‘cs229’—Notes

Malcolm

Started 1st October 2024

Contents

Chapter 1

Supervised learning

Given a dataset of n *training examples* $\{(x^{(i)}, y^{(i)}); i = 1\}, \dots, n\}$—a *training set*—where $\boldsymbol{x}$ represents the *features* and $\boldsymbol{y}$ the "output" or *target* variable we are trying to predict. If not already obvious, we denote the vector space of $\boldsymbol{x}$ as $\mathcal{X}$ and that of the outputs $\boldsymbol{y}$ as $\mathcal{Y}$.

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 y. This function h is called a *hypothesis*.

When trying to predict a continuous target variable, we call this a *regression* problem; whereas when y can take on only a small number of discrete values we call that a *classification* problem.

1.1 Linear Regression

Say we decide to approximate y as a linear function of x:

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

Where $\boldsymbol{\theta}$ represents the *parameters/weights* (parametrising the space of linear functions mapping from $\mathcal{X}$ to $\mathcal{Y}$). We can simplify our notation as such: (by convention letting $x_0 = 1$, aptly named the *intercept* term)

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

In order to formalise a measure of proximity between the predicted value $h(x)$ and the target y, we define a *cost function*:

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

This particular cost function implies an *ordinary least squares* regression model.

1.1.1 LMS algorithm

Our cost function $J(\theta)$ gives us a measure of prediction accuracy. We want to choose θ so as to minimise $J(\theta)$. Starting with an initial set of θ, we need a search algorithm that repeatedly changes θ in an attempt to minimise $J(\theta)$. Here we consider the *gradient descent* algorithm, which, given some initial θ, repeatedly performs the update:

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

Where the update is simultaneously performed for all values of $j = 0, \ldots, d$. α is called the *learning rate* (how much we move in the direction the gradient points in).

Intuition
Consider attempting to minimise the least mean squares (LMS) cost function for a single training example:

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

This gives us the update rule:

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

(we use the notation $a := b$ to denote (in a script) overwriting a with b) Notice the property of the LMS update rule that the magnitude of the update is proportional to the *error* term $\left(y^{(i)} - h_\theta(x^{(i)})\right)$; this means that predictions further off the mark result in a greater correction to θ.
(next page)

Batch Gradient Descent
We had the LMS rule for when there was only a single training example. One way to modify this method for a training set of more than one example is the following algorithm:

```
Repeat until convergence {
        $\theta_j := \theta_j + \alpha \sum_{i=1}^{n} \left( y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)},$ (for every $j$)
}
```

Written more succinctly (θ_j and x_j as vectors):

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

This method looks at every example in the entire training set on every step, and is called *batch gradient descent*.

Stochastic gradient descent
Now consider another algorithm:

```
Loop {
    for $i = 1$ to $n$, {
            $\theta_j := \theta_j + \alpha \left( y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)},$    (for every $j$)
    }
}
```

Written more compactly:

$$\theta := \theta + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x^{(i)}, \text{ (update } \theta \text{ } n \text{ times)}$$

Here we update θ for each training example during each run of the training set. This is called *stochastic/incremental gradient descent.*

Whereas batch gradient descent has to scan through the entire training set before taking a single step—which is costly if n is large—stochastic gradient descent continues to make progress with each example it looks at. Stochastic gradient descent gets θ “close” to the minimum much faster than batch gradient descent. Note that “convergence” doesn’t really occur—the parameters θ will keep oscillating around the minimum of $J(\theta)$. (though most values near minimum would be reasonably good approximations to the true minimum)

1.1.2 Gradient/Hessian of b^Tx, x^TAx

Matrix Derivatives
For a function $f : \mathbb{R}^{n\times d} \mapsto \mathbb{R}$ mapping from n-by-d matrices to real numbers, we define the derivative of f with respect to A to be

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1d}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{n1}} & \cdots & \frac{\partial f}{\partial A_{nd}} \end{bmatrix}$$

Gradient of b^Tx:
For $x \in \mathbb{R}^n$ and $f(x) = b^Tx$ for known $b \in \mathbb{R}^n$, we have

$$f(x) = \sum_{i=1}^n b_i x_i$$

and so

$$\frac{\partial f(x)}{\partial x_k} = \frac{\partial}{\partial x_k} \sum_{i=1}^n b_i x_i = b_k$$

Consider repeating this for the partial of each element of x. See that $\nabla_x b^T x = b$ (analagous to single variable calculus).

