CS221 Lecture notes #5

Supervised learning

So far in this course, we have only considered problems where the entire state of the world is known in advance. Rarely, however, can we completely specify the state of the world a priori, so often the agent must be able to learn about the world from actual observations. For instance, we might be interested in automatically distinguishing different handwritten digits. It's hard for us to formally state a set of rules which distinguish handwritten 2's from 5's, and so we can't simply program it directly into the computer. Rather, we will have an algorithm automatically figure it out from data. The general set of techniques for doing this is known as **machine learning**. Since machine learning is such a broad field, there is no single definition that everybody agrees upon, but here are some attempts:

- Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed.[1]
- A computer program is said to learn from experience E with respect to some task T and some performance measure P if its performance on T, as measured by P, improves with experience E.[2]

An example of an early machine learning program was Arthur Samuel's chess playing program, which learned to play checkers by playing many games against itself, and eventually learned to play much better than Samuel himself. Since them, machine learning has produced many practical applications.

For the next two lectures, we will focus on **supervised learning**, where our algorithm will work with *labeled* training examples, or examples which

[1] Arthur, S. "Some studies in machine learning using the game of checkers." *IBM Journal* (3): 210-229.

[2] Mitchell, T. *Machine Learning.* McGraw-Hill, 1997.

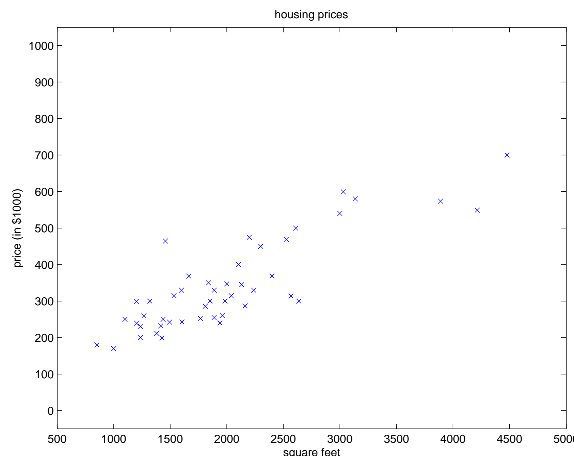
are labeled with the property we are trying to predict. For instance, if we are trying to classify handwritten digits, labeled data would constitute image files which are labeled by humans as being a particular digit.

1 Linear regression

As a motivating example, suppose we have a dataset giving the living areas and prices of 47 houses from Portland, Oregon:

Living area (feet^2)	Price (1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮

We can plot this data:

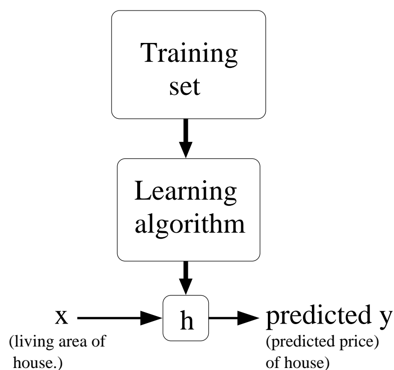


Given data like this, how can we learn to predict the prices of other houses in Portland, as a function of the size of their living areas?

To establish notation for future use, we'll use $x^{(i)}$ to denote the "input" variables (living area in this example), also called input **features**, and $y^{(i)}$ to denote the "output" or **target** variable that we are trying to predict (price). A pair $(x^{(i)}, y^{(i)})$ is called a **training example**, and the dataset that we'll be using to learn—a list of m training examples $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$—is called a **training set**. Note that the superscript "(i)" in the

notation is simply an index into the training set, and has nothing to do with exponentiation. We will also use $\mathcal{X}$ denote the space of input values, and $\mathcal{Y}$ the space of output values. In this example, $\mathcal{X} = \mathcal{Y} = \mathbb{R}$.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function $h : \mathcal{X} \mapsto \mathcal{Y}$ so that $h(x)$ is a "good" predictor for the corresponding value of y. For historical reasons, this function h is called a **hypothesis**. Seen pictorially, the process is therefore like this:



When the target variable that we're trying to predict is continuous, such as in our housing example, we call the learning problem a **regression** problem. When y can take on only a small number of discrete values (such as if, given the living area, we wanted to predict if a dwelling is a house or an apartment, say), we call it a **classification** problem.

