Machine Learning Course

Lecture 3 - Linear and Logistic Regression

Lecturer: Haim Permuter *Scribe: Ziv Aharoni*

Throughout this lecture we talk about how to use regression analysis in Machine Learning problems. First, we introduce regression analysis in general. Then, we talk about Linear regression, and we use this model to review some optimization techniques, that will serve us in the remainder of the course. Finally, we will discuss classification using logistic regression and softmax regression. Parts of this lecture are based on lecture notes from Stanfords CS229 machine learning course by Andrew NG[1]. This lecture assumes you are familiar with basic probability theory. The notation here is similar to Lecture 1.

I. An introduction to Regression vs Classification

Regression analysis, a branch in statistical modelling, is a statistical process for estimating the relationship between Y, the dependent variable, given observations of X, the independent variables. Similarly, in Supervised Learning, we seek to build a statistical model (hypothesis) that maps optimally each $x \in \mathcal{X}$ to $y \in \mathcal{Y}$ with respect to the underlying statistics of X, Y. If $y \in \mathcal{Y}$ can take values from a discrete group of unordered values, we refer the problem as classification. Usually in classification there is no meaning of distance between different labels Y. However, if $y \in \mathcal{Y}$ can take values from a continuous interval of values, we refer the problem as regression. In regression there should be a meaning of distance between different values of Y.

In machine learning, the commonly used terminology is to call problems with discrete Y as classification problems, and problems of continuous Y as regression problems. In the remainder of the course, we will use this convention and will clarify the terminology if conflicts will arise.

Let’s sharpen the differences between regression and classification by an example of weather forecasts. We set our independent variables as X_1, the temperature in given places, and X_2, the humidity in a given places. When we define our dependent variable to be the amount of precipitation, we will choose a hypothesis that will map X_1, X_2 to the actual precipitation amount. In this case, we refer the problem as a regression problem. When we define the dependent variable to be the ”weather type” (clear, cloudy, rainy, etc), our hypothesis will use the independent variables to classify X_1, X_2 to the correct group of weather types. In this case we refer the problem as classification problem.

Now, after we discussed about the terminology, let’s delve into a commonly used regression model, *Linear Regression*.

II. Linear Regression

In linear Regression, as implied from its name, we try to estimate the value of y with a linear combination of the new feature vector $x \in \mathbb{R}^m$ with respect to our training set $\{\ (x^{(1)}, y^{(1)}),\ (x^{(2)}, y^{(2)}), \dots,\ (x^{(n)}, y^{(n)})\ \}$. Let’s define $h_\theta(x)$ as the hypothesis for y from x with respect to the parameters θ, where $\theta \in \mathbb{R}^{m+1}$ is defined by

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_m \end{bmatrix}.$$

Hence, $h_\theta(x)$ is given by

$$h_\theta(x) = \theta_0 + x_1\theta_1 + \cdots + x_m\theta_m, \tag{1}$$

where θ_0 is the bias term. For convenience let’s define $x_0 = 1$ so we can rewrite (1) as

$$h_\theta(x) = \sum_{k=0}^{m} x_k\theta_k = \theta^T x. \tag{2}$$

Our Goal is to find the linear combination coefficients, θ, that will yield the hypothesis which maps $x^{(i)}$ to $y^{(i)}$ most accurately. In order to do that, we will have to assume that the training set are representative of the whole problem. If it is not the case, we should enrich our training set appropriately (if it is possible). Next, we need to choose θ that will make $h_\theta(x^{(i)}) \approx y^{(i)}$, for all $i = 1, \dots, n$ (the $\approx$ sign stands for approximately equal). By achieving a hypothesis which maps accurately $x^{(i)}$ to $y^{(i)}$ for the entire training set, combined with the assumption that our training set is representative of the whole problem, we can say that our hypothesis is optimal, with the limits of a linear model.

Now, given the training set, how should we choose θ? In order to do that, we need to define a function which measures the error between our hypothesis based on the $x^{(i)}$'s and the corresponding $y^{(i)}$'s for all $i = 1, \ldots, n$. Let's define a *Cost function* that measures the error of our hypothesis.

Definition 1 (MSE Cost Function)

MSE Cost Function, $f : \mathbb{R}^{(m+1)\times n} \to \mathbb{R}$ is defined by

$$\begin{aligned} C(\theta) &= \frac{1}{2n}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2 \\ &= \frac{1}{2n}\sum_{i=1}^{n}(\theta^T x^{(i)} - y^{(i)})^2 \end{aligned} \tag{3}$$

We can see that when $h_\theta(x^{(i)}) \approx y^{(i)}$ for all $i = 1, \ldots, n$, then our cost function satisfies $C(\theta) \approx 0$. Moreover, every term in our cost function is positive, therefore $C(\theta) \approx 0 \iff h_\theta(x^{(i)}) \approx y^{(i)}$ for all $i = 1, \ldots, n$. These are two important properties, that lead us to the conclusion that finding θ^* that minimizes the cost function, eventually forces $h_\theta(x^{(i)}) \approx y^{(i)}$ for all $i = 1, \ldots, n$. By achieving that, we are essentially accomplishing our goal.

Let's formulate our goal,

$$\theta^* = \underset{\theta \in \mathbb{R}^{m+1}}{\operatorname{argmin}} C(\theta). \tag{4}$$

Note that if we find θ^*, our linear regression model can be used on new sample x by the following equation:

$$h_\theta(x) = (\theta^*)^T x. \tag{5}$$

Now, after formulating our problem, let's survey some minimization methods to serve us in minimizing our cost function.

III. Cost Function Minimization

In this section we discuss about some various ways for minimizing $C(\theta)$ with respect to θ. The methods that we focus on are:

1) Gradient Descent:
   - Batch.
   - Stochastic (Incremental).
2) Analytically.