Gradient of $f(x) = x^TAx$:
Now consider the quadratic function $f(x) = x^TAx$ for $A \in \mathbb{S}^n$ (meaning symmetric). First see that

$$f(x) = \sum_{i=1}^n \sum_{j=1}^n A_{ij} x_i x_j$$

this can be seen from considering that

$$b = Ax \in \mathbb{R}^{n\times 1}$$

can be written as

$$\begin{aligned} b_1 &= \sum_{i=1}^n A_{1i} x_i \\ &\vdots \\ b_n &= \sum_{i=1}^n A_{ni} x_i \end{aligned}$$

(next page)

so we can write

$$\begin{aligned} x^T A x = x^T b &= \sum_{j=1}^{n} x_j b_j \in \mathbb{R} \\ &= \sum_{j=1}^{n} x_j \left(\sum_{i=1}^{n} A_{ji} x_i \right) \end{aligned}$$

Rewriting gives us

$$f(x) = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j$$

which was what we wanted. Now we take the partial derivative by considering the terms including x_k an x_k^2 factors separately:

$$\begin{aligned} \frac{\partial f(x)}{\partial x_k} &= \frac{\partial f(x)}{\partial x_k} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j \\ &= \frac{\partial f(x)}{\partial x_k} \left[\sum_{i \neq k} \sum_{j \neq k} A_{ij} x_i x_j + \sum_{i \neq k} A_{ik} x_i x_k + \sum_{j \neq k} A_{kj} x_k x_j + A_{kk} x_k^2 \right] \\ &= \sum_{i \neq k} A_{ik} x_i + \sum_{j \neq k} A_{kj} x_j + 2 A_{kk} x_k \\ &= \sum_{i=1} A_{ik} x_i + \sum_{j=1} A_{kj} x_j = 2 \sum_{i=1}^{n} A_{ki} x_i \end{aligned}$$

where the last equality follows since A is assumed to be *symmetric*. Since

$$\sum_{i=1}^{n} A_{ki} x_i$$

is just the inner product of a single row of A and x—the kth entry of $\nabla_x f(x)$ is the just the inner product of the kth row of A and x; therefore

$$\nabla_x x^T A x = 2Ax$$

also analagous to single variable calculus.
(next page)

Hessian
Finally we consider the Hessian of the quadratic function $f(x) = x^T Ax$ (it should be obvious that the Hessian of a linear function $b^T x$ is zero):

$$\frac{\partial^2 f(x)}{\partial x_k \partial x_\ell} = \frac{\partial}{\partial x_k}\left[\frac{\partial f(x)}{\partial x_\ell}\right] = \frac{\partial}{\partial x_k}\left[2\sum_{i=1}^{n} A_{\ell i} x_i\right] = 2A_{\ell k} = 2A_{k\ell}$$

See therefore that $\nabla_x^2 x^T Ax = 2A$.

Recapitulation:

- $\nabla_x b^T x = b$
- $\nabla_x x^T Ax = 2Ax$
- $\nabla_x^2 x^T Ax = 2A$

1.1.3 Least Squares matrix representation

Here we consider $J(\theta)$—the least squares cost function—in matrix-vectorial notation.

