5 Linear regression

5.1 Least squares linear regression

In this Section we formally describe the problem of *linear regression*, or the fitting of a representative line (or hyperplane in higher dimensions) to a set of input/output data points. Regression in general may be performed for a variety of reasons: to produce a so-called trend line that can be used to help visually summarize, drive home a particular point about the data under study, or to learn a model so that precise predictions can be made regarding output values in the future.

5.1.1 Notation and modeling

Data for regression problems generally come in the form of a set of P input/output observation pairs

$$\left(\mathbf{x}_1, y_1\right), \left(\mathbf{x}_2, y_2\right), ..., \left(\mathbf{x}_P, y_P\right) \tag{5.1}$$

or $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ for short, where $\mathbf{x}_p$ and y_p denote the input and output of the p^{th} observation respectively. Each input $\mathbf{x}_p$ is in general a column vector of length N

$$\mathbf{x}_p = \begin{bmatrix} x_{1,p} \\ x_{2,p} \\ \vdots \\ x_{N,p} \end{bmatrix} \tag{5.2}$$

and each output y_p is scalar-valued (we consider vector-valued outputs in Section 5.5). Geometrically speaking, the linear regression problem is then one of fitting a *hyperplane* to a scatter of points in $N + 1$ dimensional space.

In the simplest instance where the inputs are also scalar-valued (i.e., $N = 1$), linear regression simplifies to fitting a *line* to the associated scatter of data points in two dimensional space. A line in two dimensions is determined by two parameters: a vertical intercept w_0 and slope w_1. We must set the values of

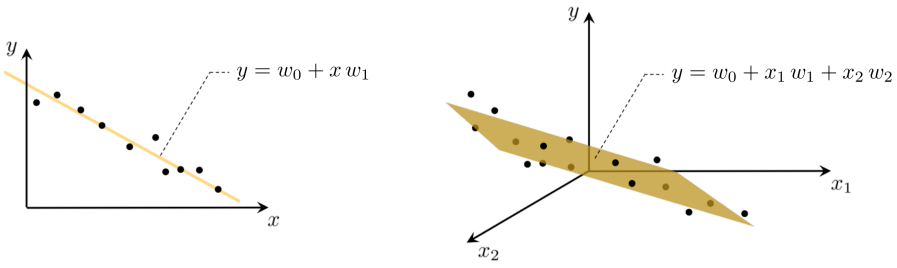

Figure 5.1 (left panel) A simulated dataset in two dimensions along with a well-fitting line. A line in two dimensions is defined as $w_0 + xw_1 = y$, where w_0 is referred to as the bias and w_1 the slope, and a point $\left(x_p, y_p\right)$ lies close to it if $w_0 + x_p w_1 \approx y_p$. (right panel) A simulated three dimensional dataset along with a well-fitting hyperplane. A hyperplane in general is defined as $w_0 + x_1 w_1 + x_2 w_2 + \cdots + x_N w_N = y$, where again w_0 is the bias and w_1, w_2, $\ldots$, w_N the hyperplane's coordinate-wise slopes, and a point $\left(\mathbf{x}_p, y_p\right)$ lies close to it if $w_0 + x_{1,p} w_1 + x_{2,p} w_2 + \cdots + x_{N,p} w_N \approx y_p$. Here $N = 2$.

these parameters in a way that the following approximate linear relationship holds between the input/output data

$$w_0 + x_p w_1 \approx y_p, \quad p = 1, ..., P. \tag{5.3}$$

Notice that we have used the approximately equal sign in Equation () because we can never be absolutely sure that all data lies completely on a single line. More generally, when dealing with N dimensional input we have a bias weight and N associated slope weights to tune properly in order to fit a hyperplane, with the analogous linear relationship written as

$$w_0 + x_{1,p} w_1 + x_{2,p} w_2 + \cdots + x_{N,p} w_N \approx y_p, \quad p = 1, ..., P. \tag{5.4}$$

Both the single-input and general mutli-input cases of linear regression are illustrated figuratively in Figure 5.1. Each dimension of the input is referred to as a *feature* or *input feature* in machine learning parlance. Hence we will often refer to the parameters w_1, w_2, ..., w_N as the *feature-touching weights*, the only weight not touching a feature is the bias w_0.

The linear relationship in Equation (5.4) can be written more compactly, using the notation $\mathring{\mathbf{x}}$ to denote an input $\mathbf{x}$ with a 1 placed on top of it. This means that we stack a 1 on top of each of our input points $\mathbf{x}_p$

$$\mathring{\mathbf{x}}_p = \begin{bmatrix} 1 \\ x_{1,p} \\ x_{2,p} \\ \vdots \\ x_{N,p} \end{bmatrix} \tag{5.5}$$

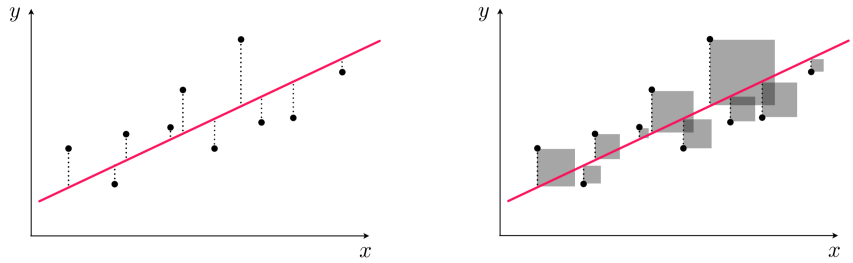

Figure 5.2 (left panel) A prototypical two dimensional dataset along with a line fit to the data using the Least Squares framework, which aims at recovering the linear model that minimizes the total squared length of the dashed error bars. (right panel) The Least Squares error can be thought of as the total area of the gray squares, having dashed error bars as sides. The cost function is called *Least Squares* because it allows us to determine a set of parameters whose corresponding line minimizes the sum of these square errors.

for all $p = 1, ..., P$. Now placing all parameters into a single column vector $\mathbf{w}$

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \tag{5.6}$$

we may write the general desired linear relationships in Equation (5.4) more compactly as

$$\mathring{\mathbf{x}}_p^T \mathbf{w} \approx y_p \quad p = 1, ..., P. \tag{5.7}$$

5.1.2 The Least Squares cost function

To find the parameters of the hyperplane that best fits a regression dataset we must first form a cost function that measures how well a a linear model with a particular choice of parameters fits the regression data. One of the more popular choices for doing this is called the *Least Squares* cost function. For a given set of parameters in the vector $\mathbf{w}$ this cost function computes the total *squared error* between the associated hyperplane and the data, as illustrated pictorially in the Figure 5.2. Naturally then the best fitting hyperplane is the one whose parameters *minimize* this error.

