code

share

★ Our Project On GitHub

(https://github.com/jermwatt/machine_learning_refined)

5.2 Least Squares Linear Regression*

* The following is part of an early draft of the second edition of **Machine Learning Refined**. The published text (with revised material) is now available on Amazon (https://www.amazon.com/Machine-Learning-Refined-Foundations-Applications/dp/1108480721) as well as other major book retailers. Instructors may request an examination copy from Cambridge University Press (https://www.cambridge.org/us/academic/subjects/engineering/communications-and-signal-processing/machine-learning-refined-foundations-algorithms-and-applications-2nd-edition?format=HB).

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 (or - more generally - a curve) 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.

In [2]:

Notation and modeling

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

$$\left(\mathbf{x}_1, y_1\right), \left(\mathbf{x}_2, y_2\right), \ldots, \left(\mathbf{x}_P, y_P\right) \tag{1}$$

or $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ for short, where $\mathbf{x}_p$ and y_p denote the p^{th} input and output respectively. In simple instances the input is scalar-valued (the output will always be considered scalar-valued here), and hence the linear regression problem is geometrically speaking one of fitting a line to the associated scatter of data points in 2-dimensional space. In general however each input $\mathbf{x}_p$ may be a column vector of length N

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

in which case the linear regression problem is analogously one of fitting a hyperplane to a scatter of points in $N+1$ dimensional space.

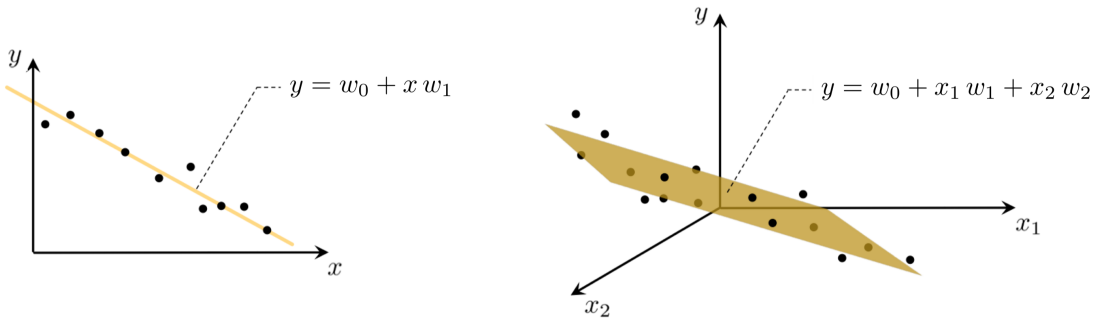


Figure 1: *(left panel) A simulated dataset in two dimensions along with a well-fitting line. A line in two dimensions is defined as* $w_0 + xw_1 = y$*, where* w_0 *is referred to as the bias and* w_1 *the slope, and a point* (x_p, y_p) *lies close to it if* $w_0 + x_p w_1 \approx y_p$*. (right panel)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

In the case of scalar input the fitting of a line to the data requires we determine a vertical intercept w_0 and slope w_1 so that the following approximate linear relationship holds between the input/output data

$$w_0 + x_p w_1 \approx y_p, \quad p = 1, \ldots, P. \tag{3}$$

Notice that we have used the approximately equal sign because we cannot be sure that all data lies completely on a single line. More generally, when dealing with N dimensional input we have a bias and N associated slope weights to tune properly in order to fit a hyperplane, with the analogous linear relationship written as

$$w_0 + x_{1,p} w_1 + x_{2,p} w_2 + \cdots + x_{N,p} w_N \approx y_p, \quad p = 1, \ldots, P. \tag{4}$$

Because each dimension of the *input* is referred to as a *feature* or *input feature* in machine learning parlance, we will often refer to $w_{1,p}$, $w_{2,p}$, $\ldots$, $w_{N,p}$ as the *feature-touching weights* (the only weight *not* touching a feature is the bias w_0).