To perform supervised learning, we must decide how we're going to represent functions/hypotheses h in a computer. In **linear regression**, our hypotheses are linear functions of the features. In the case of housing prices, we have one feature x representing the area, and so our hypothesis is

$$h_\theta(x) = \theta_0 + \theta_1 x.$$

(We will drop the subscript θ when there is no risk of confusion.) We might also happen to know the number of bedrooms as well. In this case, we have two features x_1 and x_2, so our hypothesis is

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2.$$

For the purposes of this lecture, our hypothesis will be a linear combination of n features, plus the intercept θ_0. For notational convenience, we introduce the convention of letting $x_0 = 1$. Then, we can rewrite our hypothesis as:

$$h(x) = \sum_{i=0}^{n} \theta_i x_i.$$

Equivalently, we can rewrite this using an inner product:

$$h(x) = \theta^T x,$$

where on the right-hand side we are viewing θ and x both as vectors in $\mathbb{R}^{n+1}$, and n is the number of input variables (not counting x_0). We refer to θ as the **parameters**, or **weights**, of the hypothesis. θ_0 is also referred to as the **intercept term**.

Now, given a training set, how do we pick, or learn, the parameters θ? One reasonable method seems to be to make $h(x)$ close to y, at least for the training examples we have. To formalize this, 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. We define the **cost function**:

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

In other words, we penalize the squared magnitude of the **error term** $(h_\theta(x^{(i)}) - y^{(i)})^2$ for each training example. This algorithm is commonly referred to as **ordinary least-squares**.

2 Gradient descent

We want to choose θ so as to minimize $J(\theta)$. This is an optimization search problem, since we are only interested in finding a good value of θ, and we don't care about the path we take to find it. This suggests using greedy hill-climbing search. In our previous formulation, we applied a fixed set of operators to generate all of the successor states, and then chose the successor which minimized the cost function. This formulation is only appropriate for discrete spaces, but fortunately, there is a continuous analog called **gradient descent**. The idea behind gradient descent is to take a series of small steps in the direction of steepest descent. This direction is given by the negative **gradient** of $J(\theta)$, where each component is the corresponding partial derivative of J.

In other words, we start with an arbitrary "initial guess" for θ, and then apply the following update rules:[3]

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

[3]We use := to denote assignment and = to denote mathematical equality. For instance, $a := b$ is a computer operation assigning a variable a the value b, and $a = b$ is making the statement that a and b have the same value.

(This update is simultaneously performed for all values of $j = 0, \ldots, n$.) Here, α is called the **learning rate**.[4]

In order to implement this algorithm, we have to work out what is the partial derivative term on the right hand side. We have:

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \\
&= \frac{1}{2} \sum_{i=1}^{m} \frac{\partial}{\partial \theta_j} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \\
&= 2 \cdot \frac{1}{2} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot \frac{\partial}{\partial \theta_j} (h_\theta(x^{(i)}) - y^{(i)}) \\
&= \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{k=0}^{n} \theta_k x_k^{(i)} - y^{(i)} \right) \\
&= \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y\right) x_j^{(i)}
\end{aligned}$$

This gives the update rule:

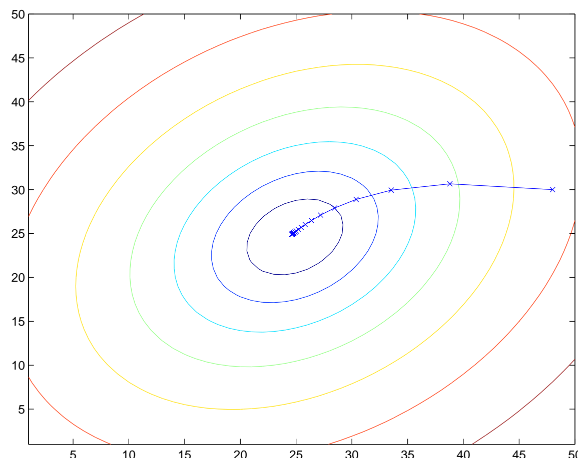
$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}.$$