Focusing on just the p^{th} approximation in Equation (5.7) we ideally want $\mathring{\mathbf{x}}_p^T \mathbf{w}$ to be as close as possible to the p^{th} output y_p, or equivalently, the error or difference between the two, i.e., $\mathring{\mathbf{x}}_p^T \mathbf{w} - y_p$, to be as small as possible. By squaring this quantity (so that both negative and positive errors of the same magnitude are treated equally) we can define

$$g_p\left(\mathbf{w}\right) = \left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2 \tag{5.8}$$

as a *point-wise cost function* that measures the error of a model (here a linear one) on the individual point $\left(\mathbf{x}_p, y_p\right)$. Now since we want all P such values to be small simultaneously we can take their average over the entire dataset, forming the *Least Squares cost function*[1] for linear regression

$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P} g_p\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2. \tag{5.9}$$

Notice that the larger the Least Squares cost becomes the larger the squared error between the corresponding linear model and the data, and hence the poorer we represent the given dataset using a linear model. Therefore we want to find the optimal parameter vector $\mathbf{w}$ that *minimizes* $g\left(\mathbf{w}\right)$, or written formally we want to solve the unconstrained optimization problem

$$\underset{\mathbf{w}}{\text{minimize}}\ \frac{1}{P}\sum_{p=1}^{P}\left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2. \tag{5.10}$$

using the tools of local optimization detailed in Chapters 2,3, and 4.

5.1.3 Minimization of the Least Squares cost function

The Least Squares cost function for linear regression in Equation (5.9) can be proven to be *convex* for any dataset (see Section 5.7). In Example 5.1 we showcase this fact using a toy linear regression dataset.

Example 5.1 **Verifying convexity by visual inspection**

The top panel of Figure 5.3 shows a toy linear regression dataset, consisting of $P = 50$ input/output pairs randomly selected off of the line $y = x$, with a small amount of random noise added to each output. In the bottom-left panel of this Figure we plot the three dimensional surface of the Least Squares cost fucntion

[1] Technically speaking, the Least Squares cost function $g\left(\mathbf{w}\right)$ is a function of both the the weights $\mathbf{w}$ as well as the data. However, for notational simplicity we often choose not to show the dependency on data explicitly. Otherwise we would have to write the cost function as

$$g\left(\mathbf{w}, \left\{\left(\mathring{\mathbf{x}}_p, y_p\right)\right\}_{p=1}^{P}\right)$$

and things start to get too messy. Moreover, for a given dataset the weights $\mathbf{w}$ are the important input to the function as they are what we need to tune in order to produce a good fit. From an optimization perspective the dataset itself is considered *fixed*. We will make this sort of notational simplification for virtually all future machine learning cost functions we study as well.

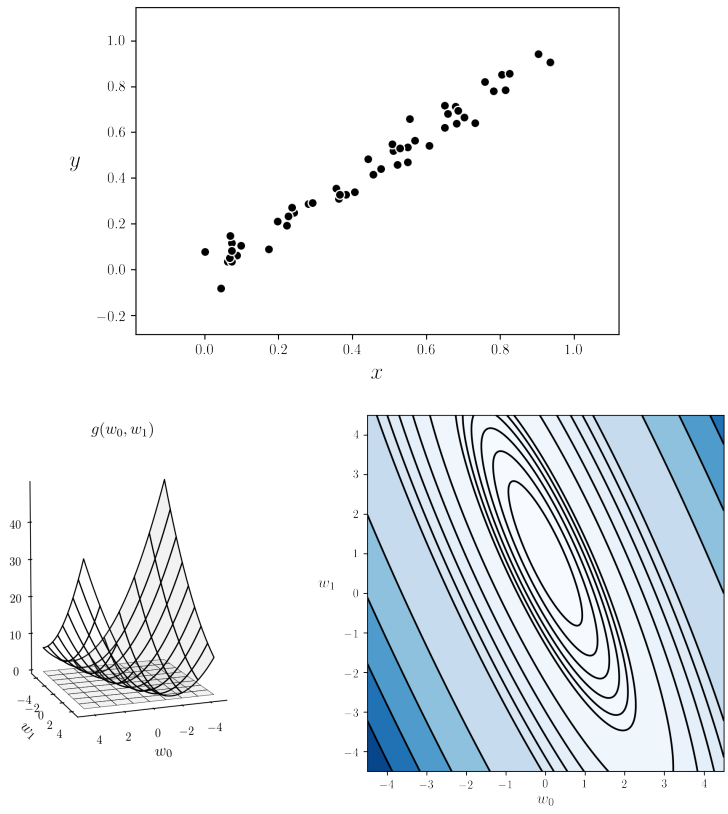

Figure 5.3 Figure associated with Example 5.1. See text for details.

associated with this dataset, with its contour plot shown in two dimensions in the bottom-right panel. We can see, by the *upward bending* shape of the cost function's surface on the left or by the elliptical shape of its countor lines on the right, that the Least Squares cost function is indeed convex for this particular dataset.

Because of its convexity, and because the Least Squares cost function is *infinitely differentiable*, we can apply virtually any local optimization scheme to minimize it properly. However the generic practical considerations associated with each local optimization method still apply: that is, the zero and second order methods do not scale gracefully, and with gradient descent we must choose a fixed steplength value, a diminishing steplength scheme, or an adjustable method like backtracking line search (as detailed in Chapter 3). Because the Least Squares cost is a convex quadratic function, a *single* Newton step can per-

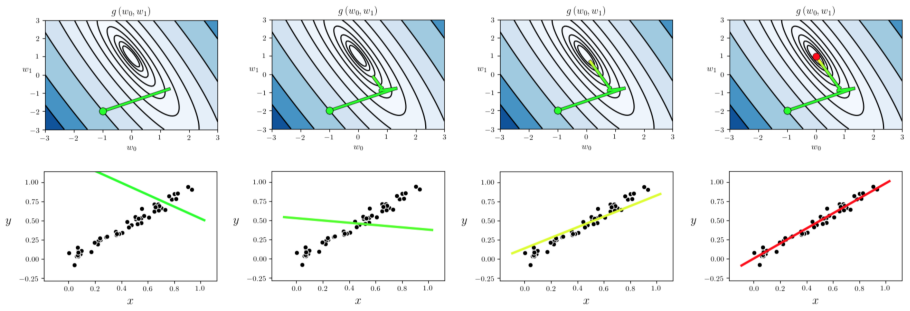

Figure 5.4 Figure associated with Example 5.2. See text for details.

fectly minimize it. This is sometimes referred to as minimizing the Least Squares via solving its *normal equations* (see Exercise 3 for further discussion).

Example 5.2 Minimizing the Least Squares cost function using gradient descent

In Figure 5.4 we show the result of minimizing the Least Squares cost using the toy dataset presented in Example 5.1. We use gradient descent and employ a fixed steplength value $\alpha = 0.5$ for all 75 steps until approximately reaching the minimum of the function.

The Figure shows progression of the gradient descent process (from left to right) both in terms of the Least Squares cost minimization (top row) and line provided by the corresponding weights (bottom row). The optimization steps in the top row are colored from green at the start of the run (leftmost panel) to red at its finale (rightmost panel). The linear model is colored to match the step of gradient descent (green near the beginning and red towards the end). Examining the figure, as gradient descent approaches the minimum of the cost function the corresponding parameters provide a better and better fit to the data, with the best fit occurring at the end of the run at the point closest to the Least Squares cost's minimizer.