For any N we can write the above more compactly - in particular using the notation $\mathring{\mathbf{x}}$ to denote an input $\mathbf{x}$ with a 1 placed on top of it as

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \qquad \mathring{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}. \tag{5}$$

In particular, this means that we stack a 1 on top of each of our input points $\mathbf{x}_p$ as

$$\mathring{\mathbf{x}}_p = \begin{bmatrix} 1 \\ x_{1,p} \\ x_{2,p} \\ \vdots \\ x_{N,p} \end{bmatrix}, \quad p = 1, \ldots, P \tag{6}$$

so that we may write our desired linear relationships in equation (4) more compactly as

$$\mathring{\mathbf{x}}_p^T \mathbf{w} \approx y_p \quad p = 1, \ldots, P. \tag{7}$$

The bottom N elements of an input vector $\mathring{\mathbf{x}}_p$ are referred to as *input features* to a regression problem. For instance, in the GDP growth rate data described in the Example below the first element of the input feature vector might contain the feature *unemployment rate* (that is, the unemployment data from each country under study), the second might contain the feature *education level*, and so on.

Example 1: Predicting Gross Domestic Product (GDP) growth rates

As an example of a regression problem with vector-valued input consider the problem of predicting the growth rate of a country's Gross Domestic Product (GDP), which is the value of all goods and services produced within a country during a single year. Economists are often interested in understanding factors (e.g., unemployment rate, education level, population count, land area, income level, investment rate, life expectancy, etc.,) which determine a country's GDP growth rate in order to inform better financial policy making. To understand how these various features of a country relate to its GDP growth rate economists often perform linear regression.

In the Figure below we show a heat map of the world where countries are color-coded based on their GDP growth rate in 2013, as reported by the International Monetary Fund (IMF).

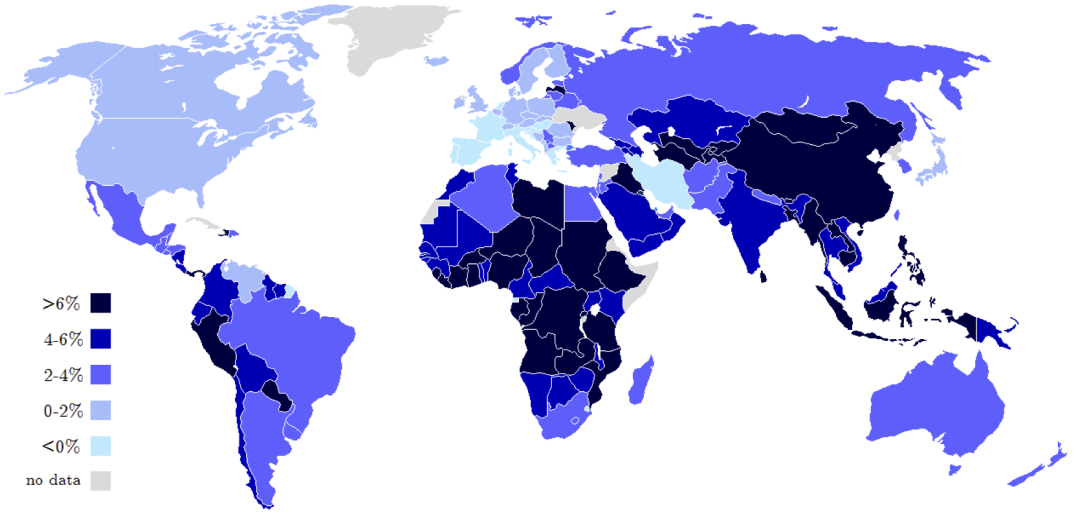


Figure 2: *A map of the world where countries are color-coded by their GDP growth rates (the darker the color the higher the growth rate) as reported by the International Monetary Fund (IMF) in 2013.*

The Least Squares cost function

To find the parameters of the hyperplane which best fits a regression dataset, it is common practice to first form the *Least Squares cost function*. For a given set of parameters $\mathbf{w}$ this cost function computes the total squared error between the associated hyperplane and the data (as illustrated pictorially in the Figure below), giving a good measure of how well the particular linear model fits the dataset. Naturally then the best fitting hyperplane is the one whose parameters minimize this error.