The rule is called the **LMS** update rule (LMS stands for "least mean squares"). This rule has several properties that seem natural and intuitive. For instance, the magnitude of the update is proportional to the error term $(y^{(i)} - h_\theta(x^{(i)}))$; thus, for instance, if we are encountering a training example on which our prediction nearly matches the actual value of $y^{(i)}$, then we find that there is little need to change the parameters; in contrast, a larger change to the parameters will be made if our prediction $h_\theta(x^{(i)})$ has a large error (i.e., if it is very far from $y^{(i)}$).

The update rule we just described looks at every example in the entire training set on every step, and is called **batch gradient descent**. Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges

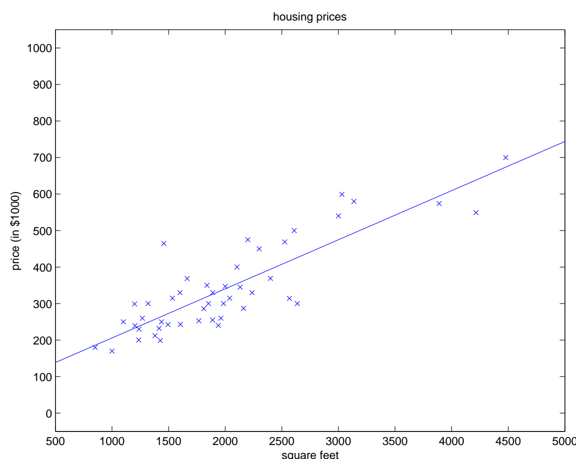
[4]The learning rate is a parameter of the algorithm which must be set by hand. Choosing the wrong value can lead to poor performance. Specifically, large values can cause gradient descent not to converge, while small values can cause it to converge too slowly.

(assuming the learning rate α is not too large) to the global minimum.[5] Here is an example of gradient descent as it is run to minimize a quadratic function.



The ellipses shown above are the contours of a quadratic function. Also shown is the trajectory taken by gradient descent, with was initialized at (48,30). The x's in the figure (joined by straight lines) mark the successive values of θ that gradient descent went through.

When we run batch gradient descent to fit θ on our previous dataset, to learn to predict housing price as a function of living area, we obtain $\theta_0 = 71.27$, $\theta_1 = 0.1345$. If we plot $h_\theta(x)$ as a function of x (area), along with the training data, we obtain the following figure:



[5]More formally, it turns out that our least-squares cost function is **convex**, which implies it has no local optima.

If the number of bedrooms were included as one of the input features as well, we get $\theta_0 = 89.60, \theta_1 = 0.1392$, $\theta_2 = -8.738$.

The above results were obtained with batch gradient descent. There is an alternative to batch gradient descent that also works very well. Consider the following algorithm:

Loop {

for i=1 to m, {

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)} \qquad \text{(for every } j\text{)}.$$

}

}

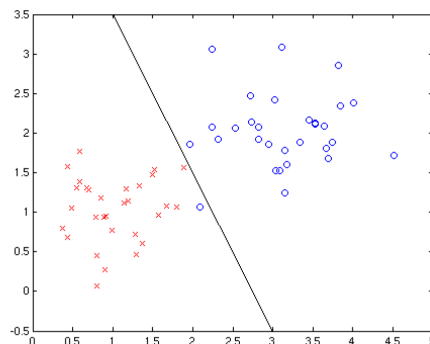
(We arrive at the update rule by taking the partial derivatives of $\frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$ with respect to θ_j.) In this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that single training example only. This algorithm is called **stochastic gradient descent** (also **incremental gradient descent**). Whereas batch gradient descent has to scan through the entire training set before taking a single step—a costly operation if m is large—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 $J(\theta)$; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum.[6]) For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