Definition 2 (Gradient Descent algorithm)

The *Gradient Descent algorithm* finds a minimum of a function by an iterative process with the following steps:

1) Set θ_0 as the initial guess for θ^*.
2) Repeat the following updating rule until one of the following stopping condition:
   - no change in parameters, i.e., $d(\theta^{(t+1)}, \theta^{(t)}) < \delta$, where δ is defined to be the convergence tolerance, $d(\cdot)$ is defined to be a distance norm.
   - no change in the cost. i.e., $|C(\theta^{(t)}) - C(\theta^{(t+1)})| < \delta$.
   - number of iteration fixed, i.e., $t < T$, where T is the number of iterations.

$$\theta^{(t+1)} := \theta^{(t)} - \alpha \nabla_\theta C(\theta^{(t)}) \tag{6}$$

   where:
   - $t = 0, 1, \ldots$ is the iteration number.
   - α is the "Learning Rate" and it determines how large step the algorithm takes every iteration.
   - ∇_θ is the gradient of $C(\theta)$ with respect to the parameters θ and is defined by:

$$\nabla_\theta C(\theta) = \begin{bmatrix} \frac{\partial C(\theta)}{\partial \theta_0} & \frac{\partial C(\theta)}{\partial \theta_1} & \ldots & \frac{\partial C(\theta)}{\partial \theta_m} \end{bmatrix}^T$$

Intuitively, we can think of the algorithm as guessing an initial place in the domain. Then, it calculates the direction and magnitude of the steepest descent of the function for the current θ ($\nabla_\theta C(\theta)$) and finally, in the domain, it takes a step, proportional to the descent magnitude and to a fixed predefined step size α to the next θ. In case we want to use the algorithm for maximization, we will exchange the '$-$' sign with a '$+$' sign and then we will take step in the direction of the steepest ascent of the function. In this case the algorithm is called *gradient ascent*. Note that, generally, the algorithm searches 'a' minimum and not 'the' minimum of the function. If we want to claim that the algorithm finds the global minimum of the function we will need to prove that the function has only one minimum, which means that the function is convex.

Here are some examples of gradient descent as it runs to minimize convex or non convex functions.

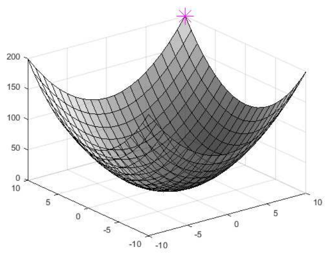

Fig. 1: initial guess of GD on a convex function

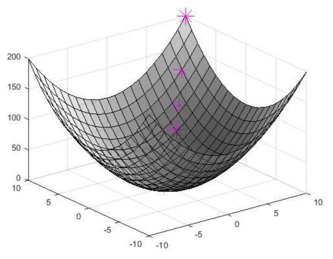

Fig. 2: 3 iterations of GD on a convex function

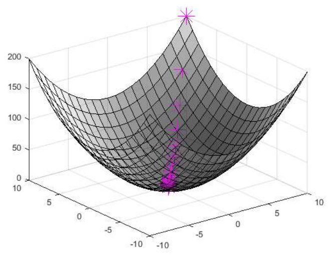

Fig. 3: 64 iterations of GD on a convex function

The leftmost figure shows us the value of $f(\theta_0)$. The middle figure shows the algorithm progress after three iterations and the rightmost figure shows the algorithm after achieving convergence. We can visualize the fact that gradient descent applied on convex function will converge to the global minimum of the function, assuming that α is relatively small.

Next, let's examine the performance of gradient descent on an non-convex function given by $f(x, y) = x^5 - 3x^2 + y^4 + 7y^2$. This function has no global minimum nor maximum but has a local minimum at $(1.06266, 0)$. At the three upper figures we can see that for some initial guess the algorithm converged to the local minimum of the function. At the three lower figures we can see that for some other initial guess the algorithm began to make progress in the domain to the area where $f(x, y) \to -\infty$. This approves the fact that gradient descent applied on a non-convex function generally

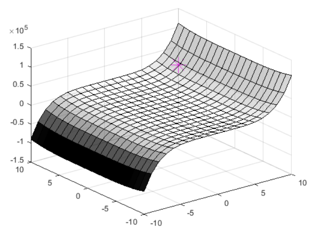

Fig. 4: initial guess of GD on a non-convex function

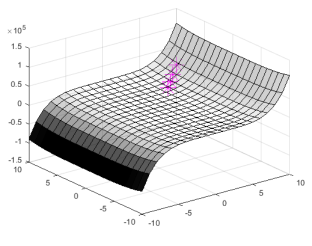

Fig. 5: 3 iterations of GD on a non-convex function

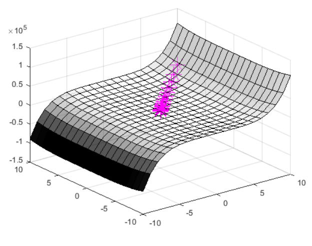

Fig. 6: 64 iterations of GD on a non-convex function

doesn't converge to the global minimum of the function.

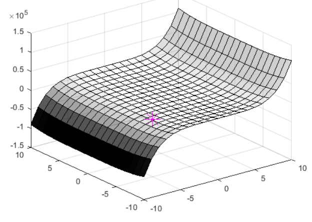

Fig. 7: initial guess of GD on a non-convex function

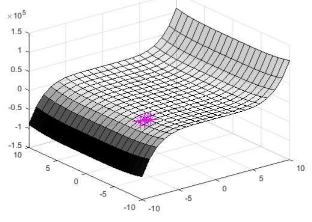

Fig. 8: 3 iterations of GD on a non-convex function

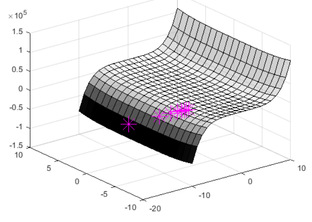

Fig. 9: 7 iterations of GD on a non-convex function

Therefore, we can conclude that by choosing a convex cost function to our problems, we can assure finding its global minimum by an iterative process such as gradient descent. Luckily, the MSE cost function is a convex function with respect to θ so we can assure that gradient descent will find its global minimum.