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

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

and y the n-dimensional vector containing the target values:

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

Now we try to represent the least squares function over the entire dataset; see that since

$$h_\theta(x^{(i)}) = (x^{(i)})^T\theta$$

we can write

$$\begin{aligned} X\theta - y &= \begin{bmatrix} (x^{(1)})^T\theta \\ \vdots \\ (x^{(n)})^T\theta \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix} \\ &= \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ \vdots \\ h_\theta(x^{(n)}) - y^{(n)} \end{bmatrix} \in \mathbb{R}^n \end{aligned}$$

Using the fact that for a vector z, $z^Tz = \sum_i z_i^2$:

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

(next page)

Minimising the LMS
To minimise the cost function $J(\theta)$ we want its derivatives with respect to θ:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}(X\theta - y)^T(X\theta - y) \\
&= \frac{1}{2}\nabla_\theta \left((X\theta)^T - y^T\right)(X\theta - y) \\
&= \frac{1}{2}\nabla_\theta(\theta^T X^T X\theta - y^T X\theta - (X\theta)^T y - y^T y) \\
&= \frac{1}{2}\nabla_\theta(\theta^T X^T X\theta - y^T X\theta - y^T X\theta) \\
&= \frac{1}{2}\nabla_\theta(\theta^T X^T X\theta - 2(X^T y)\theta) \\
&= \frac{1}{2}(2X^T X\theta - 2X^T y) \\
&= X^T X\theta - X^T y
\end{aligned}$$

Of note:

- The third equality comes from $(Ab)^T = b^T A^T$
- The fourth equality comes from $a^T b = b^T a$
- To differentiate we use the facts $\nabla_x b^T x = b$ and $\nabla_x x^T Ax = 2Ax$. Note the second fact assumes A is symmetric, which is true here since $X^T X$ is symmetric.

To minimise J we set the derivative to zero and obtain the *normal equations*:

$$X^T X\theta = X^T y$$

Thus we have, in closed-form, the value of θ that minimises $J(\theta)$, given by

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

Note that this final step implicitly assumes that $X^T X$ is invertible. This can be checked before calculating the inverse; if either the number of linearly independent examples is fewer than the number of features, or if the features are not linearly independent:

See that a matrix $A \in \mathbb{R}^{n\times m}$ can have rank m $(= \min(n, m))$ only if $n \geq m$ (thus more linearly independent examples than features are required—m representing features and n samples.) If the features (m) were not linearly independent, the matrix A would not have rank m. See that

$$\begin{aligned}
A^T Ax = 0 &\implies x^T A^T Ax = 0 \implies (Ax)^T Ax = 0 \\
&\implies ||Ax||^2 = 0 \implies Ax = 0
\end{aligned}$$

Intuitively, only if $Ax = 0$ is the trivial solution (meaning A is a one-to-one mapping for m sized vectors), then would $A^T Ax$ also be one-to-one and also invertible.

1.1.4 Probabalistic interpretation of linear regression simplifies to LMS

Hypothesis
Here we make some probabalistic assumptions, under which least-squares regression is derived as a very natural algorithm. Assume that the target variables and the inputs are related via the equation:

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

were ϵ 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) according to a Gaussian/Normal distribution with mean zero and some variance σ^2. We can write this assumption as $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$, where $\epsilon^{(i)}$ has the probability density function:

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

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 distribution of $y^{(i)}$ can also be written as $y^{(i)}|x^{(i)};\theta \sim \mathcal{N}(\theta^T x^{(i)}, \sigma^2)$

Likelihood
Given all our data in X (the design matrix) and θ, our distribution of all the $y^{(i)}$, when viewed for a fixed value of θ (we don't actually know θ, but we come up with a function given a value), we have the *likelihood function*

$$L(\theta) = L(\theta; X, y) = p(y|X;\theta)$$

(the combined probability of all these $y^{(i)}$ occuring given X and a value of θ) By the independence assumption on the $\epsilon^{(i)}$ this can be written as

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

(next page)

Maximum Likelihood

Given this probabilistic model relating the $y^{(i)}$ and $x^{(i)}$, we should choose θ so as to maximise the probability of the data—we want to maximise $L(\theta)$—the *maximum likelihood*.