3 Classification and logistic regression

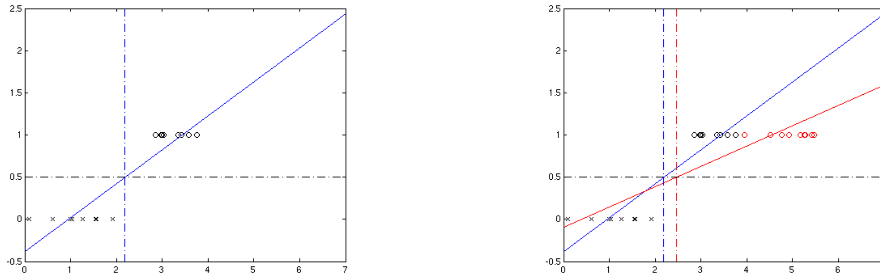
Lets now talk about classification. This is just like the regression problem, except that the values y we want to predict now take on only a small number of discrete values. For now, we will focus on the **binary classification**

[6]While it is more common to run stochastic gradient descent as we have described it and with a fixed learning rate α, by slowly letting the learning rate α decrease to zero as the algorithm runs, it is also possible to ensure that the parameters will converge to the global minimum rather then merely oscillate around the minimum.

problem in which y can take on only two values, 0 and 1. (Most of what we say here will also generalize to the multiple-class case.) For instance, if we are trying to build a spam classifier for email, then $x^{(i)}$ may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. 0 is also called the **negative class**, and 1 the **positive class**, and they are sometimes also denoted by the symbols "-" and "+." Given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the **label** for the training example. As before, we will focus on linear models, so our goal is to find a linear function which can distinguish one class from the other, such as the one shown below.



We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y given x. However, it is easy to construct examples where this method performs very poorly. Furthermore, linear regression would lead to the counter-intuitive notion of predictions less than 0 or greater than 1. Linear regression also leads to the strange effect shown below. This figure represents a decision problem with a single feature, where we use linear regression to predict the 0 or 1 value of y given that one feature. The solid blue line shows the linear regression fit, and the dashed blue line shows where the decision boundary would lie if we use the arbitrary cutoff of 0.5. On the right, we add data points far to the right of the decision boundary. Counter-intuitively, this causes the estimated decision boundary to shift to the right.



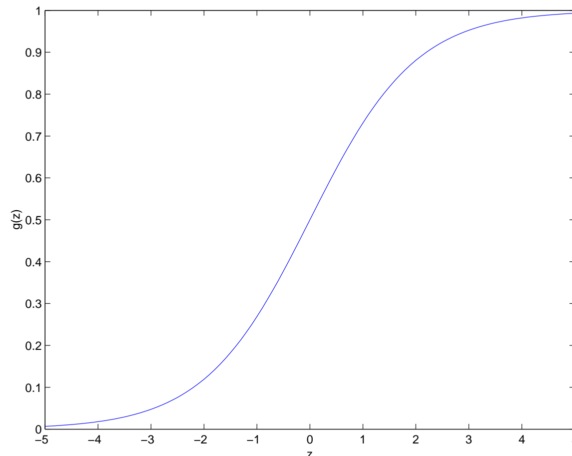
To fix this, lets change the form for our hypotheses $h_\theta(x)$. We will choose

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

where

$$g(z) = \frac{1}{1 + e^{-z}}$$

is called the **logistic function** or the **sigmoid function**. Here is a plot showing $g(z)$:



Notice that $g(z)$ tends towards 1 as $z \to \infty$, and $g(z)$ tends towards 0 as $z \to -\infty$. Moreover, g(z), and hence also $h(x)$, is always bounded between 0 and 1. As before, we are keeping the convention of letting $x_0 = 1$, so that $\theta^T x = \theta_0 + \sum_{j=1}^{n} \theta_j x_j$.