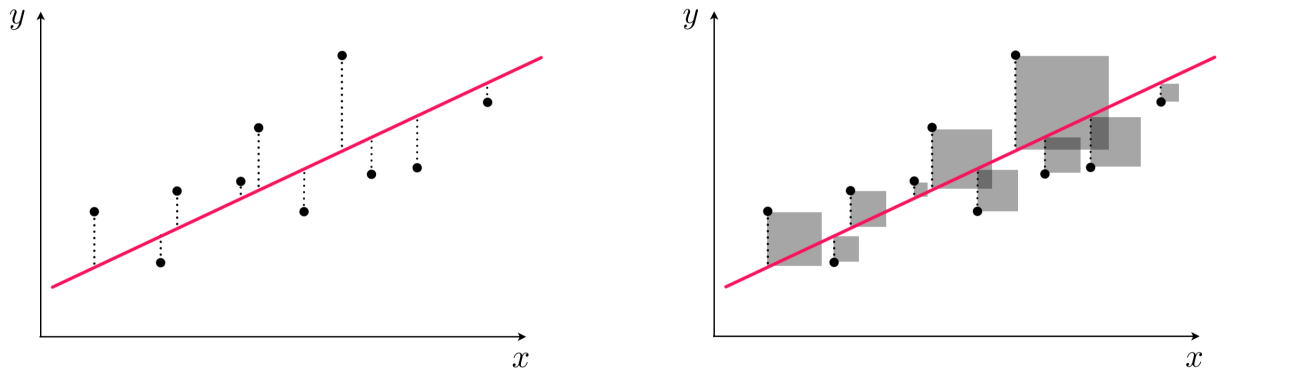


Figure 3: *(left panel) A simulated two dimensional dataset along with a line fit to the data using the Least Squares framework, which aims at recovering the line that minimizes the total squared length of the dashed error bars. (right panel) The Least Squares error can be thought of as the total area of the gray squares, having dashed error bars as sides.*

We want to find a weight vector $\mathbf{w}$ so that each of P approximate equalities

$$\mathring{\mathbf{x}}_p^T\mathbf{w} \approx y_p \tag{8}$$

holds as well as possible. Another way of stating the above is to say that the *error* between $\mathring{\mathbf{x}}_p^T\mathbf{w}$ and y_p is small. One natural way to measure error between two quantities like this measure its *square* (so that both negative and positive errors are treating equally) as

$$g_p\left(\mathbf{w}\right) = \left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2. \tag{9}$$

This squared error $g_p\left(\cdot\right)$ is one example of a *point-wise cost* that measures the error of a model (here a linear one) on the point $\left\{\mathbf{x}_p, y_p\right\}$.

Since we want all P such values to be small we can take their average - forming a *Least Squares* cost function

$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P} g_p\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2 \tag{10}$$

for linear regression.

Note that the Least Squares cost function is not just a function of the weights $\mathbf{w}$, but of the data as well - however when we express the function in mathematical shorthand as $g\left(\mathbf{w}\right)$ (as we do on the lefthand side above) we only show dependency on the weights $\mathbf{w}$. Why do we do this? Largely for notational simplicity: if we show dependency in our functional shorthand and write $g\left(\mathbf{w}; \left\{\mathring{\mathbf{x}}_p, y_p\right\}_{p=1}^P\right)$ things start to get too messy. Moreover, for a given dataset the weights $\mathbf{w}$ are the important input - since this is what we need to tune in order to produce a good fit. Because of this we will often refer to the Least Squares cost using the notation $g\left(\mathbf{w}\right)$, but the reader can keep in mind this subtle point that it is indeed a function of the data as well. We will make this sort of notational simplification for virtually all future machine learning cost functions we study as well.

Indeed we want to tune our parmeters $\mathbf{w}$ to *minimize* the Least Squares cost, since the larger this value 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. In other words, we want to determine a value for weights $\mathbf{w}$ that *minimizes* $g\left(\mathbf{w}\right)$, or written formally we want to solve the unconstrained problem

$$\underset{\mathbf{w}}{\text{minimize}}\ \frac{1}{P}\sum_{p=1}^{P}\left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2 \tag{11}$$

Implementing the Least Squares cost in `Python`

When implementing a cost function like Least squares it is helpful to think modularly, with the aim lightening the amount of mental 'bookkeeping' required, breaking down the cost into a few distinct pieces. Here we can really break things down into two chunks: we have our *model* - a linear combination of input - and the cost (squared error) itself.

We can express our linear model - a function of our input and weights - is a function worthy enough of its own notation. We will write it as

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) = \mathring{\mathbf{x}}_p^T\mathbf{w}. \tag{12}$$

If we were to go back then and use this modeling notation we could equally well e.g., our ideal settings of the weights in equation (7) as

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) \approx y_p \tag{13}$$

and likewise our Least Squares cost function itself as

$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\left(\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) - y_p\right)^2. \tag{14}$$