Now, let's implement the gradient descent method on our linear regression problem. For doing that, we have to calculate the term $\nabla_\theta C(\theta)$. Let's find an expression for the j^{th} coordinate of $\nabla_\theta C(\theta)$ in our problem.

$$\begin{aligned}\frac{\partial C(\theta)}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j}\frac{1}{2n}\sum_{i=1}^{n}(\theta^T x^{(i)} - y^{(i)})^2 \\ &\overset{(a)}{=} \frac{1}{2n}\sum_{i=1}^{n}\frac{\partial}{\partial \theta_j}(\theta^T x^{(i)} - y^{(i)})^2\end{aligned}$$

$$\begin{aligned}
&\overset{(b)}{=} \frac{1}{2n}\sum_{i=1}^{n} 2(\theta^T x^{(i)} - y^{(i)})x_j^{(i)} \\
&\overset{(c)}{=} \frac{1}{n}\sum_{i=1}^{n} (\theta^T x^{(i)} - y^{(i)})x_j^{(i)} \qquad (7)
\end{aligned}$$

Where in (a) we used the linearity of the derivative, In (b) and (c) we differentiated and simplified the expression. By inserting Equation (7) in Equation (6) we get the following update rule which is given by

$$\theta_j^{(t+1)} := \theta_j^{(t)} - \frac{\alpha}{n}\sum_{i=1}^{n} (\theta^T x^{(i)} - y^{(i)})x_j^{(i)}, \ \forall j \qquad (8)$$

Now lets understand the intuition of Eq. (8). First, for simplicity, lets examine it for $n = 1$. We get $\theta_j^{(t+1)} := \theta_j^{(t)} - \alpha(\hat{y}^{(i)} - y^{(i)})x_j^{(i)}$, where $\hat{y}^{(i)}$ is the estimator of $y^{(i)}$ using the linear regression model of the current iteration. Now, if denote by $error^{(i)} = (\hat{y}^{(i)} - y^{(i)})$, then the update is $\theta_j^{(t+1)} := \theta_j^{(t)} - \alpha \cdot error^{(i)} \cdot x_j^{(i)}$. Namely, update $\theta_j^{(t)}$ by a linear offset that is proportional to the estimation error multiply by the input. And if now we have $n > 1$, then we average over all the updates, so the noise would be less effective at each iteration.

It can be easily verified that at every update, the algorithm uses the entire training set to compute the gradient. For that reason we call this way of using gradient descent as *batch gradient descent*. Alternatively, we can use only one training example to update θ every iteration. The update rule at this case is given by,

for i=1 to n, $\{$

$$\theta_j^{(t+1)} := \theta_j^{(t)} - \alpha(\theta^T x^{(i)} - y^{(i)})x_j^{(i)}, \ \forall j \qquad (9)$$

$\}$

In this case, we call the algorithm *Stochastic Gradient Descent (or Incremental Gradient Descent)*. Next, let talk about our second method for optimizing $C(\theta)$. Our next method uses the fact that finding an extremum of an a function is equal to finding its derivative's zeros.

Analytic Minimization

In some cases, there could be found an analytic, closed-formed solution for θ^* which minimizes the cost function. This is not always the case, but for linear regression there is an analytic solution. In order to find a solution we take the derivatives of the cost function and set them to zero. In terms of convenience, let's make some notations that will ease our process of finding θ^*.

Let X be the *design matrix*, whose rows represent the training example's index, and its columns represent the feature's index. E.g, X_{ij} represents the j^{th} feature of the i^{th} training example. $X \in \mathbb{R}^{m+1\times n}$ is given by

$$X = \begin{bmatrix} | & | & & | \\ (x^{(1)}) & (x^{(2)}) & \cdots & (x^{(n)}) \\ | & | & & | \end{bmatrix}.$$

Now, let $\vec{y}$ be the vector of the target values, such as $\left(\ \vec{y}\ \right)_i = y^{(i)}$. This means that $\vec{y} \in \mathbb{R}^n$ is given by

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

Next, given that $h_\theta(x^{(i)}) = (x^{(i)})^T\theta$, we can express the term $h_\theta(x^{(i)}) - y^{(i)}$ by

$$h_\theta(x^{(i)}) - y^{(i)} = \left(\ X^T\theta - \vec{y}\ \right)_i.$$

Combined with the fact that for some vector $\vec{a}$, the property $\vec{a}^T\vec{a} = \sum_i a_i^2$ holds, we can conclude that

$$\frac{1}{2n}\left(X^T\theta - \vec{y}\right)^T\left(X^T\theta - \vec{y}\right) = \frac{1}{2n}\sum_{i=1}^{n}(\theta^T x^{(i)} - y^{(i)})^2 \triangleq C(\theta). \tag{10}$$

Now, let's take derivatives with respect to θ.

$$\begin{aligned}\nabla_\theta C(\theta) &\overset{(a)}{=} \nabla_\theta\left\{\frac{1}{2n}\left(X^T\theta - \vec{y}\right)^T\left(X^T\theta - \vec{y}\right)\right\} \\ &\overset{(b)}{=} \frac{1}{2n}\nabla_\theta\left\{\theta^T X X^T\theta - \theta^T X\vec{y} - \vec{y}^T X^T\theta + \vec{y}^T\vec{y}\right\}\end{aligned}$$

$$
\begin{aligned}
&\overset{(c)}{=} \frac{1}{2n}\nabla_\theta \left\{\operatorname{tr}\theta^T X X^T \theta - \operatorname{tr}\theta^T X\vec{y} - \operatorname{tr}\vec{y}^T X^T \theta + \operatorname{tr}\vec{y}^T\vec{y}\right\} \\
&\overset{(d)}{=} \frac{1}{2n}\nabla_\theta \left\{\operatorname{tr}\theta^T X X^T \theta - 2\operatorname{tr}\vec{y}^T X^T \theta\right\} \\
&\overset{(e)}{=} \frac{1}{2n}\left(2XX^T\theta - 2X\vec{y}\right) \\
&= \frac{1}{n}\left(XX^T\theta - X\vec{y}\right)
\end{aligned}
$$

Where in (a) we used the Equation (10). In (b) we simplified the cost function and used the linearity of the gradient operator. In (c) we used the fact that a trace of a real number is the number itself, and we eliminated the $\vec{y}^T\vec{y}$ term, because it has no dependency on θ. In (d) we used the fact that $\operatorname{tr} a = \operatorname{tr} a^T$. In (e) we used the following rules of matrix calculus,

- $\nabla_A \operatorname{tr} A^T B A = (B + B^T)A$, where $A = \theta, B = XX^T$.
- $\nabla_A \operatorname{tr} B^T A = B$, where $A = \theta, B = X\vec{y}$.