Given the monotonicity of the logarithm, and that we are simply looking for the θ to maximise the function, and not the value of the function itself, maximising over the *log likelihood* $\ell(\theta)$ simplifies things:

$$\begin{aligned}
\ell(\theta) &= \log L(\theta) \\
&= \log \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\
&= \sum_{i=1}^{n} \log \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\
&= \sum_{i=1}^{n} \log \frac{1}{\sqrt{2\pi}\sigma} + \sum_{i=1}^{n} \log \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\
&= n \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y^{(i)} - \theta^T x^{(i)})^2 \\
&= \underbrace{n \log \frac{1}{\sqrt{2\pi}\sigma}}_{\text{constant}} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^{n} (y^{(i)} - \theta^T x^{(i)})^2
\end{aligned}$$

see therefore that maximising $\ell(\theta)$ is the same as maximising

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

which is our original least-squares cost function. Note that the final choice of θ did not depend on what σ^2 was.

1.1.5 Locally Weighted Linear Regression

LWR
In the original linear regression algorithm, to make a *prediction* at a query point x (to evaluate $h(x)$), we would

1. Fit θ to minimise $\sum_i (y^{(i)} - \theta^T x^{(i)})^2$
2. Output $\theta^T x$

In contrast, the *locally weighted linear regression* algorithm does the following:

1. Fit θ to minimise $\sum_i w^{(i)}(y^{(i)} - \theta^T x^{(i)})^2$
2. Output $\theta^T x$

Here the $w^{(i)}$ are non-negative valued *weights*. Intuitively, if $w^{(i)}$ is large for a particular value of i then that particular example $x^{(i)}$ will have a larger effect on the optimisation of θ. If $w^{(i)}$ is small, then that example carries less impact on our selection of θ. A fairly standard choice for the weights is

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

See that the weights depend on the particular point x at which we're trying to predict at—our entire selection of θ *changes with each predicton*—we re-train the model each time we make a prediction.

If $|x^{(i)} - x|$ is small, then $w^{(i)}$ is close to 1; if $|x^{(i)} - x|$ is large, then $w^{(i)}$ is small—θ is chosen giving a much higher 'weight' to the (errors on) training examples close to the query point (the point we want to predict) x. The parameter τ is called the *bandwith* parameter; it controls how quickly the weight of a training example falls off with distance from the query point x.

Non-parametric vs parametric algorithms
Locally weighted linear regression is an example of a *non-parameteric* algorithm, as opposed to (unweighted) linear regression which is a *parametric* learning algorithm. Parametric learning algorithms have a fixed, finite number of parameters θ which are fit to the data and stored, after which we no longer require the training data (since all predictions are made on the same θ). In contrast, for LWR we need to keep the entire training set around (since w changes with our prediction input and we need to optimise θ again). 'Non-parametric' refers to the idea that the amound of stuff we need to keep in order to represent the hypothesis h grows with the size of the training set.

1.1.6 Binary Classification and Logistic Regression

Classification
Classification problems are applicable in situations where the values y we want to predict take on only a small number of discrete values. Here we consider *binary classification*, where y can take on only two values—0 and 1. 0 is called the *negative class* and 1 the *positive class*. For a given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the *label* for the training example.

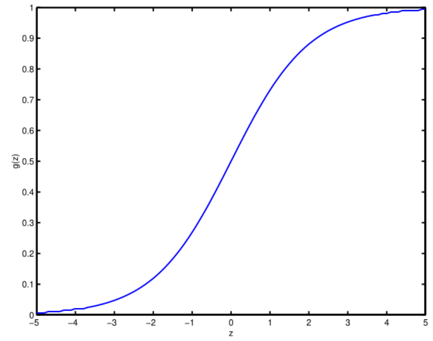
Logistic Regression
A linear regression algorithm in this case would perform poorly given the nature of the problem—intuitively it doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we already know that $y \in \{0, 1\}$. To fix this we consider a different hypothesis h:

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

where

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

is called the *logistic/sigmoid* function:



See that $g(z)$ tends toward 1 as $z \to \infty$, and to 0 as $z \to -\infty$. Also see that it is always bounded between 0 and 1. As before we keep the convention of $x_0 = 1$, so that $\theta^T x = \theta_0 + \sum_{j=1}^{d} \theta_j x_j$.