This kind of simple deconstruction of the Least Squares cost lends itself to an organized and modular implementation. First we can implement the linear model as shown below. Note here that while it is more convenient *mathematically* to write the linear combination $\mathring{\mathbf{x}}_p^T\mathbf{w}$, in *implementing* this 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 bias and feature-touching weights *seperately* as

$$\mathring{\mathbf{x}}_p^T\mathbf{w} = b + \mathbf{x}_p^T\boldsymbol{\omega}. \tag{15}$$

Here we use the following bias / feature weight notation

$$\text{(bias): } b = w_0 \quad \text{(feature-touching weights): } \quad \boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}. \tag{16}$$

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

This notation is used to match the `Pythonic` slicing operation (as shown in the implementation given below), which we implement in `Python` analagously as

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

That is $b = w[0]$ denotes the bias and $\boldsymbol{\omega} = w[1:]$ denotes the remaining N feature-touching weights $w_1, w_2, \ldots, w_N$. Another reason to implement in this way is that the particular linear combination $\mathbf{x}_p^T\mathbf{w}_{[1:]}$ - implemented using `np.dot` as `np.dot(x_p.T,w[1:])` below - is an especially effecient since `numpy` 's `np.dot` operation is far more effecient than constructing a linear combination in `Python` via an explicit `for` loop.

In [3]:
```
# 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
```

Then, with our linear model implemented we can easily use it to form the associated Least Squares cost function like below. 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` .

In [4]:
```
# 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 that this really is a direct implementation of the algebraic form of the cost in equation (13), where we think of the cost modularly 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 (e.g., it being a dynamically typed interpreted language (https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/)).

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 (http://www.numpy.org/). `numpy` is an API for some very efficient vector/matrix manipulation libraries written in `C`. In fact `Python` code, employing heavy use of `numpy` functions, can often execute almost as fast a raw `C` implementation itself.

Broadly speaking, when scribing 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 we have written this code - and in particular the model function - to mirror its respective formula notationally as close as possible.

In [5]:
```
# 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 $g\left(\mathbf{w}\right)$ instead of $g\left(\mathbf{w}, \mathbf{x}, \mathbf{y}\right)$ - and either format is perfectly fine practically speaking as `autograd` 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.

Minimization of the Least Squares cost function

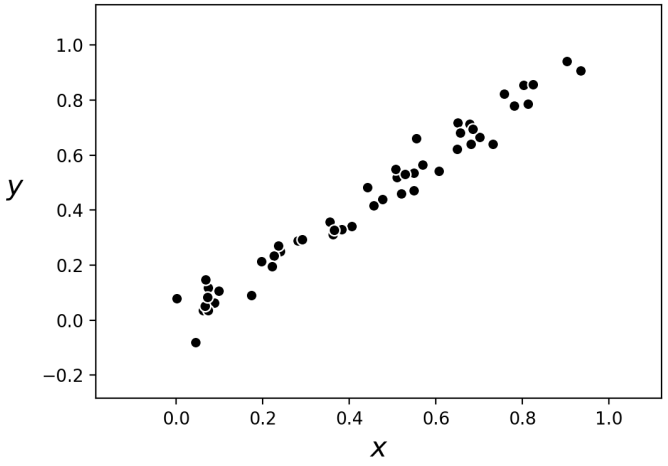
Now, determining the overall shape of a function - i.e., whether or not a function is convex - helps determine the appropriate optimization method(s) we can apply to efficiently determine the ideal parameters. In the case of the Least Squares cost function for linear regression it is easy to check that *the cost function is always convex regardless of the dataset*.

For small input dimensions (e.g., $N = 1$) we can empirically verify this claim for any given dataset by simply plotting the function g - as a surface and/or contour plot - as we do in the example below.

Example 2: Visually verifying the convexity of the cost function for a toy dataset