By setting the gradient to the zero vector we get that

$$
XX^T\theta^* = X\vec{y}
$$

$$
\Downarrow
$$

$$
\theta^* = \left(XX^T\right)^{-1} X\vec{y} \tag{11}
$$

Where the term $\left(XX^T\right)^{-1} X$ is an m+1-by-n matrix and is called the *pseudo-inverse* of X. So, by taking derivatives and setting them to zero, we can try finding explicitly θ^* for any model that maps $x \in \mathcal{X}$ to $y \in \mathcal{Y}$, or any type of cost function. So, why should we use an iterative process that finds θ^*? The first reason is that inverting matrix could be very inaccurate calculation in the computer due to rounding errors. The second reason is that inverting an m-by-m matrix takes $\sim m^3$ operations, and for large m it could be more time-efficient to calculate θ^* with an iterative process. The Third reason is that it is sometimes not possible to find a closed-form solution for θ^* due to the complexity of the cost function nor the hypothesis'.

IV. Probabilistic Interpretation

In the previous sections, when using the linear regression on a regression problem, we used some heuristic explanations for choosing the cost function to be the quadratic cost function, and explained why we minimizes it. Now, let's make some probabilistic assumptions on our problem, then let's try to back up our heuristic explanations with probabilistic justifications, and specifically we will use the Maximum Likelihood principle that we introduced in the first lecture.

Let's assume that our target variables, the $y^{(i)}$'s, are given by the equation

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}. \tag{12}$$

Where $\epsilon^{(i)}$ is the error, which can be interpreted as random noise, inaccuracy in measurements, limitation of the linear model etc. Moreover, let's assume that the $\epsilon^{(i)}$'s are i.i.d (independently and identically distributed) and its distribution is given by

$$\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2), \ \ \forall i \ . \tag{13}$$

Now, lets examine the probability of $P(y^{(i)}|x^{(i)};\theta)$ (the semi-colon indicates that $y^{(i)}$ is parameterized by θ, since θ is not a random variable). Given $x^{(i)}, \theta$, the term $\theta^T x^{(i)}$ is deterministic and hence $P(y^{(i)}|x^{(i)};\theta)$ is a Normal distribution with expectation $\theta^T x^{(i)}$ and variance σ^2, i.e., $\mathcal{N}(\theta^T x^{(i)}, \sigma^2), \ \ \forall i$. Let $L(\theta)$ be the *likelihood function*, which is given by

$$L(\theta) = p(\vec{y}|X;\theta). \tag{14}$$

The likelihood function represents the probability for all $y^{(i)}$'s given all $x^{(i)}$'s. Intuitively it measures how likely is that $y^{(i)}$ is the correct value of $x^{(i)}$ for all i, under the probabilistic assumptions we made. Due to that the entire training set is given, the likelihood function depends exclusively on θ. Likewise, note that due to the fact $\epsilon^{(i)}$'s are i.i.d,we have,

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

$$\overset{(b)}{=} \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right). \tag{15}$$

Where in (a) we used the independency and in (b) we used the identical distribution. Given the design matrix X (or equivalently $x^{(1)}, \ldots, x^{(n)}$) and the corresponding $y^{(i)}$'s, we want to make $p\left(\vec{y}|X;\theta\right)$ as high as possible (because we know that $y^{(i)}$ is the correct value that corresponds to $x^{(i)}$ for all i's) by adjusting θ. Therefore we derived the *maximum likelihood* criteria, which is maximizing $L(\theta)$. Instead of maximizing $L(\theta)$, we can maximize any strictly increasing function of $L(\theta)$. Let $l(\theta)$ be the *log likelihood* function which is given by $l(\theta) = \log L(\theta)$. Now let's maximize $l(\theta)$.

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right) \\ &\overset{(a)}{=} \sum_{i=1}^{n} \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right) \\ &\overset{(b)}{=} \sum_{i=1}^{n} \log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2} \\ &\overset{(c)}{=} n \log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{i=1}^{n} \left(y^{(i)} - \theta^T x^{(i)}\right)^2 \end{aligned} \tag{16}$$

Where in (a), (b) we used the logarithm properties, and in (c), we simplified the expression. So in fact, maximizing the likelihood function is equivalent to minimizing the term

$$\sum_{i=1}^{n} \left(y^{(i)} - \theta^T x^{(i)}\right)^2. \tag{17}$$

Note that the exact value of the error's variance, σ, doesn't affect the values of θ^*. In fact, it tells us that adjusting θ can't minimize the errors which are caused by the $\epsilon^{(i)}$'s. Finally, we can conclude that under the probabilistic assumptions we made earlier, we derived the same goal, minimizing the term (17) which is exactly $C(\theta)$ as we defined earlier.

V. Logistic Regression

Logistic regression, as opposed to its misleading name, is used for binary classification problems, which means that $y^{(i)}$ can take values from the set $\{0, 1\}$. As we clarified in the introduction section, the term ”regression” is rooted in regression analysis and not in machine learning’s conventions. Note that for classification problems we cannot use the linear regression model because in that case, y could get any value in the interval $(-\infty, \infty)$. Therefore, we should try other attitude to solving this problem.

Let’s adopt the probabilistic interpretation from last section. First, we will make some probabilistic assumptions at the problem and then, we will use the maximum likelihood criteria to adjust θ.

Assume that $P(y^{(i)}|x^{(i)};\theta)$ is $Bernoulli(\phi^{(i)})$, where $\phi^{(i)}$ is the Bernoulli parameter for the i^{th} training example. Therefore,

$$p\left(y^{(i)}|x^{(i)};\theta\right) = (\phi^{(i)})^{y^{(i)}}(1-\phi^{(i)})^{1-y^{(i)}}.$$

Now we use our hypothesis to estimate the probability that $x^{(i)}$ belong to a certain class. Let’s choose our hypothesis to be

$$\begin{aligned}\hat{\phi^{(i)}} &= h_\theta(x^{(i)}) \\ &= \sigma\left(\theta^T x^{(i)}\right) = \frac{1}{1+e^{-\theta^T x^{(i)}}}.\end{aligned} \tag{18}$$

Where $\sigma(z) = \frac{1}{1+e^{-z}}$ is called the *sigmoid function* or the *logistic function*. Note that $\sigma(z)$ can output values in the interval $[0, 1]$, such as

$$\begin{aligned}&\sigma(z) \to 1,\ z \to \infty \\ &\sigma(z) = \frac{1}{2},\ z = 0 \\ &\sigma(z) \to 0,\ z \to -\infty\end{aligned}$$