Whenever we use a local optimization method like gradient descent we must properly tune the steplength parameter α (as described previously, e.g., in Section 3.6). In Figure 5.4 we show the cost function history plot for two steplength values: $\alpha = 0.01$ (in purple), and $\alpha = 0.5$ (in black) which we ended up using for the run shown in Figure 5.5. This illustrates why (in machine learning contexts) the steplength parameter is often referred to as the *learning rate*, since this value does indeed determine how quickly the proper parameters of our linear regression model (or any machine learning model in general) are learned.

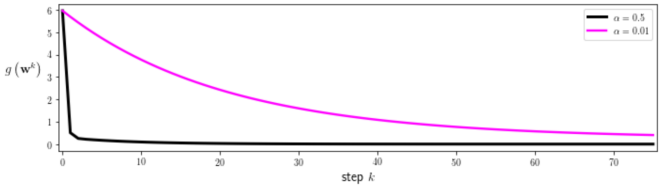

Figure 5.5 Figure associated with Example 5.2. See text for details.

5.1.4 Python implementation

When implementing a cost function like Least Squares it is helpful to think in a modular fashion, with the aim of lightening the amount of mental 'bookkeeping' required, by breaking down the cost into a few distinct components. Here we break the cost function down into two main parts: the *model* that is a linear combination of input and weights, and the *cost* itself (i.e., squared error).

We express our (linear) model as a function worthy enough of its own notation, as

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) = \mathring{\mathbf{x}}_p^T \mathbf{w}. \tag{5.11}$$

If we were to go back and use this new modeling notation we could re-write our ideal settings of the weights in Equation (5.7), as

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) \approx y_p \tag{5.12}$$

and likewise our Least Squares cost function in Equation (5.9), as

$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\left(\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) - y_p\right)^2. \tag{5.13}$$

This kind of simple deconstruction of the Least Squares cost lends itself to an organized, modular, and easily extendable implementation. Starting with the model, note that while it is more compact and convenient *mathematically* to write the linear combination $\mathring{\mathbf{x}}_p^T \mathbf{w}$ by tacking a 1 on top of the raw input $\mathbf{x}_p$, in *implementing* this we can more easily compute the linear combination by exposing the bias and feature-touching weights *separately* as

$$\mathring{\mathbf{x}}_p^T \mathbf{w} = w_0 + \mathbf{x}_p^T \boldsymbol{\omega}. \tag{5.14}$$

where vector $\boldsymbol{\omega}$ contains all the feature-touching weights

$$\boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}. \tag{5.15}$$

Remember that w_0 is called the *bias* since it controls where our linear model pierces the y axis, and $w_1, w_2, ..., w_N$ are called *feature-touching weights* because they touch each individual dimension of the input (which in the jargon of machine learning are called *features*).

Using the efficient `numpy`'s `np.dot` operation[2] must be replaced in Pyhton we can now implement our linear model as

```
a = w[0] + np.dot(x_p.T,w[1:])
```

which matches the right-hand-side of Equation (5.14) where `w[0]` denotes the bias w_0 and `w[1:]` denotes the remaining N feature-touching weights in $\boldsymbol{\omega}$. Wrapping this into a `Python` function we have our linear model implemented as

```
# compute linear combination of input point
def model(x_p,w):
    # compute linear combination and return
    a = w[0] + np.dot(x_p.T,w[1:])
    return a.T
```

which we can then use to form the associated Least Squares cost function

```
# a least squares function for linear regression
def least_squares(w,x,y):
    # loop over points and compute cost contribution from each input/
        output pair
    cost = 0
    for p in range(y.size):
        # get pth input/output pair
        x_p = x[:,p][:,np.newaxis]
        y_p = y[p]

        ## add to current cost
        cost += (model(x_p,w)  - y_p)**2

    # return average least squares error
    return cost/float(y.size)
```

Notice here we explicitly show the *all* of the inputs to the cost function here, not just the $(N+1)\times 1$ weights $\mathbf{w}$ - whose `Python` variable is denoted `w`. The Least Squares cost also takes in all inputs (with ones stacked on top of each point) $\mathring{\mathbf{x}}_p$ - which together we denote by the $(N+1)\times P$ `Python` variable `x` as well as the entire set of corresponding outputs which we denote as the $1\times P$ variable `y`.

[2] As a general rule whenever vectorized implementations are available, one must refrain from implementing algebraic expressions in `Python` entry-wise using for instance, explicit `for` loops.

Notice that this really is a direct implementation of the algebraic form of the cost in Equation 5.1.4, where we think of the cost as the sum of squared errors of a linear `model` of input against its corresponding output. However *explicit* `for` loops (including list comprehensions) written in `Python` are rather slow due to the very nature of the language.

It is easy to get around most of this inefficiency by replacing explicit `for` loops with numerically equivalent operations performed using operations from the `numpy` library. `numpy` is an API for some very efficient vector/matrix manipulation libraries written in `C`. Broadly speaking, when writing a `Pythonic` function like this one with heavy use of `numpy` functionality one tries to package each step of computation - which previously was being formed sequentially at each data point - together for the entire dataset simultaneously. This means we do away with the explicit `for` loop over each of our P points and make the same computations (numerically speaking) for every point simultaneously. Below we provide one such `numpy` heavy version of the Least Squares implementation shown previously which is far more efficient.

Note that in using these functions the input variable `x` (containing the entire set of P inputs) is size $N \times P$, and its corresponding output `y` is size $1 \times P$. Here we have written this code - and in particular the model function - to mirror its respective formula as close as possible.

```
# compute linear combination of input points
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T

# an implementation of the least squares cost function for linear
    regression
def least_squares(w):
    # compute the least squares cost
    cost = np.sum((model(x,w) - y)**2)
    return cost/float(y.size)
```

Notice too that for simplicity we write the the `Pythonic` Least Squares cost function `least_squares(w)` instead of `least_squares(w,x,y)`, where in the latter case we explicitly list its other two arguments: the input `x` and output `y` data. This is done for notational simplicity - we do this with our math notation as well denoting our Least Squares cost $g(\mathbf{w})$ instead of $g(\mathbf{w}, \mathbf{x}, \mathbf{y})$ - and either format is perfectly fine practically speaking as `autograd` (see Section 7.6) will correctly differentiate both forms (since by default it computes the gradient of a `Python` function with respect to its first input only). We will use this kind of simplified `Pythonic` notation when introducing future machine learning cost functions as well.

