$ follows the Gaussian distribution (also called a Normal distribution) with mean zero and variance σ^2. We can write this assumption as "$\epsilon^{(i)}$~$N(0, \sigma^2)$". I.e., the probability density of $\epsilon^{(i)}$ is given by

$$p(\epsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(\epsilon^{(i)})^2}{2\sigma^2}\right).$$

This implies that

$$p(y^{(i)}|x^{(i)};\theta) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y^{(i)}-\theta^T x^{(i)})^2}{2\sigma^2}\right).$$

The notation $p(y^{(i)}|x^{(i)};\theta)$ indicates that this is the distribution of $y^{(i)}$ given $x^{(i)}$ and parameterized by θ. Note that we should not condition on θ, as it is not a random variable. Now, the above equation is a probability density function of $y^{(i)}$, which is a Gaussian distribution with mean $\theta^T x^{(i)}$. As we are more interested in viewing the above pdf as function of θ, lets call it **likelihood** function:

$$L(\theta) = L(\theta; X, \vec{y}) = p(\vec{y}|X;\theta).$$

Note that by the independence assumption on the $\epsilon^{(i)}$'s and hence also the $y^{(i)}$'s given the $x^{(i)}$'s, this can also be written

$$\begin{aligned} L(\theta) &= \prod_{i=1}^{n} p(y^{(i)}|x^{(i)};\theta) \\ &= \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y^{(i)}-\theta^T x^{(i)})^2}{2\sigma^2}\right). \end{aligned}$$

The principal of maximum likelihood says that we should choose θ so as to make the data as high probability as possible. I.e., we should choose θ to maximize $L(\theta)$. Instead of maximizing $L(\theta)$, we can also maximize any strictly increasing function of $L(\theta)$. In particular, the derivations will be a bit simpler if we instead maximize the **log likelihood** $L(\theta)$:

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y^{(i)}-\theta^T x^{(i)})^2}{2\sigma^2}\right) \\ &= \sum_{i=0}^{n} \log \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y^{(i)}-\theta^T x^{(i)})^2}{2\sigma^2}\right) \\ &= n\log\frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2}\frac{1}{2}\sum_{i=0}^{n}(y^{(i)}-\theta^T x^{(i)})^2. \end{aligned}$$

Hence maximizing $l(\theta)$ is same as minimizing

$$\frac{1}{2}\sum_{i=0}^{n}(y^{(i)}-\theta^T x^{(i)})^2,$$

which we defined to be our cost function $J(\theta)$. Thus, this is one set of assumptions under which least-squares regression can be justified as a very natural method that's just doing maximum likelihood estimation. (Note however that the probabilistic assumptions are by no means are necessary conditions, there are other natural assumptions that can also be used to justify it. Note also that, in our previous discussion, our final choice of θ did not depend on what was σ^2 , and indeed we'd have arrived at the same result even if σ^2 were unknown. If we can assume the error terms are IID's and follow Gaussian and if we want to use maximum likelihood estimation then we should use Least squares.

- Independence: All observations are independent of one another.
- Normality: The distribution of Y is assumed to be normal.
- The variance of the residuals is constant.

Locally Weighted Linear Regression

In a normal linear regression to fit θ, we try to minimize $\sum_i (y^{(i)}-\theta^T x^{(i)})^2$. But in Locally Weighted Linear Regression, we will be looking at neighboring training examples and ignore the rest. This can be achieved by simply assigning weights to each term in the cost function i.e. we try to fit θ to minimize $\sum_i w^{(i)}(y^{(i)}-\theta^T x^{(i)})^2$,

where $w^{(i)}$ is the non-negative weights. A fairly standard choice for the weights is

$$w^{(i)} = exp\left(- \frac{(x^{(i)} - x)^2}{2\tau^2} \right)$$

The parameter τ controls how quickly the weight of a training example falls off with distance of its $x^{(i)}$ from the query point x, τ is called the **bandwidth** parameter, and is also something that we need to experiment. A Locally Weighted Linear Regression is a non-parametric algorithm.