Our objective function (which we are trying to maximize, rather than minimize) is the following:

$$\ell(\theta) = \sum_{i=1}^{m} y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)}))$$

As we show in the next section, this objective function can be maximized using the same batch gradient descent rule as before:

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} (y^{(i)} - h_\theta(x^{(i)})) x_j^{(i)}.$$

4 Maximum likelihood

So far, we have presented recipes for the cost functions in linear regression and logistic regression. Now we show the justification behind these formulas. These two algorithms are both special cases of a general parameter estimation method called **maximum likelihood**. The intuition behind maximum likelihood is that we want to choose the hypothesis which makes the data as probable as possible. What do we mean by probable? To answer this question, we will have to define probabilistic models of how our data are generated.

As a motivating example, suppose we are given a (possibly biased) coin, and we want to determine the probability that this coin will come up heads. A coin toss can be modeled as a Bernoulli random variable, i.e. one which takes the value 1 with probability ϕ and 0 with probability $1 - \phi$. Suppose now that we toss this coin m times. This will give us m **independently and identically distributed (IID)** samples from this Bernoulli random variable. By IID, we mean that each sample is drawn from the same distribution, and each sample is independent of all of the others.

Let h represent the number of heads in the m tosses. You can check that the probability of seeing a particular sequence of exactly h heads and $m - h$ tails is:

$$\phi^h (1 - \phi)^{m-h}.$$

We call this probability the **likelihood**; it is the probability of the observed data assuming the model parameters. We often denote this probability as $p(\text{data}; \phi)$.[7]

In maximum likelihood, we choose our parameters to maximize the likelihood. Here, this gives us

$$\phi = \arg\max_{\phi} \; P(\text{data}; \phi) = \arg\max_{\phi} \phi^h (1 - \phi)^{m-h}.$$

Sometimes we can compute the maximum analytically by setting the partial derivatives equal to zero, and when this is not possible, we use an iterative

[7] Note that, in this class, we do *not* write $p(\text{data}|\phi)$, because we are not treating our parameter ϕ as a random variable, and therefore it makes no sense to condition on it.

procedure such as gradient descent. However, we don't usually apply either of these techniques to the likelihood itself, since it's awkward to take derivatives of products. Rather, we usually maximize the **log likelihood** $\log p(\text{data}; \phi)$, and this will always give us the same answer because the logarithm is a monotonically increasing function. In the case of coin flips,

$$\phi = \arg\max_{\phi} \log p(\text{data}; \phi) = \arg\max_{\phi} \; h \log \phi + (m-h)\log(1-\phi).$$

We can find the maximum likelihood solution by setting $\frac{\partial \log p(\text{data};\phi)}{\partial \phi}$ to zero:

$$\begin{aligned}
\frac{\partial}{\partial \phi} \log p(\text{data}; \phi) &= \frac{\partial}{\partial \phi} \left(h \log \phi + (m-h)\log(1-\phi) \right) \\
&= \frac{h}{\phi} - \frac{m-h}{1-\phi} \\
\frac{\phi}{1-\phi} &= \frac{h}{m-h} \\
\phi &= \frac{h}{m}.
\end{aligned}$$

Hence, the maximum likelihood estimate of ϕ turns out to be what we would expect: $\phi = \frac{h}{m}$, the proportion of flips which came up heads.

Before we move on, it's worth noting what maximum likelihood does *not* give us. It does not give us the probability of a given value of ϕ being the correct parameter. For instance, if you flip the coin once and it comes up heads, the likelihood for $\phi = 1$ would be 1, and the likelihood for $\phi = 0$ would be 0. But this doesn't mean we know that $\phi = 1$; in fact, we cannot make any statements about the probability of the parameters. (There is a branch of statistics called Bayesian statistics which does make such statements, but it is beyond the scope of this lecture.)