While we recommend most users employ the automatic differentiator (`autograd` (see Section 3.5) to perform both gradient descent and Newton's method on our

machine learning cost functions, here one can (since this cost function is simple enough to) 'hard code' the gradient by formally by writing it out 'by hand' (using the derivative rules detailed in Section 7.2). Doing so one can compute the gradient of the Least Squares cost in closed form as

$$\nabla g\left(\mathbf{w}\right) = \frac{2}{P}\sum_{p=1}^{P}\mathring{\mathbf{x}}_p\left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right) \tag{5.16}$$

Furthermore, the in performing Newton's method one can also compute the Hessian of the Least Squares cost by hand. Moreover since the cost is a convex quadratic *only a single Newton step can completely minimize it*. This single Newton step solution is often referred to as minimizing the Least Squares cost via its *normal equations*. The system of equations solved in taking this single Newton step is equivalent to the *first order system* (see Section 3.2) for the Least Squares cost function

$$\left(\sum_{p=1}^{P}\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T\right)\mathbf{w} = \sum_{p=1}^{P}\mathring{\mathbf{x}}_p y_p. \tag{5.17}$$

5.2 Least Absolute Deviations

In this Section we discuss a slight twist on the derivation of the Least Squares cost function that leads to an alternative cost for linear regression called *Least Absolute Deviations*. This alternative cost function is much more robust to outliers in a dataset than the original Least Squares.

5.2.1 Susceptibility of Least Squares to outliers

One downside of using the squared error in the Least Squares cost (as a measure that we then minimize to recover optimal linear regression parameters) is that *squaring the error increases the importance of larger errors*. In particular, squaring errors of length greater than 1 makes these values considerably larger. This forces weights learned via the Least Squares cost to produce a linear fit that is especially focused on trying to minimize these large errors, sometimes due to *outliers* in a dataset. In other words, the Least Squares cost produces linear models that tend to *overfit* to outliers in a dataset. We illustrate this fact via a simple dataset in Example 5.3.

Example 5.3 **Least Squares overfits to outliers**

In this Example we use the dataset plotted in Figure 5.6, which can largely be represented by a proper linear model with the exception of a single outlier,

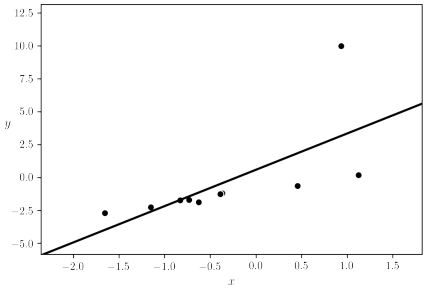

Figure 5.6 Figure associated with Example 5.3. See text for details.

to show how the Least Squares cost function for linear regression tends to create linear models that overfit to outliers. We tune the parameters of a linear regressor to this dataset by minimizing the Least squares cost via gradient descent (see Section 3.6), and plot the associated linear model on top of the data. This fit (shown in black) does not fit the majority of the data points well, bending upward clearly with the aim of minimizing the large squared error on the singleton outlier point.

5.2.2 Replacing squared error with absolute error

Our original derivation of the Least Squares cost function in Section 5.1 aimed at learning a set of ideal weights so that we have

$$\mathring{\mathbf{x}}_p^T\mathbf{w} \approx y_p \qquad p = 1, ..., P \tag{5.18}$$

for a dataset of P points $\left\{\left(\mathbf{x}_p,\, y_p\right)\right\}_{p=1}^{P}$. We then squared the difference between both sides of each desired approximation

$$g_p\left(\mathbf{w}\right) = \left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2 \qquad p = 1, ..., P \tag{5.19}$$

and took the average of these P squared error terms to form the full Least Squares cost function.

As an alternative to using a *squared error* for our point-wise cost in Equation 5.19 we can instead measure the *absolute error* for each desired approximation

$$g_p\left(\mathbf{w}\right) = \left|\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right| \qquad p = 1, ..., P. \tag{5.20}$$

By using absolute error instead of the squared version we still treat negative and positive errors equally, but we do not exaggerate the importance of large errors greater than 1. Taking the average of these absolute error point-wise costs gives us the cousin of Least Squares, the so-called *Least Absolute Deviations* cost function

$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P} g_p\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\left|\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right|. \tag{5.21}$$

The only price we pay in employing the absolute error instead of the squared error is a technical one: while this cost function is also always convex regardless of the input dataset, since its second derivative is zero (almost everywhere) we can only use zero and first order methods to properly minimize it (and not second order methods).

Example 5.4 **Least Squares versus Least Absolute Deviations**

In Figure 5.7 we compare the result of tuning a linear model by minimizing the Least Squares versus the Least Absolute Deviation cost functions employing the dataset in Example 5.3. In both cases we run gradient descent for the same number of steps and using the same choice of steplength parameter. We show the cost function histories for both runs in the bottom panel of Figure 5.7, where the runs using the Least Squares and Least Absolute Deviations are shown in black and magenta respectively. Examining the cost function histories we can see that the cost function value of the Least Absolute Deviations cost is considerably lower than that of Least Squares. This alone provides evidence that the former will provide a considerably better fit than Least Squares.

This advantage can also be seen in the top panel of Figure 5.7 where we plot and compare the best fit line provided by each run. The Least Squares fit is shown in black, while the Least Absolute Deviation fit is shown in magenta. The latter fits considerably better, since it does not exaggerate the large error produced by the single outlier.

5.3 Regression quality metrics

In this brief Section we describe how to make predictions using a trained regression model followed by simple metrics for judging the quality of such a model.

5.3.1 Making predictions using a trained model

If we denote the optimal set of weights found by minimizing a regression cost function by $\mathbf{w}^\star$ then our fully trained linear model can be written as

$$\text{model}\left(\mathbf{x}, \mathbf{w}^\star\right) = \mathring{\mathbf{x}}^T\mathbf{w}^\star = w_0^\star + x_1 w_1^\star + x_2 w_2^\star + \cdots + x_N w_N^\star. \tag{5.22}$$

Regardless of how we determine optimal parameters $\mathbf{w}^\star$, by minimizing a