Moreover, $\sigma(z)$ is differentiable over $\mathbb{R}$ and its derivative satisfies the following property

$$\begin{aligned}\sigma'(z) &= -\frac{1}{(1+e^{-z})^2}\left(-e^{-z}\right) \\ &= \frac{1}{1+e^{-z}}\frac{e^{-z}}{1+e^{-z}}\end{aligned}$$

$$
\begin{aligned}
&= \left(\frac{1}{1+e^{-z}}\right)\left(1-\frac{1}{1+e^{-z}}\right) \\
&= \sigma(z)\left(1-\sigma(z)\right) \qquad (19)
\end{aligned}
$$

Our last assumption is that the training set was generated independently. Backed up with these assumptions, let's adjust θ to maximize the classification accuracy over the training set, hopefully to yield accurate results for unseen x. Let's use the maximum likelihood criteria to adjust θ. The likelihood function in this case is given by

$$
\begin{aligned}
L(\theta) &= p\left(y^n|x^n;\theta\right) \\
&= p\left(y^{(1)},\ldots,y^{(n)}|x^{(1)},\ldots,x^{(n)};\theta\right) \\
&= \prod_{i=1}^{n} p\left(y^{(i)}|x^{(i)};\theta\right) \\
&= \prod_{i=1}^{n} (\sigma\left(\theta^T x^{(i)}\right))^{y^{(i)}}(1-\sigma\left(\theta^T x^{(i)}\right))^{1-y^{(i)}} \qquad (20)
\end{aligned}
$$

As we stated in the previous section, we can maximize θ over any strictly increasing function of $L(\theta)$. So, to simplify our calculations, let's maximize the log likelihood function

$$
\begin{aligned}
l(\theta) &= \log\prod_{i=1}^{n} (\sigma\left(\theta^T x^{(i)}\right))^{y^{(i)}}(1-\sigma\left(\theta^T x^{(i)}\right))^{1-y^{(i)}} \\
&= \sum_{i=1}^{n} y^{(i)}\log(\sigma\left(\theta^T x^{(i)}\right)) + (1-y^{(i)})\log(1-\sigma\left(\theta^T x^{(i)}\right)) \qquad (21)
\end{aligned}
$$

The log likelihood has an interpretation of Maximum likelihood criteria as developed in (20)-(21) and its also related to cross entropy cost function.

Consider the following cost:

$$
C_n(\theta) = -\sum_{i=1}^{n} \log \hat{P}(y^{(i)}|x^{(i)},\theta) \qquad (22)
$$

By the law of large number this cost convergence to

$$
\begin{aligned}
\lim_{n\to\infty} C_n(\theta) &= -E[\log P(Y|X,\theta) \\
&= -\sum_{x,y} p(y,x)\log p(y|x.\theta)
\end{aligned}
$$

$$= -\sum_x p(x) \sum_y p(y|x) \log p(y|, \theta)$$
$$= \sum_x p(x) H(p_{Y|X=x}, p_{Y|X=x,\theta}), \tag{23}$$

where last equality follows from the fact that for a fixed x, $-\sum_y p_{Y|X}(y|x) \log p_{Y|X,\theta}(y|x, \theta)$ is the cross entropy $H(p_{Y|X=x}, p_{Y|X=x,\theta})$. Hence our objective is for any x to minimize $H(p_{Y|X=x}, p_{Y|X=x,\theta})$. Recall that cross entropy has the property

$$H(p, q) = H(p) + D(p||q), \tag{24}$$

hence we minimize in the objective, for any x the divergence $D(p_{Y|X=x}||p_{Y|X=x,\theta})$ where $p_{Y|X}(y|x)$ is the true conditional pmf and $p_{Y|X,\theta}(y|x, \theta)$ is the one given by the model.

After finding the expression for the log likelihood function, we can use any of our optimization methods we discussed earlier. Let's implement the batch gradient ascent method to maximize the log likelihood function of the Binary case Eq. (21). To do so, we will need to find $\nabla_\theta l(\theta)$.

$$\begin{aligned}
\frac{\partial l(\theta)}{\partial \theta_j} &= \sum_{i=1}^n \frac{\partial}{\partial \theta_j} y^{(i)} \log(\sigma\left(\theta^T x^{(i)}\right)) + (1 - y^{(i)}) \log(1 - \sigma\left(\theta^T x^{(i)}\right)) \\
&\overset{(a)}{=} \sum_{i=1}^n y^{(i)} \frac{\sigma'\left(\theta^T x^{(i)}\right)}{\sigma\left(\theta^T x^{(i)}\right)} x_j^{(i)} + (1 - y^{(i)}) \frac{-\sigma'\left(\theta^T x^{(i)}\right)}{1 - \sigma\left(\theta^T x^{(i)}\right)} x_j^{(i)} \\
&\overset{(b)}{=} \sum_{i=1}^n y^{(i)} \left(1 - \sigma\left(\theta^T x^{(i)}\right)\right) x_j^{(i)} - (1 - y^{(i)}) \sigma\left(\theta^T x^{(i)}\right) x_j^{(i)} \\
&\overset{(c)}{=} \sum_{i=1}^n \left(y^{(i)} - \sigma\left(\theta^T x^{(i)}\right)\right) x_j^{(i)}
\end{aligned} \tag{25}$$

Where in (a) we took derivatives, in (b) we used property (19), and in (c) we simplified the expression. After finding the gradient, we can write the logistic regression update rule

$$\theta_j^{(t+1)} := \theta_j^{(t)} + \alpha \sum_{i=1}^n \left(y^{(i)} - \sigma\left(\theta^T x^{(i)}\right)\right) x_j^{(i)} . \tag{26}$$

Next we will talk about how to generalize the logistic regression algorithm to a non-binary classifier.

VI. Additional way of obtaining Cross Entropy as a cost function

In the first part of this lecture we talked about linear regression and we introduced the MSE cost function. The MSE cost function is widely used in applied mathematics and machine learning. Using different cost functions will end up giving us different results when using our optimization methods such as Gradient Descent for our regression and classification problems.

In the second part of this lecture we introduced Logistic Regression and showed that maximizing the log likelihood function resulted in achieving the Cross Entropy cost function. The Cross Entropy cost function has an advantage over the MSE cost function in some cases. In order to understand the advantage we shall first address a problem.