(next page)

Derivative of the Sigmoid function
Defining g as the sigmoid function, we note its derivative:

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

Probabalistic assumptions and maximum likelihood
Here we fit our model via maximum likelihood. For that we need to make a set of probabalistic assumptions; let us assume that

$$\begin{aligned}
P(y=1|x;\theta) &= h_\theta(x) \\
P(y=0|x;\theta) &= 1-h_\theta(x)
\end{aligned}$$

We can write down the probability function for a *single* trial as

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

Assuming the n training examples were generated independently, we write down the likelihood as

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

To make things easier we maximise the log likelihood:

$$\ell(\theta) = \log L(\theta) = \sum_{i=1}^{n}\left[y^{(i)}\log(h(x^{(i)}) + (1-y^{(i)})\log(1-h(x^{(i)}))\right]$$

Here we have the choice of batch or stochastic gradient *ascent*, since we are *maximising* the likelihood (same as if we maximised log likelihood).
(next page)

Stochastic gradient ascent rule
Since we are trying to *maximise*, our update rule looks like

$$\theta := \theta + \alpha \nabla_\theta \ell(\theta)$$

(Note the positive rather than the negative sign in the update). Here we consider *stochastic* gradient ascent, so we look at *one* training example and take the derivative to get the rule (if we took the derivative of the log likelihood we found earlier that would be batch gradient ascent):

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

In the second equality we used the chain rule and the derivative of g as derived earlier. We now have the stochastic gradient ascent rule:

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

This appears similar to the LMS update rule, but this time our hypothesis h is different.

Link to logistic loss
Our rule can be viewed as a loss function by defining $\ell_{\text{logistic}} : \mathbb{R} \times \{0,1\} \to \mathbb{R}_{\geq 0}$ to be the *logistic loss*:

$$\ell_{\text{logistic}}(t, y) \triangleq y \log(1 + \exp(-t)) + (1-y)\log(1+\exp(t))$$

One can verify by plugging in $h_\theta(x) = 1/(1+e^{-\theta^T x})$ (which is our hypothesis here) that the *negative* log likelihood can be rewritten as

$$-\ell(\theta) = \ell_{\text{logistic}}(\theta^T x, y)$$

(maximising the likelihood is the same as minimising the *negative* likelihood)

1.1.7 Logistic regression and Perceptrons

Consider modifying the logistic regression method to ‘force’ it to output values that are either 0 or 1 exactly. This can be done by changing the definition of g to be the threshold function:

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

If we then let $h_\theta(x) = g(\theta^T x)$ as before but using this modified definition of g, with the same update rule:

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

then we have the *perceptron learning algorithm.* Note that the perceptron is a very type of algorithm to logistic regression or least squares regression.

1.1.8 Multi-class softmax classification

Multinomial distribution
Consider a classification problem where the response variable y takes on any one of k discrete values, so $y \in \{1, 2, \dots, k\}$. We can model this as distributed according to a *multinomial distribution.*

The idea of a multinomial distribution is just a generalisation of the binomial—think instead of 2 possible outcomes per trial for n trials (binomial), we have k possible outcomes per trial for n trials (multinomial).

In this case, $p(y|x;\theta)$ is a distribution over k possible discrete outcomes (multinomial). A multinomial distribution involves k numbers $\phi_1, \dots, \phi_k$ specifying the probability of each outcome. Naturally this also means they must satisfy

$$\sum_{i=1}^{k} \phi_i = 1$$

Softmax
We introduce k groups of parameters $\theta_1, \dots, \theta_k$ (for k possible outcomes), each of them being a vector in $\mathbb{R}^d$ (corresponding to number of features). Intuitively, we would like to use $\theta_1^T x, \dots, \theta_k^T x$ to represent $\phi_1, \dots, \phi_k$ which correspond to the probabilities:

$$\phi_i = P(y = i|x;\theta)$$

However, this won't work since the all the $\theta_j^T x$ don't sum to 1, and they don't necessarily individually fall within $[0, 1]$ anyway.