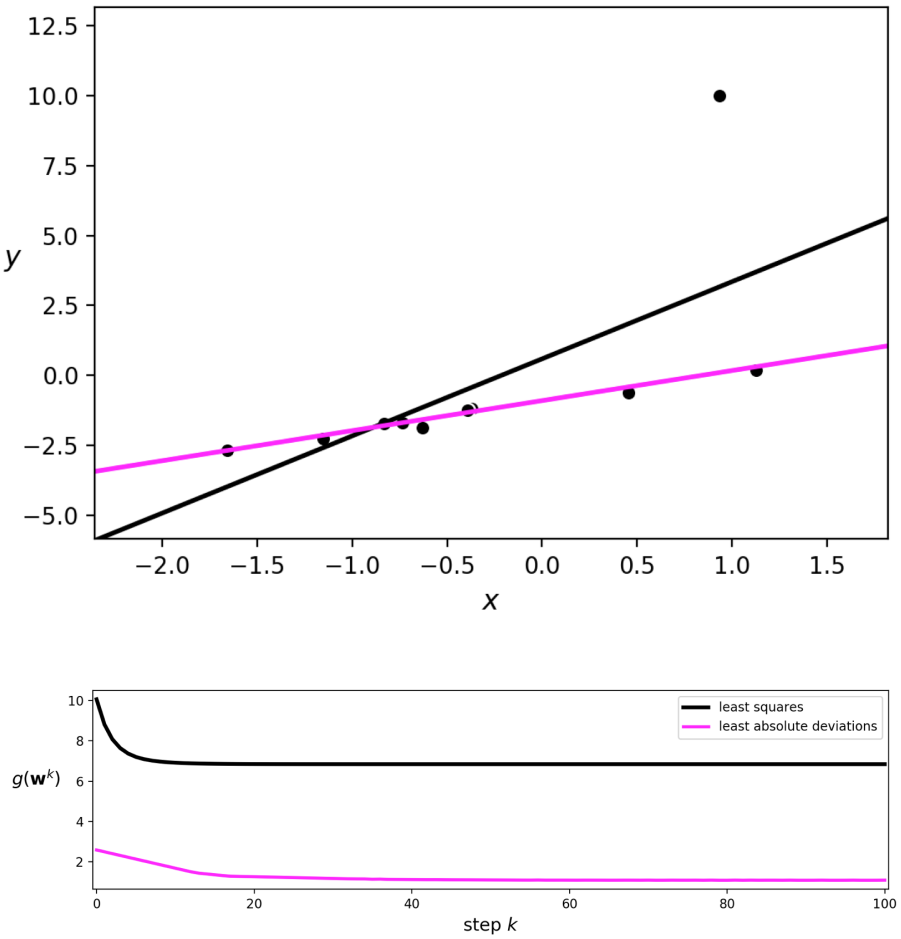

Figure 5.7 Figure associated with Example 5.4. See text for details.

regression cost like the Least Squares or Least Absolute Deviations, we make predictions employing our linear model in the same way. That is, given an input $\mathbf{x}_0$ (whether it is from our training dataset or a brand new input) we predict its output y_0 by simply passing it along with our trained weights into our model as

$$\text{model}\left(\mathbf{x}_0, \mathbf{w}^\star\right) = y_0 \tag{5.23}$$

This is illustrated pictorially on a prototypical linear regression dataset in Figure 5.8.

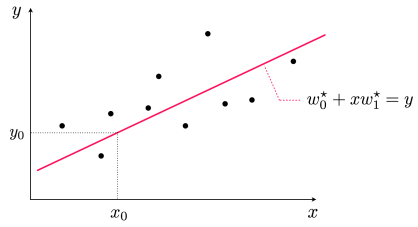

Figure 5.8 Once optimal parameters $w_0^\star$ and $w_1^\star$ of a regression line are found via minimizing an appropriate cost function they may be used to predict the output value of any input x_0 by substituting it into Equation (5.22). Here $N = 1$.

5.3.2 Judging the quality of a trained model

Once we have successfully minimized a linear regression cost function it is an easy matter to determine the quality of our regression model: we simply evaluate a cost function using our optimal weights. For example, we can evaluate the quality of a trained model using the Least Squares cost, which is especially natural to use when we employ this cost in training. To do this we plug in our learned model parameters along with the data into the Least Squares cost, giving the so-called Mean Squared Error (or MSE for short)

$$\text{MSE} = \frac{1}{P}\sum_{p=1}^{P}\left(\text{model}\left(\mathbf{x}_p, \mathbf{w}^\star\right) - y_p\right)^2. \tag{5.24}$$

The name for this regression quality metric describes precisely what the Least Squares cost computes, i.e., the average (or mean) squared error.

In the same way we can employ the Least Absolute Deviations cost to determine the quality of our trained model. Plugging in our learned model parameters along with the data into this cost computes the Mean Absolute Deviations (or MAD for short) which is precisely what this cost function computes

$$\text{MAD} = \frac{1}{P}\sum_{p=1}^{P}\left|\text{model}\left(\mathbf{x}_p, \mathbf{w}^\star\right) - y_p\right|. \tag{5.25}$$

These two metrics differ in precisely the ways we have seen their respective cost functions differ (e.g., the MSE measure is far more sensitive to outliers). In general the *lower* one can make these quality metrics (by proper tuning of model weights) the *better* the quality of the corresponding trained model, and vice versa. However the threshold for what one considers 'good' or 'great' performance can depend on personal preference, an occupational or institutionally-set benchmark, or some other problem-dependent concern.

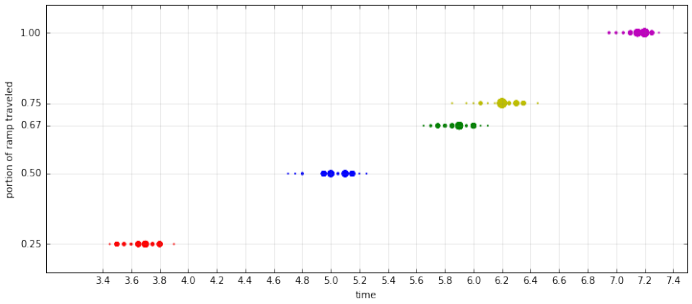

Figure 5.9 Figure associated with Example 5.5. See text for details.

5.4 Weighted regression

Because regression cost functions can be decomposed over individual datapoints, we see in this Section that it is possible to weight these points in order to emphasize or de-emphasize their importance to a regression model. This practice is called *weighted regression*.

5.4.1 Dealing with duplicates

Imagine we have a linear regression dataset that contains multiple copies of the same point, generated not by error but for example by necessary quantization (or binning) of input features in order to make human-in-the-loop analysis or modeling of the data easier. Needless to say, 'duplicate' datapoints should not be thrown away in a situation like this.

Example 5.5 **Quantization of input features can create duplicate points.**

In Figure 5.9 we show a raw set of data from a modern reenactment of Galileo's famous ramp experiment where, in order to quantify the effect of gravity, he repeatedly rolled a ball down a ramp to determine the relationship between distance and amount of time it takes an object to fall to the ground. This dataset consists of multiple trials of the same experiment, where each output's numerical value has been rounded to two decimal places. Performing this natural numerical rounding (sometimes referred to as *quantizing*) produces multiple duplicate datapoints, which we denote visually by scaling the dot representing each point in the image. The larger the dot's radius, the more duplicate points it represents.

Let us now examine what happens to a regression cost fucntion (e.g., Least Squares) when a dataset contains duplicate datapoints. Specifically we assume there are β_p versions of the input/output pair $\left(\mathbf{x}_p,\, y_p\right)$ in our data. For regression datasets we have seen so far (excluding the one shown in Figure 5.9) we have

always had $\beta_p = 1$ for all $p = 1, \ldots, P$. Using our `model` notation to represent our linear model (see e.g., Section 5.3.1) we can write the sum of all point-wise squared errors as

$$\begin{aligned} &\underbrace{\left(\text{model}\left(\mathbf{x}_1, \mathbf{w}\right) - y_1\right)^2 + \cdots + \left(\text{model}\left(\mathbf{x}_1, \mathbf{w}\right) - y_1\right)^2}_{\beta_1} \\ &+ \underbrace{\left(\text{model}\left(\mathbf{x}_2, \mathbf{w}\right) - y_2\right)^2 + \cdots + \left(\text{model}\left(\mathbf{x}_2, \mathbf{w}\right) - y_2\right)^2}_{\beta_2} \\ &\vdots \\ &+ \underbrace{\left(\text{model}\left(\mathbf{x}_P, \mathbf{w}\right) - y_P\right)^2 + \cdots + \left(\text{model}\left(\mathbf{x}_P, \mathbf{w}\right) - y_P\right)^2}_{\beta_P} \end{aligned} \tag{5.26}$$