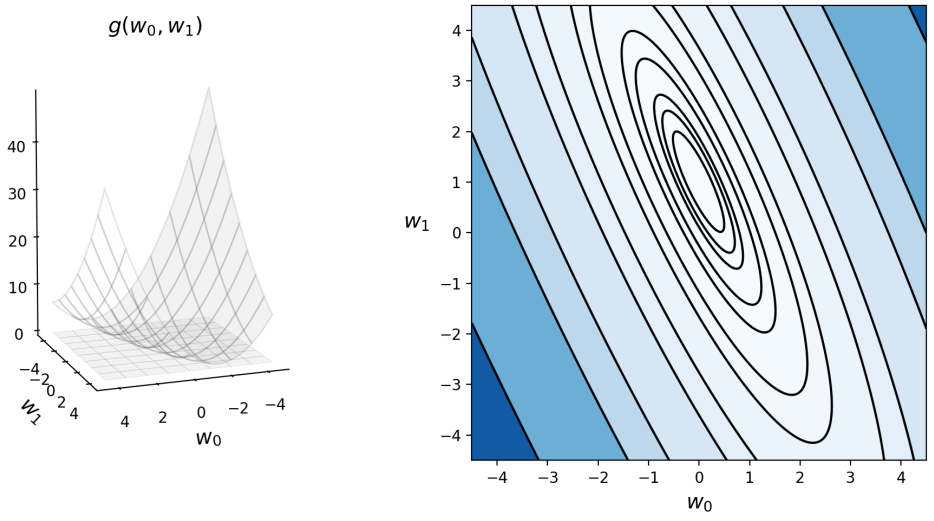
In this example we plot the contour and surface plot for the Least Squares cost function for linear regression for a toy dataset. This toy dataset consists of 50 points randomly selected off of the line $y = x$, with a small amount of Gaussian noise added to each. Notice that the data is packaged in a $(N + 1) \times P$ array, with the input being in the top N rows and the corresponding output is the last row.

In [6]:



The contour plot and corresponding surface generated by the Least Squares cost function using this data are shown below.

In [7]:



In the previous example we plotted the contour/surface for the Least Squares cost function for linear regression on a specific dataset. There we saw the elliptical contours and 'upward bending' shape of the surface indeed confirms the function's convexity in that case. However the Least Squares cost function for linear regression can mathematically shown to be - in general - a convex function for *any dataset* (this is because one can show that it is always a convex quadratic - which is shown formally below). Because of this we can easily apply either gradient descent or Newton's method in order to minimize it.

> The Least Squares cost function for linear regression is always convex regardless of the input dataset, hence we can easily apply first or second order methods to minimize it.

The generic practical considerations associated with each method still exist here i.e., with gradient descent we must choose a steplength / learning rate scheme, and Newton's method is practically limited to cases when N is of moderate value (e.g., in the thousands). For the case of gradient descent we can use a fixed steplength value, a diminishing steplength scheme, or an adjustable method like backtracking line search.

While we will by default be using an automatic differentiator (`autograd` - as detailed in Section 3.5 (https://jermwatt.github.io/machine_learning_refined/notes/3_First_order_methods/3_5_Automatic.html)) 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) 'hard code' the gradient by formally by writing it out 'by hand' (using the derivative rules detailed in the Appendix). Doing so one can compute the gradient of the Least Squares cost in closed form as

$$\nabla g\left(\mathbf{w}\right) = \frac{2}{P}\sum_{p=1}^{P}\mathring{\mathbf{x}}_p\left(\mathring{\mathbf{x}}_p^T\mathbf{w} \ - y_p\right) \tag{17}$$

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 (https://jermwatt.github.io/machine_learning_refined/notes/3_First_order_methods/3_2_First.html)) for the Least Squares cost function

$$\left(\sum_{p=1}^{P}\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T\right)\mathbf{w} \ = \sum_{p=1}^{P}\mathring{\mathbf{x}}_p y_p. \tag{18}$$

Example 3: Using gradient descent to minimize the Least Squares cost on our toy dataset

In the next Python cell minimize the Least Squares cost using the toy dataset presented in Example 2. 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. Here we employ the file `optimizers.py` which contains a short list of optimization algorithms we constructed explicitly in Chapters 2-4, including gradient descent and Newton's method.

```
In [8]:
```