Thus, we consider the *softmax* function to turn $(\theta_1^T x, \cdots, \theta_k^T x)$ into a probability vector with nonnegative entries that sum to 1. We define softmax $:\mathbb{R}^k \to \mathbb{R}^k$ as

$$\text{softmax}(t_1, \dots, t_k) = \begin{bmatrix} \frac{\exp(t_1)}{\sum_{j=1}^{k} \exp(t_j)} \\ \vdots \\ \frac{\exp(t_k)}{\sum_{j=1}^{k} \exp(t_j)} \end{bmatrix}$$

The inputs t are often called *logits.* See that the entries of the output are always nonnegative and sum to 1.
(next page)

Softmax (cont.)
Let $(t_1, \dots, t_k) = (\theta_1^T x, \cdots, \theta_k^T x)$. We apply the softmax function to $(t_1, \dots, t_k)$ and use the output as probabilities $P(y = 1|x; \theta), \dots, P(y = k|x; \theta)$:

$$\begin{bmatrix} P(y = 1|x; \theta) \\ \vdots \\ P(y = k|x; \theta) \end{bmatrix} = \text{softmax}(t_1, \cdots, t_k) = \begin{bmatrix} \frac{\exp(t_1)}{\sum_{j=1}^k \exp(t_j)} \\ \vdots \\ \frac{\exp(t_k)}{\sum_{j=1}^k \exp(t_j)} \end{bmatrix}$$

Essentially we let

$$P(y = i|x; \theta) = \phi_i = \frac{\exp(t_i)}{\sum_{j=1}^k \exp(t_j)} = \frac{\exp(\theta_i^T x)}{\sum_{j=1}^k \exp(\theta_j^T x)}$$

Likelihood and cross-entropy
Considering a *single example*, we can write down the *negative* log-likelihood as

$$-\log p(y|x, \theta) = -\log\left(\frac{\exp(t_y)}{\sum_{j=1}^k \exp(t_j)}\right) = -\log\left(\frac{\exp(\theta_y^T x)}{\sum_{j=1}^k \exp(\theta_j^T x)}\right)$$

See that we can write down the negative log-likelihood of *all the training data* as

$$\ell(\theta) = \sum_{i=1}^n -\log\left(\frac{\exp(\theta_{y^{(i)}}^T x^{(i)})}{\sum_{j=1}^k \exp(\theta_j^T x^{(i)})}\right)$$

Defining the *Cross-entropy loss* $\ell_{\text{ce}} : \mathbb{R}^k \times \{1, \dots, k\} \to \mathbb{R}_{\geq 0}$ (its just the negative log likelihood for one trial):

$$\ell_{\text{ce}}((t_1, \dots, t_k), y) = -\log\left(\frac{\exp(t_y)}{\sum_{j=1}^k \exp(t_j)}\right)$$

With that we can rewrite the negative log-likelihood of all the training data as

$$\ell(\theta) = \sum_{i=1}^n \ell_{\text{ce}}((\theta_1^T x^{(i)}, \dots, \theta_k^T x^{(i)}), y^{(i)})$$

(We want to 'tune' all the θ to maximise the likelihood/minimise the negative likelihood of the features $x^{(i)}$ 'predicting' $y^{(i)}$.)
(next page)

Minimising the cross-entropy
We had the cross-entropy

$$\ell_{\text{ce}}((t_1,\ldots,t_k),y) = -\log\left(\frac{\exp(t_y)}{\sum_{j=1}^k \exp(t_j)}\right)$$

The cross-entropy has a simple gradient. Letting $t = (t_1,\ldots,t_k)$ and recalling $\phi_i = \frac{\exp(t_i)}{\sum_{j=1}^k \exp(t_j)}$, we can derive

$$\frac{\partial \ell_{\text{ce}}(t,y)}{\partial t_i} = \phi_i - 1\{y=i\}$$

In vectorised notation we have the following form

$$\frac{\partial \ell_{\text{ce}}(t,y)}{\partial t} = \phi - e_y$$

where $e_s \in \mathbb{R}^k$ is the s-th natural basis vector (where the s-th entry is 1 and all other entries are zeros).