The natural grouping in Equation (5.26) then helps us write the overall Least Squares cost function as

$$g\left(\mathbf{w}\right) = \frac{1}{\beta_1 + \beta_2 + \cdots + \beta_P} \sum_{p=1}^{P} \beta_p \left(\text{model}\left(x_p, \mathbf{w}\right) - y_p\right)^2 \tag{5.27}$$

As we can see here the Least Squares cost function naturally *collapses* into a weighted version of itself in the sense that we can combine summands so that a repeated point in the dataset is represented in the cost function by a single weighted summand. Since the weights β_1, β_2, ..., β_P are fixed for any given dataset, we can minimize a weighted regression cost precisely as we would any other (by tuning $\mathbf{w}$ alone). Finally notice that setting $\beta_p = 1$ (for all p) in Equation (5.27) gives us back the original (unweighted) Least Squares cost function in Equation (5.1.4).

5.4.2 Weighting points by confidence

Weighted regression can also be employed when we wish to weight each point based on our confidence in the *trustworthiness* of each datapoint. For example if our dataset came in two batches, one batch from a trustworthy source and another from a less trustworthy source (where some datapoints could be noisy or fallacious), we would want to weight datapoints from the trustworthy source more in our final regression. This can be done very easily using precisely the weighted regression paradigm introduced previously, only now we set the weights β_1, β_2, ..., β_P ourselves based on our confidence of each point. If we believe that a point is very trustworthy we can set its corresponding weight β_p

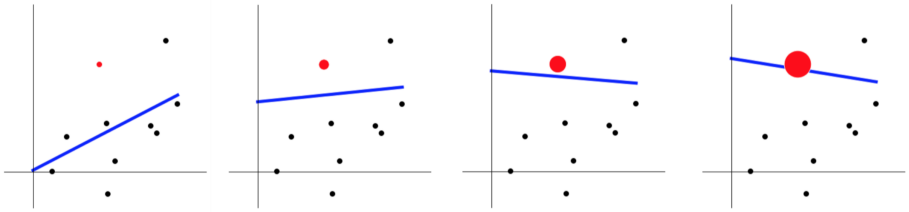

Figure 5.10 Figure associated with Example 5.6. See text for details.

high, and vice versa. Notice in the extreme case, a weight value of $\beta_p = 0$ effectively removes its corresponding datapoint from the cost function, implying we do not trust that point at all.

Example 5.6 **Adjusting a single datapoint's weight to reflect confidence**

In Figure 5.10 we show how adjusting the weight associated with a single datapoint affects the final learned model in a weighted linear regression setting. The toy regression dataset shown in this Figure includes a red datapoint whose diameter changes in proportion to its weight. As the weight (which can be interpret as 'confidence') is increased from left to right, the regressor focuses more and more on representing the red point. If we increase its weight enough the fully trained regression model naturally starts fitting to this single datapoint alone while disregarding all other points, as illustrated in the rightmost panel of Figure 5.10.

5.5 Multi-output regression

Thus far we have assumed that datapoints for linear regression consist of *vector-valued* inputs and *scalar-valued* outputs. In other words, a prototypical regression datapoint takes the form of an input/output pair $\left(\mathbf{x}_p,\, y_p\right)$ where the input $\mathbf{x}_p$ is an N dimensional vector, and the output y_p a scalar. While this configuration covers the vast majority of regression cases one may encounter in practice, it is possible to perform (linear) regression where *both* input and output are vector-valued. This is often called *multi-output regression* which we now discuss.

5.5.1 Notation and modeling

Suppose our regression dataset consists of P input/output pairs

$$\left(\mathbf{x}_1, \mathbf{y}_1\right),\, \left(\mathbf{x}_2, \mathbf{y}_2\right),\, ...,\, \left(\mathbf{x}_P, \mathbf{y}_P\right) \tag{5.28}$$

where each input $\mathbf{x}_p$ is N dimensional and each output $\mathbf{y}_p$ is C dimensional. While in principle we can treat $\mathbf{y}_p$ as a $C \times 1$ column vector, in order to keep the formulae that follows looking similar to what we have already seen in the scalar case we will treat the input as an $N \times 1$ *column* vector and the output as a $1 \times C$ *row* vector, as[3]

$$\mathbf{x}_p = \begin{bmatrix} x_{1,p} \\ x_{2,p} \\ \vdots \\ x_{N,p} \end{bmatrix} \qquad \mathbf{y}_p = \begin{bmatrix} y_{0,p} & y_{1,p} & \cdots & y_{C-1,p} \end{bmatrix}. \tag{5.29}$$

If we assume that a *linear* relationship holds between the input $\mathbf{x}_p$ and *just* the c^{th} dimension of the output $y_{c,p}$, we are back to precisely the sort of regression framework we have seen thus far, and we can write

$$\mathring{\mathbf{x}}_p^T \mathbf{w}_c \approx y_{c,p} \qquad p = 1, ..., P \tag{5.30}$$

where $\mathbf{w}_c$ is a set of weights

$$\mathbf{w}_c = \begin{bmatrix} w_{0,c} \\ w_{1,c} \\ w_{2,c} \\ \vdots \\ w_{N,c} \end{bmatrix} \tag{5.31}$$

and $\mathring{\mathbf{x}}_p$ is the vector formed by stacking 1 on top of $\mathbf{x}_p$. If we then further assume that a linear relationship holds between the input and *all* C entries of the output, we can place each weight vector $\mathbf{w}_c$ into the c^{th} column of an $(N+1) \times C$ weight *matrix* $\mathbf{W}$ as

$$\mathbf{W} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,c} & \cdots & w_{0,C-1} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,c} & \cdots & w_{1,C-1} \\ w_{2,0} & w_{2,1} & \cdots & w_{2,c} & \cdots & w_{2,C-1} \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ w_{N,0} & w_{N,1} & \cdots & w_{N,c} & \cdots & w_{N,C-1} \end{bmatrix} \tag{5.32}$$

and write the entire set of C linear models via a vector-matrix product

$$\mathring{\mathbf{x}}_p^T \mathbf{W} = \begin{bmatrix} \mathring{\mathbf{x}}_p^T \mathbf{w}_0 & \mathring{\mathbf{x}}_p^T \mathbf{w}_1 & \cdots & \mathring{\mathbf{x}}_p^T \mathbf{w}_{C-1} \end{bmatrix} \tag{5.33}$$

This allows us to write the entire set of C linear relationships very compactly as

[3] Notice that unlike the input we index the output starting from 0. We do this because eventually we will stack a 1 on top of each input $\mathbf{x}_p$ (as we did with standard regression in Section 5.1) and this entry will have the 0^{th} index of our input.

$$\mathring{\mathbf{x}}_p^T\mathbf{W} \approx \mathbf{y}_p \qquad p = 1, ..., P. \tag{5.34}$$