4.1 Linear regression

Now we'll show that linear regression and logistic regression are both special cases of maximum likelihood. For linear regression, we assume we have a fixed set of inputs $x^{(1)}, \ldots, x^{(m)}$. For each input $x^{(i)}$, we assume that $y^{(i)}$ is a Gaussian random variable whose mean is a linear function of $x^{(i)}$, and whose variance is fixed at σ^2. Mathematically,

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

Taking the log as we usually do,

$$\log p(y|x;\theta) = -\frac{1}{2}\log(2\pi\sigma^2) - \frac{(y-\theta^T x)^2}{2\sigma^2}.$$

When summed over all of our training examples,

$$\begin{aligned}
\log p(y^{(1)},\ldots,y^{(m)}|x^{(1)},\ldots,x^{(m)};\theta) &= \log\prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta) \\
&= \sum_{i=1}^{m}\log p(y^{(i)}|x^{(i)};\theta) \\
&= -\frac{m}{2}\log(2\pi\sigma^2) - \sum_{i=1}^{m}\frac{(y-\theta^T x)^2}{2\sigma^2}
\end{aligned}$$

The $-\frac{m}{2}\log(2\pi\sigma^2)$ term doesn't depend on θ, and so maximizing this is equivalent to minimizing our least-squares objective function from before, so linear regression is a special case of maximum likelihood where the observations are assumed to be Gaussian. Note that we are treating the $x^{(i)}$'s as fixed, and the $y^{(i)}$'s as random variables.

4.2 Logistic regression

The derivation for logistic regression is similar. As in linear regression, we assume a fixed set of inputs $x^{(i)}$, and the targets $y^{(i)}$'s are treated as random variables. This time, the $y^{(i)}$'s are Bernoulli random variables whose parameter ϕ depends on x. In particular, we model ϕ as a logistic function of $\theta^T x^{(i)}$. Mathematically, we have

$$\begin{aligned}
p(y=1|x,\theta) &= h_\theta(x) \\
&= g(\theta^T x) \\
&= \frac{1}{1+\exp(-\theta^T x)} \\
\log p(y|x,\theta) &= y\log h_\theta(x) + (1-y)\log(1-h_\theta(x))
\end{aligned}$$

The log likelihood, therefore, is

$$\begin{aligned}
\log p(y^{(1)},\ldots,y^{(m)}|x^{(1)},\ldots,x^{(m)};\theta) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta) \\
&= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)};\theta) \\
&= \sum_{i=1}^{m} y \log h_\theta(x) + (1-y)\log(1-h_\theta(x))
\end{aligned}$$

We have arrived at the objective function we presented earlier for logistic regression.

Now we take the partial derivatives of the likelihood with respect to θ in order to derive the gradient descent update rules. Before moving on, here's a useful property of the derivative of the sigmoid function $g(z)$, which we write a g':

$$\begin{aligned}
g'(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\
&= \frac{1}{(1+e^{-z})^2}\left(e^{-z}\right) \\
&= \frac{1}{(1+e^{-z})}\cdot\left(1-\frac{1}{(1+e^{-z})}\right) \\
&= g(z)(1-g(z)).
\end{aligned}$$

Written in vectorial notation, our updates will be given by $\theta := \theta + \alpha\nabla_\theta \ell(\theta)$. (Note the positive rather than negative sign in the update formula, since we're maximizing, rather than minimizing, a function now.) Lets start by working with just one training example (x, y), and take derivatives to derive the stochastic gradient ascent rule:

$$\begin{aligned}
\frac{\partial}{\partial\theta_j}\ell(\theta) &= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right)\frac{\partial}{\partial\theta_j}g(\theta^T x) \\
&= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right)g(\theta^T x)(1-g(\theta^T x)\frac{\partial}{\partial\theta_j}\theta^T x \\
&= \left(y(1-g(\theta^T x)) - (1-y)g(\theta^T x)\right)x_j \\
&= \left(y-h_\theta(x)\right)x_j
\end{aligned}$$

Above, we used the fact that $g'(z) = g(z)(1-g(z))$. This therefore gives us the stochastic gradient ascent rule

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

The batch gradient descent rule will simply use the partial derivative summed over all of the training examples:

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}$$

Thus, logistic regression is simply the special case of maximum likelihood where the target variables are Bernoulli(ϕ), and ϕ is a sigmoidal function of x. As it turns out, maximum likelihood is a widely applicable formalism, and we will see one more example when we discuss decision tree learning in the next lecture.