Assume we're solving a Logistic regression problem, while using the logistic function $\sigma(\theta^T x^{(i)})$ and choosing our cost function to be the MSE cost function:

$$\begin{aligned} C(\theta) &= \frac{1}{2n}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2 \\ &= \frac{1}{2n}\sum_{i=1}^{n}(\sigma(\theta^T x^{(i)}) - y^{(i)})^2 \end{aligned} \tag{27}$$

Now derive $C(\theta)$ in order to use Gradient Descent:

$$\frac{\partial C(\theta)}{\partial \theta_j} = \frac{1}{n}\sum_{i=1}^{n}(\sigma(\theta^T x^{(i)}) - y^{(i)})x_j^{(i)}\sigma'(\theta^T x^{(i)}) \tag{28}$$

Updating, we will achieve:

$$\theta_j^{(t+1)} := \theta_j^{(t)} - \frac{\alpha}{n}\sum_{i=1}^{n}(\sigma(\theta^T x^{(i)}) - y^{(i)})x_j^{(i)}\sigma'(\theta^T x^{(i)}),\ \forall j \tag{29}$$

Notice that for large or small values of $\theta^T x^{(i)}$ the term $\sigma'(\theta^T x^{(i)})$ is very small (see Fig. 10), causing what is called “Learning Slowdown”. This is a problem, because our learning process can take many iterations to compute due to the slow update of θ_j (the exact meaning of the term ’learning process’ will be clarified more in the next lecture).

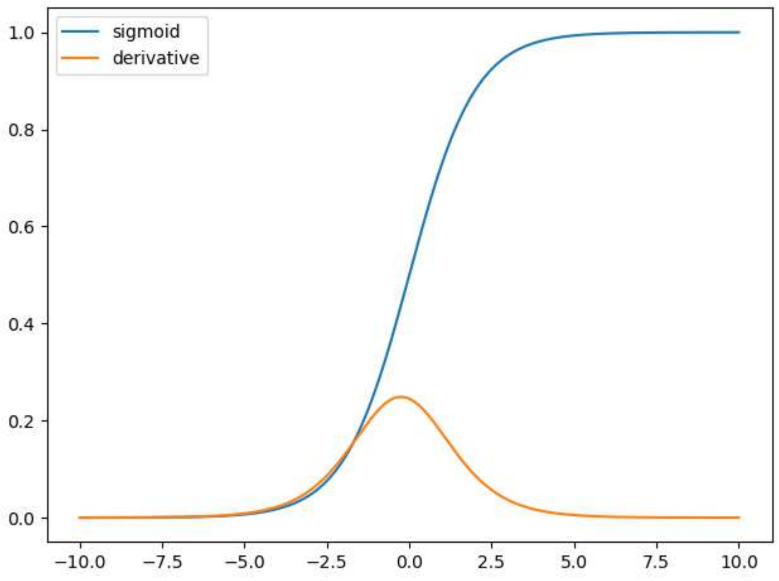

Fig. 10: Sigmoid, $\sigma(x)$, and its derivative, $\sigma'(x)$.

Now let's try a different approach. What if we could choose a cost function so that the term $\sigma'(\theta^T x^{(i)})$ disappeared? In that case, the cost for a single training example X would be:

$$\frac{\partial C(\theta)}{\partial \theta_j} = (\sigma(\theta^T x) - y)x_j. \tag{30}$$

For simplicity, define $z = \theta^T x$. Now derive our cost function:

$$\frac{\partial C(\theta)}{\partial \theta_j} = \frac{\partial C(\theta)}{\partial \sigma(z)} \frac{\partial \sigma(z)}{\partial z} \frac{\partial z}{\partial \theta_j} = \frac{\partial C(\theta)}{\partial \sigma(z)} \sigma(z)(1 - \sigma(z))x_j = (\sigma(z) - y)x_j, \tag{31}$$

Hence,

$$\frac{\partial C(\theta)}{\partial \sigma(z)} = \frac{(\sigma(z) - y)}{\sigma(z)(1 - \sigma(z))} = \frac{-y}{\sigma(z)} + \frac{1 - y}{1 - \sigma(z)} \tag{32}$$

Integrating this expression with respect to $\sigma(z)$ will give us:

$$C_x = -[y \log(\sigma(z)) + (1 - y) \log(1 - \sigma(z))] \tag{33}$$

This is the contribution to the cost from a single example. To get the full cost function we must average over all training examples, obtaining:

$$C = -\frac{1}{n}\sum_{x}(y\log(\sigma(z)) + (1-y)\log(1-\sigma(z)). \tag{34}$$

We achieved the Cross-Entropy cost function.

To summarize, we showed in this section that the Cross-Entropy cost function has an advantage. When using cross-entropy cost function the larger the error $(\sigma(z)-y)$, the larger the change in θ. This means that the learning process will potentially accelerate.

VII. Softmax Regression

Softmax regression is used when our target value y can take values from the discrete set $\{1, 2, \ldots, k\}$. In this case we assume that

$$\begin{aligned} p\left(y=1|x;\theta\right) =& \ \phi_1 \\ p\left(y=2|x;\theta\right) =& \ \phi_2 \\ &\vdots \\ p\left(y=k-1|x;\theta\right) =& \ \phi_{k-1} \\ p\left(y=k|x;\theta\right) =& \ 1-\sum_{i=1}^{k-1}\phi_i = \phi_k. \end{aligned} \tag{35}$$

As you might noticed, in order to estimate the parameters of a classifier of k classes, we need to estimate $k-1$ parameters and the last one is determined by the other ones (exactly like the case of a binary classifier). Due to (35) we can write the conditional probability as

$$p\left(y|x;\theta\right) = p\left(y=1|x;\theta\right)^{\mathbb{1}\{y=1\}} p\left(y=2|x;\theta\right)^{\mathbb{1}\{y=2\}} \cdots p\left(y=k|x;\theta\right)^{\mathbb{1}\{y=k\}} \tag{36}$$

where $\mathbb{1}\{\cdot\}$ is the indicator function that returns 1 if the argument statement is true and 0 otherwise. Now we will estimate $\phi_1, \ldots, \phi_{k-1}$ with the following functions

$$\hat{\phi_1} = h_{\theta^{(1)}}(x) = \frac{\exp{(\theta^{(1)})^T x}}{\sum_{i=1}^{k}\exp{(\theta^{(i)})^T x}}$$

$$\hat{\phi_2} = h_{\theta^{(2)}}(x) = \frac{\exp{(\theta^{(2)})^T x}}{\sum_{i=1}^{k}\exp{(\theta^{(i)})^T x}}$$

$$\vdots$$

$$\hat{\phi_{k-1}} = h_{\theta^{(k-1)}}(x) = \frac{\exp{(\theta^{(k-1)})^T x}}{\sum_{i=1}^{k} \exp{(\theta^{(i)})^T x}}$$

$$\hat{\phi_k} = h_{\theta^{(k)}}(x) = \frac{1}{\sum_{i=1}^{k} \exp{(\theta^{(i)})^T x}}$$