5.5.2 Cost functions

The thought process involved in deriving a regression cost function for the case of multi-output regression mirrors almost exactly the scalar-output case discussed in Sections 5.1 and 5.2. For example, to derive a Least Squares cost function we begin (in the same way we did in Section 5.1) by taking the difference of both sides of Equation (5.34). However the error associated with the p^{th} point, written as $\mathring{\mathbf{x}}_p^T\mathbf{W} - \mathbf{y}_p$, is now a vector of C values. To square this error we must therefore employ the *squared* ℓ_2 vector norm (see Section 7.12 if not familiar with this vector norm). The Least Squares cost function in this case is then the average squared ℓ_2 norm of each point's error, written as

$$g(\mathbf{W}) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathring{\mathbf{x}}_p^T\mathbf{W} - \mathbf{y}_p\right\|_2^2 = \frac{1}{P}\sum_{p=1}^{P}\sum_{c=0}^{C-1}\left(\mathring{\mathbf{x}}_p^T\mathbf{w}_c - y_{c,p}\right)^2. \tag{5.35}$$

Notice that when $C = 1$, this reduces to the original Least Squares cost we saw in Section 5.1.

Likewise, the Least Absolute Deviations cost (which measures the absolute value of each error as opposed to its square) for our present case takes the analogous form

$$g(\mathbf{W}) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathring{\mathbf{x}}_p^T\mathbf{W} - \mathbf{y}_p\right\|_1 = \frac{1}{P}\sum_{p=1}^{P}\sum_{c=0}^{C-1}\left|\mathring{\mathbf{x}}_p^T\mathbf{w}_c - y_{c,p}\right| \tag{5.36}$$

where $\|\cdot\|_1$ is the ℓ_1 vector norm, the generalization of the absolute value function for vectors (see Section 7.12.1 if not familiar with this vector norm).

Just like their scalar-valued versions, these cost functions are always convex regardless of the dataset used. They also decompose over the weights $\mathbf{w}_c$ associated with each output dimension. For example, we can rewrite the right hand side of the Least Absolute Deviations cost in Equation (5.36) by swapping the summands over P and C, giving

$$g(\mathbf{W}) = \sum_{c=0}^{C-1}\left(\frac{1}{P}\sum_{p=1}^{P}\left|\mathring{\mathbf{x}}_p^T\mathbf{w}_c - y_{c,p}\right|\right) = \sum_{c=0}^{C-1} g_c(\mathbf{w}_c) \tag{5.37}$$

where we have denoted $g_c(\mathbf{w}_c) = \frac{1}{P}\sum_{p=1}^{P}\left|\mathring{\mathbf{x}}_p^T\mathbf{w}_c - y_{c,p}\right|$. Since the weights from each of the C subproblems do not interact we can, if desired, minimize each g_c for an optimal setting of $\mathbf{w}_c$ independently, and then take their sum to form the full cost function g.

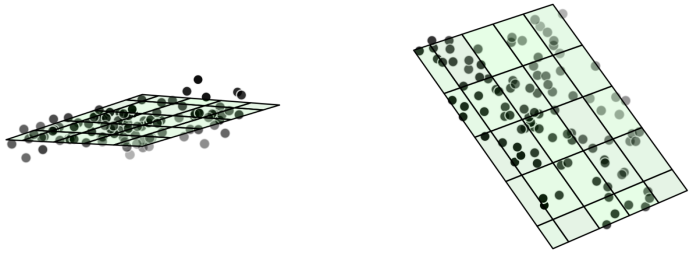

Figure 5.11 Figure associated with Example 5.7. See text for details.

Example 5.7 **Fitting a linear model to a multi-output regression dataset**

In Figure 5.11 we show an example of multi-output linear regression using a toy dataset with input dimension $N = 2$ and output dimension $C = 2$, where we have plotted the input and *one* output value in each of the two panels of the Figure.

We tune the parameters of an appropriate linear model via minimizing the Least Squares cost using gradient descent, and illustrate the fully trained model (shown in green in each panel) by evaluating a fine mesh of points in the input region of the dataset.

5.5.3 `Python` implementation

Because `Python` and `numpy` have such flexible syntax, we can implement the linear model

$$\text{model}\left(\mathbf{x}, \mathbf{W}\right) = \mathring{\mathbf{x}}^T\mathbf{W} \tag{5.38}$$

precisely as we did in the scalar-output case in Section 5.1.4. In *implementing* this linear combination we need not form the adjusted input $\mathring{\mathbf{x}}_p$ (by tacking a 1 on top of the raw input $\mathbf{x}_p$) and can more easily compute the linear combination by exposing the biases as

$$\mathring{\mathbf{x}}_p^T\mathbf{W} = \mathbf{b} + \mathbf{x}_p^T\boldsymbol{\mathcal{W}} \tag{5.39}$$

where we denote bias $\mathbf{b}$ and feature-touching weights $\boldsymbol{\mathcal{W}}$ as

$$\mathbf{b} = \begin{bmatrix} w_{0,0} \\ w_{0,1} \\ w_{0,2} \\ \vdots \\ w_{0,C-1} \end{bmatrix} \qquad \boldsymbol{\mathcal{W}} = \begin{bmatrix} w_{0,1} & w_{0,2} & \cdots & w_{0,C-1} \\ w_{1,1} & w_{1,2} & \cdots & w_{1,C-1} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,C-1} \\ \vdots & \vdots & \vdots & \vdots \\ w_{N,1} & w_{N,2} & \cdots & w_{N,C-1} \end{bmatrix}. \tag{5.40}$$

This notation is used to match the `Pythonic` slicing operation (as shown in

the implementation given below), which we implement in `Python` analogously as

```
a = w[0] + np.dot(x_p.T,w[1:])
```

That is $\mathbf{b} = w[0]$ denotes the bias and $\mathcal{W} = w[1:]$ denotes the remaining feature-touching. Another reason to implement in this way is that the particular linear combination $\mathbf{x}_p^T\mathcal{W}$ - implemented using `np.dot` as `np.dot(x_p.T,w[1:])` below - is an especially efficient since `numpy`'s `np.dot` operation is far more efficient than constructing a linear combination in `Python` via an explicit `for` loop.

`Pythonic` implementation of regression cost functions can also be implemented precisely as we have seen previously. For example, our linear model and Least Squares cost can be written as shown below.

```
# linear model
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T

# least squares cost
def least_squares(w):
    cost = np.sum((model(x,w) - y)**2)
    return cost/float(np.size(y))
```

5.6 Exercises

Section 5.1 exercises

1. **Fitting a regression line to the student debt data**

   Fit a linear model to the U.S. student load debt dataset shown in Figure 1 by minimizing the associated linear regression Least Squares problem using a single Newton step (also known as solving the normal equations). If this linear trend continues what will be the total student debt be in 2050?