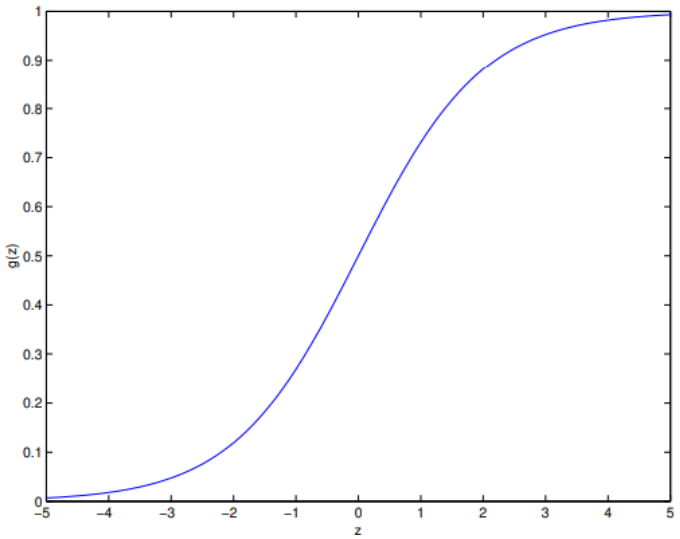
> **Non parametric algorithms**
>
> A **non-parametric algorithm** is a type of machine learning algorithm that does not make strong assumptions about the form or structure of the underlying data distribution. Unlike parametric algorithms, which are defined by a fixed number of parameters (e.g., coefficients in linear regression), non-parametric algorithms can grow in complexity as they see more data, and their structure is often determined directly from the data itself.

Logistic Regression

In Linear regression the value of y can take continuous values. Now if want y to take some discrete especially 0 or 1. **Logistic Regression** is a supervised learning algorithm commonly used for **classification tasks**, especially binary classification. In Logistic Regression we choose our hypothesis function to be

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}},$$

where $g(z) = \frac{1}{1+e^{-z}}$ is called logistic function or sigmoid function.



Other functions that smoothly increase from 0 to 1 can also be used, but for a couple of reasons that we'll see later. Before moving on, here's a useful property of the derivative of the sigmoid function, $g'(z) = g(z)(1 - g(z))$. Let

$$P(y = 1|\ x; \theta) = h_\theta(x),\ \text{then}$$
$$P(y = 0|\ x; \theta) = 1 - h_\theta(x)$$

This is can be written in a compact fashion as

$$p(y \mid x; \theta) = (h_\theta(x))^y (1 - h_\theta(x))^{1-y}$$

Assuming that the n training examples were generated independently, we can then write down the likelihood of the parameters as

$$
\begin{aligned}
L(\theta) &= p(\vec{y}|\ X;\theta) \\
&= \prod_{i=1}^{n} p(y^{(i)}|\ x^{(i)};\theta)
\end{aligned}
$$

It will be easier to maximize the log likelihood rather than the likelihood itself

$$
\begin{aligned}
l(\theta) = \log L(\theta) &= \sum_{i=1}^{n} y^{(i)} \log h(x^{(i)}) + (1-y^{(i)}) \log(1-h(x^{(i)})) \\
\Rightarrow \frac{\partial}{\partial\theta_j} l(\theta) &= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)} \right) \frac{\partial}{\partial\theta_j} g(\theta^T x) \\
&= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)} \right) g(\theta^T x)(1-g(\theta^T x)) \frac{\partial}{\partial\theta_j} \theta^T x \\
&= \big(y(1-g(\theta^T x)) - (1-y) g(\theta^T x) \big) x_j \\
&= (y - h_\theta(x)) x_j
\end{aligned}
$$

This therefore gives us the stochastic gradient ascent rule:

$$
\theta_j := \theta_j + \alpha \big(y^{(i)} - h_\theta(x^{(i)}) \big) x_j^{(i)}
$$

If we compare this to the LMS update rule, we see that it looks identical; but this is not the same algorithm, because $h_\theta(x^{(i)})$ is now defined as a non-linear function of $\theta^T x^{(i)}$.