Where $\theta^{(i)}$ for all $i = 1, \ldots, k-1$, and $\theta^{(k)} = \vec{0}$ are the parameter that are used to generate the hypothesis. Let Θ be the parameters matrix that contains $\theta^{(i)}$ for all $i = 1, \ldots, k$ such as

$$\Theta = \begin{bmatrix} | & | & & | & | \\ (\theta^{(1)}) & (\theta^{(2)}) & \cdots & (\theta^{(k-1)}) & 0 \\ | & | & & | & | \end{bmatrix}. \tag{37}$$

Moreover, note that $\sum_{i=1}^{k} \hat{\phi_i} = 1,\ 0 \le \hat{\phi_i} \le 1$ so the $\phi^{(i)}$'s form a probability distribution. Next, let's use the maximum likelihood criteria on the log likelihood function. The log likelihood is given by

$$\begin{aligned}
l(\Theta) &= \log p\,(\vec{y}|X;\Theta) \\
&= \log p\left(y^{(1)}, \ldots, y^{(n)}|x^{(1)}, \ldots, x^{(n)};\Theta\right) \\
&\overset{(a)}{=} \log \prod_{i=1}^{n} p\left(y^{(i)}|x^{(i)};\Theta\right) \\
&\overset{(b}{=} \log \prod_{i=1}^{n} h_{\theta^{(1)}}(x^{(i)})^{\mathbb{1}\{y^{(i)}=1\}} h_{\theta^{(2)}}(x^{(i)})^{\mathbb{1}\{y^{(i)}=2\}} \cdots h_{\theta^{(k)}}(x^{(i)})^{\mathbb{1}\{y^{(i)}=k\}} \\
&\overset{(c)}{=} \sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 1\} \log h_{\theta^{(1)}}(x^{(i)}) + \cdots + \mathbb{1}\{y^{(i)} = k\} \log h_{\theta^{(k)}}(x^{(i)}) \\
&\overset{(d)}{=} \sum_{i=1}^{n} \sum_{q=1}^{k} \mathbb{1}\{y^{(i)} = q\} \log h_{\theta^{(q)}}(x^{(i)})
\end{aligned} \tag{38}$$

Where in (a) we used the assumption that the training set was generated independently, in (b), (c) we used the logarithm properties and in (d) we simplified the expression. Let substitute the hypothesis with its explicit function to get

$$l(\Theta) = \sum_{i=1}^{n} \sum_{q=1}^{k} \mathbb{1}\{y^{(i)} = q\} \log h_{\theta^{(q)}}(x^{(i)})$$

$$
\begin{aligned}
&= \sum_{i=1}^{n}\sum_{q=1}^{k} \mathbb{1}\{y^{(i)} = q\} \log \frac{\exp{(\theta^{(q)})^T x^{(i)}}}{\sum_{p=1}^{k} \exp{(\theta^{(p)})^T x^{(i)}}} \\
&\overset{(a)}{=} \sum_{i=1}^{n}\sum_{q=1}^{k} \mathbb{1}\{y^{(i)} = q\} \left[\log \exp{(\theta^{(q)})^T x^{(i)}} - \log \sum_{p=1}^{k} \exp{(\theta^{(p)})^T x^{(i)}} \right] \\
&\overset{(b)}{=} \sum_{i=1}^{n} \left[\sum_{q=1}^{k} \mathbb{1}\{y^{(i)} = q\} (\theta^{(q)})^T x^{(i)} - \sum_{q=1}^{k} \mathbb{1}\{y^{(i)} = q\} \log \sum_{p=1}^{k} \exp{(\theta^{(p)})^T x^{(i)}} \right] \\
&\overset{(c)}{=} \sum_{i=1}^{n} \left[\sum_{q=1}^{k} \mathbb{1}\{y^{(i)} = q\} (\theta^{(q)})^T x^{(i)} - \log \sum_{p=1}^{k} \exp{(\theta^{(p)})^T x^{(i)}} \right] \qquad (39)
\end{aligned}
$$

Where (a), (b) we used logarithm properties and in (c) we used the fact that the indicator function return 1 only once for every training example. Now, let's find the derivatives of $l(\Theta)$ with respect to $\theta_j^{(r)}$ to form our gradient ascent update rule that will maximize our log likelihood function.

$$
\begin{aligned}
\frac{\partial l(\Theta)}{\partial \theta_j^{(r)}} &= \frac{\partial}{\partial \theta_j^{(r)}} \sum_{i=1}^{n} \left[\sum_{q=1}^{k} \mathbb{1}\{y^{(i)} = q\} (\theta^{(q)})^T x^{(i)} - \log \sum_{p=1}^{k} \exp{(\theta^{(p)})^T x^{(i)}} \right] \\
&\overset{(a)}{=} \sum_{i=1}^{n} \left[\sum_{q=1}^{k} \frac{\partial}{\partial \theta_j^{(r)}} \mathbb{1}\{y^{(i)} = q\} (\theta^{(q)})^T x^{(i)} - \frac{\partial}{\partial \theta_j^{(r)}} \log \sum_{p=1}^{k} \exp{(\theta^{(p)})^T x^{(i)}} \right] \\
&\overset{(b)}{=} \sum_{i=1}^{n} \left[\mathbb{1}\{y^{(i)} = r\} x_j^{(i)} - \frac{\exp{(\theta^{(r)})^T x^{(i)}}}{\sum_{p=1}^{k} \exp{(\theta^{(p)})^T x^{(i)}}} x_j^{(i)} \right] \\
&\overset{(c)}{=} \sum_{i=1}^{n} \left[\mathbb{1}\{y^{(i)} = r\} - \frac{\exp{(\theta^{(r)})^T x^{(i)}}}{\sum_{p=1}^{k} \exp{(\theta^{(p)})^T x^{(i)}}} \right] x_j^{(i)} \\
&\overset{(d)}{=} \sum_{i=1}^{n} \left[\mathbb{1}\{y^{(i)} = r\} - p\left(y^{(i)} = r | x^{(i)}; \Theta\right) \right] x_j^{(i)} \qquad (40)
\end{aligned}
$$