2. **Kleiber's law and linear regression**

   After collecting and plotting a considerable amount of data comparing the body mass versus metabolic rate (a measure of at rest energy expenditure) of a variety of animals, early 20^{th} century biologist Max Kleiber noted an interesting relationship between the two values. Denoting by x_p and y_p the

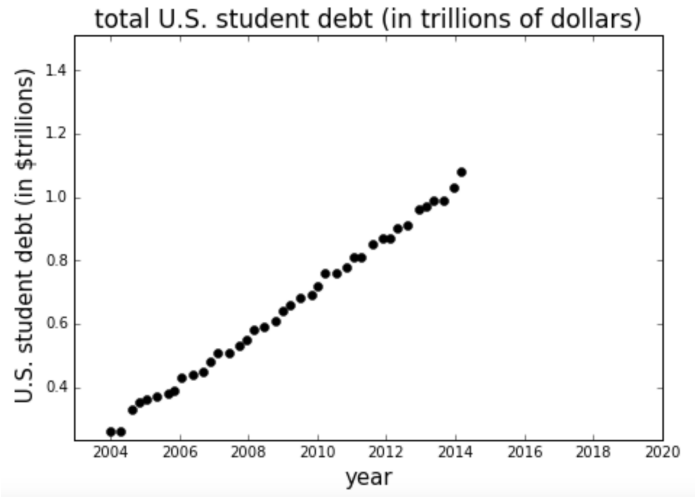

Figure 5.12 Figure associated with Exercise 1. See text for details.

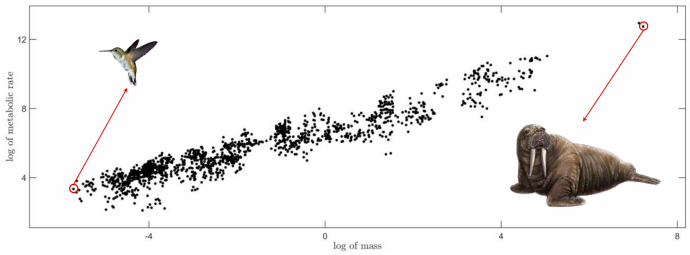

Figure 5.13 Figure associated with Exercise 2. A large set of body mass/metabolic rate data points, transformed by taking the log of each value, for various animals over wide range of different masses.

body mass (in kg) and metabolic rate (in kJ/day) of a given animal respectively, treating the body mass as the input feature Kleiber noted (by visual inspection) that the natural log of these two values were linearly related. That is

$$w_0 + \log\left(x_p\right) w_1 \approx \log\left(y_p\right). \tag{5.41}$$

In the Figure below we show a large collection of transformed data points $\left\{\left(\log\left(x_p\right), \log\left(y_p\right)\right)\right\}_{p=1}^{P}$, each representing an animal ranging from a small black-chinned hummingbird in the bottom left corner to a large walrus in the top right corner.

a) Fit a linear model to the data shown in Figure 2 Make sure to take the log of both arguments!

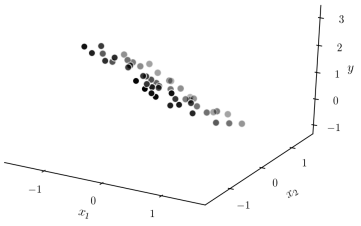

Figure 5.14
Figure associated with Example 3 . See text for further details.

b) Use the optimal parameters you found in part **a)** along with the properties of the log function to write the nonlinear relationship between the body mass x and the metabolic rate y.

c) Use your fitted line to determine how many calories an animal weighing 10 kg requires (note each calorie is equivalent to 4.18 J)

3. **Completely minimize the Least Squares cost function using a single Newton step**

   As mentioned in Section 5.1.3, a single Newton step can perfectly minimize the Least Squares cost for linear regression. Use a single Newton step to perfectly minimize the Least Squares cost over the dataset shown in Figure 5.14. This dataset roughly lies on a hyperplane, thus the fit provided by a perfectly minimized Least Squares cost should fit very well. Use a cost function history plot to check that you have tuned the parameters of your linear model properly.

4. **Lipschitz constant for the Least Squares cost**

   Compute the Lipschitz constant (see Section 3.12.4) of the Least Squares cost function.

Section 5.2 exercises

5. **Empirically confirm convexity for a toy dataset**

   Empirically confirm that the Least Absolute Deviations cost function is convex for the toy dataset shown in Section 5.2.

6. **Compare the Least Squares and Least Absolute Deviation costs**

Repeat the experiment outlined in Example 5.4. You will need to implement the Least Absolute Deviations cost, which can be done similar to the Least Squares implementation in Section 5.1.4.

Section 5.5 exercises

7. **Multi-output regression**

Repeat the experiment outlined in Example 5.7. You can use the implementation described in Section 5.5.3 as a base for your implementation.

5.7 Endnotes

5.7.1 Proof that the Least Squares cost function is always convex

A little re-arrangement shows that the Least Squares cost function for linear regression is always a convex quadratic, and hence is a convex function. Here we will briefly ignore the bias term w_0 for notational simplicity, but the same argument holds with it as well.

We can do this by first examining just the p^{th} summand. By expanding (performing the squaring operation) we have

$$\left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2 = \left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)\left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right) = y_p^2 - 2\mathring{\mathbf{x}}_p^T\mathbf{w}y_p + \mathring{\mathbf{x}}_p^T\mathbf{w}\mathring{\mathbf{x}}_p^T\mathbf{w} \tag{5.42}$$

where we have arranged the terms in increasing order of degree.

Now - since the inner product $\mathring{\mathbf{x}}_p^T\mathbf{w} = \mathbf{w}^T\mathring{\mathbf{x}}_p$ we can switch around the second inner product in the first term on the right, giving equivalently

$$= y_p^2 - 2\mathring{\mathbf{x}}_p^T\mathbf{w}y_p + \mathbf{w}^T\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T\mathbf{w} \tag{5.43}$$

This is only the p^{th} summand. Summing over all the points gives analogously

$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\left(y_p^2 - 2\mathring{\mathbf{x}}_p^T\mathbf{w}y_p + \mathbf{w}^T\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}y_p^2 - \frac{2}{P}\sum_{p=1}^{P}y_p\mathring{\mathbf{x}}_p^T\mathbf{w} + \frac{1}{P}\sum_{p=1}^{P}\mathbf{w}^T\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T\mathbf{w} \tag{5.44}$$

And from here we can spot that indeed the Least Squares cost function is a quadratic, since denoting

$$\begin{aligned} a &= \frac{1}{P}\sum_{p=1}^{P} y_p^2 \\ \mathbf{b} &= -\frac{2}{P}\sum_{p=1}^{P} \mathring{\mathbf{x}}_p y_p \\ \mathbf{C} &= \frac{1}{P}\sum_{p=1}^{P} \mathring{\mathbf{x}}_p \mathring{\mathbf{x}}_p^T \end{aligned} \tag{5.45}$$

we can write the Least Squares cost equivalently as

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\,\mathbf{w} \tag{5.46}$$

which is of course a general quadratic. But furthermore because the matrix $\mathbf{C}$ is constructed from a sum of *outer product* matrices it is also convex, since the eigenvalues of such a matrix are always non-negative (see Section 4.1 and 4.2 for further details about convex quadratic functions of this form).