Loss Functions

Mean Square Error (MSE)

Mean Squared Error (MSE) is one of the most widely used loss functions in regression tasks. It measures the average of the squared differences between predicted and actual values. Mathematically, it is defined as:

$$
MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i) 2
$$

Drawbacks of MSE

- **Sensitivity to Outliers**: Because errors are squared, outliers can disproportionately influence the loss. A single large error can dominate the gradient updates, potentially leading to poor model performance or slow convergence unless carefully managed.
- **May Overfit to Noise**: In datasets with noisy or incorrect labels, MSE may cause the model to overfit to these anomalies in an attempt to minimize large squared errors.

Mean Absolute Error (MAE) / L1 Loss

MAE, also known as L1 loss, computes the average of the absolute differences between predicted and actual values. It is defined mathematically as:

$$
MAE = \frac{1}{n} \sum_{i=1}^{n} \ | \ y_i - \hat{y}_i \ |
$$

One of the key advantages of MAE is its **robustness to outliers**. Because it uses absolute differences rather than squared ones, large errors do not disproportionately influence the total loss compared to Mean Squared Error (MSE)

Drawbacks of MAE

- It is **not differentiable at zero** (when the prediction exactly matches the true value), which complicates gradient-based optimization methods like Stochastic Gradient Descent.
- MAE applies uniform weighting to all errors, meaning it treats small and large errors with equal relative penalty, which may slow convergence compared to quadratic loss functions.

Huber Loss / Smooth L1 Loss

Huber loss is a hybrid loss function designed to combine the best properties of both MAE and MSE. It behaves quadratically (like MSE) for small errors and transitions to linear behavior (like MAE) for larger errors, depending on a threshold parameter δ (delta). The mathematical formulation is:

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & if|y - \hat{y}| \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & otherwise \end{cases}$$

This dual behavior gives Huber loss several advantages:

- **Robustness to outliers**: Like MAE, it reduces the influence of large errors by applying linear penalties beyond δ, making it less sensitive to extreme values than MSE.
- **Differentiability**: Unlike MAE, Huber loss is fully differentiable at all points, including at the transition point δ, which facilitates efficient optimization using gradient descent methods
- **Faster convergence for small errors**: For residuals smaller than δ, the quadratic component helps the model converge more quickly, similar to MSE
- **Smooth optimization landscape**: The transition from quadratic to linear loss helps avoid issues like vanishing or exploding gradients, leading to more stable training.

Cross Entropy loss

Cross-entropy loss measures the difference between two probability distributions: the **true distribution** (the actual labels) and the **predicted distribution** (the model's output probabilities)

- For multi class classification the cross entropy loss is given by

  $$\text{Cross-Entropy } (y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i),$$

  where y_i and $\hat{y}_i$ are true label and predicted label, and for binary classification we have

  $$\text{Cross-Entropy } (y, \hat{y}) = -[y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})]$$

  It gives a **bigger penalty when the model is very sure but wrong**, compared to when it's only a little wrong.

Drawbacks of Cross Entropy loss

- **Sensitive to Incorrect Confident Predictions**, like If the model is very confident in a wrong prediction (e.g., outputting 0.99 for the wrong class), the loss becomes very high. This can lead to instability during training.
- It assumes that the true labels are accurate and one-hot (i.e., absolute certainty about the correct class). No room for label noise or uncertainty.
- Suffers in Imbalanced Datasets the model may get biased toward the majority class without additional techniques like class weighting.
- Tends to push the model to output probabilities very close to 0 or 1, which can harm generalization.