With respect to θ
We want the gradient with respect to θ; using the chain rule we have (for single example)

$$\frac{\partial \ell_{\text{ce}}((\theta_1^T x,\ldots,\theta_k^T x),y)}{\partial \theta_i} = \frac{\partial \ell(t,y)}{\partial t_i}\cdot\frac{\partial t_i}{\partial \theta_i} = (\phi_i - 1\{y=i\})\cdot x$$

We now have the gradient of the loss with respect to θ_i for the training data as

$$\frac{\partial \ell(\theta)}{\partial \theta_i} = \sum_{j=1}^n (\phi_i^{(j)} - 1\{y^{(j)} = i\})\cdot x^{(j)}$$

Remember that $\phi_i = \frac{\exp(t_i)}{\sum_{j=1}^k \exp(t_j)}$ denotes the predicted probability *given the training example* and θ, and thus its value changes with each example (hence the superscript).

1.1.9 Derivation of gradient for cross-entropy loss

The cross-entropy loss is given by

$$\ell_{\mathrm{ce}}((t_1,\ldots,t_k),y) = -\log\left(\frac{\exp(t_y)}{\sum_{j=1}^{k}\exp(t_j)}\right)$$

Letting $t = (t_1,\ldots,t_k)$ and recalling $\phi_i = \frac{\exp(t_i)}{\sum_{j=1}^{k}\exp(t_j)}$, we take the gradient with respect to t_i. This falls into two cases:

First case: $y = i$
Using the chain rule:

$$\frac{\partial}{\partial t_y}(-\log(\phi_y)) = \frac{\partial}{\partial \phi_y}(-\log(\phi_y)) \cdot \frac{\partial \phi_y}{\partial t_y} = -\frac{1}{\phi_y} \cdot \frac{\partial \phi_y}{\partial t_y}$$

now product rule:

$$\begin{aligned} -\frac{1}{\phi_y} \cdot \frac{\partial \phi_y}{\partial t_y} &= -\frac{1}{\phi_y} \cdot \left(\phi_y + \left(-\frac{(\exp(t_y))^2}{\left(\sum_{j=1}^{k}\exp(t_j)\right)^2}\right)\right) \\ &= -1 + \frac{\sum_{j=1}^{k}\exp(t_j)}{\exp(t_y)} \cdot \frac{(\exp(t_y))^2}{\left(\sum_{j=1}^{k}\exp(t_j)\right)^2} \\ &= \frac{\exp(t_y)}{\sum_{j=1}^{k}\exp(t_j)} - 1 = \phi_y - 1 \end{aligned}$$

Second case: $y \neq i$
Same as before

$$\frac{\partial}{\partial t_{i\neq y}}(-\log(\phi_y)) = \frac{\partial}{\partial \phi_y}(-\log(\phi_y)) \cdot \frac{\partial \phi_y}{\partial t_{i\neq y}} = -\frac{1}{\phi_y} \cdot \frac{\partial \phi_y}{\partial t_{i\neq y}}$$

but now instead we have

$$-\frac{1}{\phi_y} \cdot \frac{\partial \phi_y}{\partial t_{i\neq y}} = \frac{\sum_{j=1}^{k}\exp(t_j)}{\exp(t_y)} \cdot \frac{\exp(t_y)\exp(t_{i\neq y})}{\left(\sum_{j=1}^{k}\exp(t_j)\right)^2} = \phi_{i\neq y}$$

Compact form
We can therefore write

$$\frac{\partial \ell_{\mathrm{ce}}(t,y)}{\partial t_i} = \phi_i - 1\{y = i\}$$

Where $1\{\cdot\}$ denotes the indicator function.

1.1.10