Where in (a) we used the linearity of the derivative, in (b) we took derivatives, and in (c), (d) we simplified the expression. Finally we can write our batch gradient ascent update rule to be

$$
\theta_j^{(r)} := \theta_j^{(r)} + \alpha \sum_{i=1}^{n} \left[\mathbb{1}\{y^{(i)} = r\} - p\left(y^{(i)} = r | x^{(i)}; \Theta\right) \right] x_j^{(i)} \qquad (41)
$$

Note that the gradient's value goes to 0 when

$$
y^{(i)} = r, \;\; p\left(y^{(i)} = r | x^{(i)}; \Theta\right) \to 1
$$

$$y^{(i)} \neq r, \;\; p\left(y^{(i)} = r|x^{(i)}; \Theta\right) \rightarrow 0 \;\; , \;\; \forall i, r$$

so the gradient stops adjusting Θ when $p\left(y^{(i)} = r|x^{(i)}; \Theta\right)$ is close to 1 for the correct class and close to 0 for the incorrect classes, which approves that our results makes sense.

Appendix

The following optimization method is called Newton-Raphson and it trys to find oint where the derivative is 0 or cery close to 0.

Definition 3 (Newton-Raphson Method)

For some $f : \mathbb{R} \rightarrow \mathbb{R}$ differentiable over $\mathbb{R}$, its zeros can be found by an iterative process with the following update rule.

1) Set θ_0 as the initial guess for θ^*, such as $f(\theta^*) = 0$.
2) Repeat the following update rule until $d(\theta^{(t+1)}, 0) < \delta$, where δ is defined to be the convergence tolerance, $d(\cdot)$ is defined to be a distance norm.

$$\theta^{(t+1)} := \theta^{(t)} - \frac{f(\theta^{(t)})}{f'(\theta^{(t)})} \tag{42}$$

where:

- $t = 0, 1, \ldots$ is the iteration number.
- $f'(\theta)$ is the first derivative of $f(\theta)$.

First, to get some intuition about the algorithm's process, let's examine the private case where $f(x) = \frac{1}{3}x^3$. In our case the update rule is given by

$$x^{(t+1)} := x^{(t)} - \frac{1}{3}x^{(t)}. \tag{43}$$

The following figures presents the algorithm process in the first iterations.

In the leftmost figure, we can see the initial guess for x, denoted by x_0. In the middle figure, we can see how the algorithm, calculate its next guess of x. First, it calculates the tangent of $f(x)$ at x_0, and then sets the next guess at the point where the tangent cross the x axis. We can think of that as estimating the zeros of $f(x)$ as if $f(x)$ was a linear function. From this point of view let's find the derive the update rule.

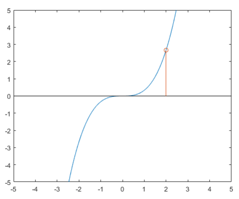
Fig. 11: initial guess of Newton-Raphson method

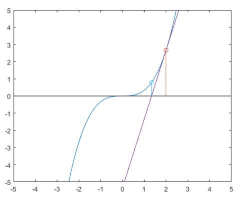
Fig. 12: 1 iteration of Newton-Raphson method

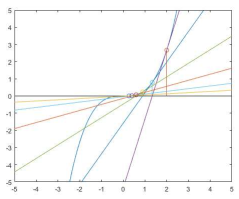
Fig. 13: 5 iterations of Newton-Raphson method

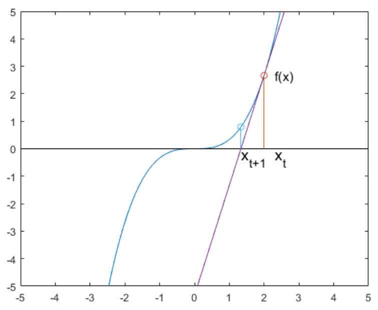

Fig. 14: Newton-Raphson method visualization

The tangent's slope is the derivative of $f(x)$ at x_t so we can conclude that

$$f'(x^{(t)}) = \frac{f(x^{(t)}) - 0}{x^{(t)} - x^{(t+1)}}. \tag{44}$$

By simplifying the equation above we get

$$x^{(t+1)} := x^{(t)} - \frac{f(x^{(t)})}{f'(x^{(t)})}, \tag{45}$$

which is the update rule noted in the algorithm's definition. Now, When applying Newton-Raphson's (N-R's) method in minimizing the cost function, We will choose

$$x \triangleq \theta$$

$$f(\cdot) \triangleq C'(\cdot)$$

$$f'(\cdot) \triangleq C''(\cdot).$$

Therefore, the update rule for minimizing the cost function will be

$$\theta^{(t+1)} := \theta^{(t)} - \frac{C'(\theta^{(t)})}{C''(\theta^{(t)})}. \tag{46}$$

In case that θ is a vector we can generalize the update rule to be

$$\theta^{(t+1)} := \theta^{(t)} - H_\theta^{-1} \nabla_\theta C(\theta^{(t)}), \tag{47}$$

where

- H_θ is the *Hessian* matrix of C with respect to θ, whose entries are given by
$$H_{i,j} = \frac{\partial^2 C(\theta)}{\partial \theta_i \partial \theta_j}$$
- $\nabla_\theta C(\theta^{(t)})$ is the gradient of C with respect to θ.

Note that for a quadratic cost function, its gradient will be a linear function of θ and therefore the algorithm will converge after one iteration (verify that you understand why it is true). More generally, N-R method converges faster that gradient descent (this will not be proven here), but it is more computationally expensive. That is because every update rule the algorithm needs to find and invert ($\sim m^3$ operations) the Hessian matrix.

References

[1] Andrew NG's machine learning course. Lecture on *Supervised Learning* http://cs229.stanford.edu/materials.html.