‘cs229’—Notes

Malcolm

Started 1st October 2024

Chapter 1

Supervised learning

Given a dataset of n *training examples* $\{(x^{(i)}, y^{(i)}); i = 1\}, \ldots, n\}$—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.

1.0.1 Linear Regression

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

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

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.

1.0.2 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 {
    $$\theta_j := \theta_j + \alpha \sum_{i=1}^{n} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}, \text{ (for every } j)$$
}
```

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

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)

1.0.3 Gradients of

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 $b^T x$:
For $x \in \mathbb{R}^n$ and $f(x) = b^T x$ 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 $f(x) = x^T A x$:
Now consider the quadratic function $f(x) = x^T A x$ 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

$$b_1 = \sum_{i=1}^{n} A_{1i} x_i$$

$$\vdots$$

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

(next page)

so we can write

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

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

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

also analagous to single variable calculus. (next page)

Hessian

1.0.4 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* $\boldsymbol{X}$ 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:

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

and $\boldsymbol{y}$ the n-dimensional vector containing the target values:

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

See that we can represent our least squares function over the entire dataset as