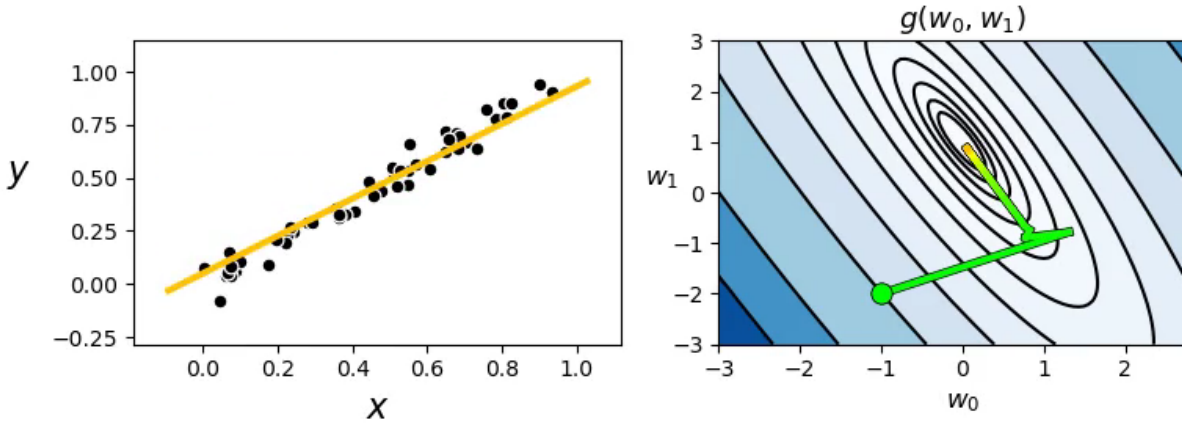
Now we animate the process of gradient descent run above. The contour of the cost function is shown in the right panel with each step plotted on top, colored from green at the start of the run to red at its end (green and red points mark the initialization and final weights reached by gradient descent). As you move the slider from left to right the gradient descent process animates, until completion when the slider is all the way to the right. Simultaneously, in the left panel the corresponding linear model given by the weights at each step of gradient descent is drawn. The linear model is colored to match the step of gradient descent, so near the beginning of the run the line is green whereas near the end it is plotted red.

As can be seen while pushing the slider to the right, as the minimum of the cost function is neared, the corresponding weights 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 minimum).

```
In [14]:
```

In [15]:

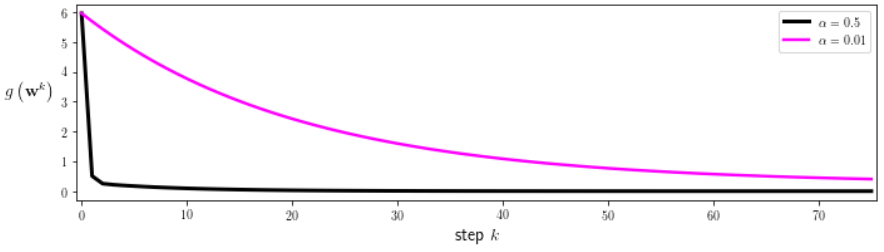
Out[15]:



Whenever we use a local optimization method like gradient descent we must properly tune the steplength parameter α. We did this for the run above by trying several fixed steplength values. Below we re-create those runs using $\alpha = 0.5$, $\alpha = 0.01$, showing the the cost function history plot for each steplength value choice. We can see from the plot that indeed the first steplength value works considerably better.

This illustrates why - in machine learning / deep 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 are learned.

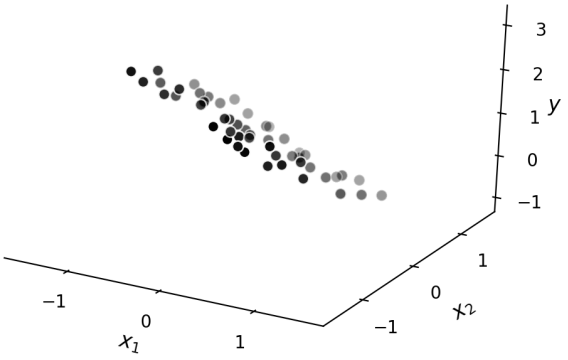
In [9]:



Example 4: An example with input dimension $N = 2$

In this example we look at another toy dataset with $N = 2$ inputs, which is plotted by the next Python cell. This dataset consists of 50 data points taken randomly from the hyperplane $y = 1 - x_1 - x_2$ with the addition of a small amount of random Gaussian noise to their y value.

In [18]:




```
/Users/Nurgetson/anaconda3/lib/python3.7/site-packages/matplotlib/cbook/deprecation.py:107: Matpl
otlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reu
ses the earlier instance.  In a future version, a new instance will always be created and returne
d.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a uniq
ue label to each axes instance.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```

In the next Python cell we minimize the Least Squares cost using the gradient descent, a constant steplength value $\alpha = 10^{-1}$ for 100 iterations beginning at the point

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

In [19]:

Now we animate this descent run. Since the linear model in this case has 3 parameters we cannot visualize each step on the contour / surface of the cost function itself, and thus must use a cost function plot (first introduced in our series on *mathematical optimization*) to keep visual track of the algorithm's progress.

In the left panel below we show the dataset, along with the hyperplane defined by $y = w_0 + x_1 w_1 + x_2 w_2$ whose weights are given at the current step in the gradient descent run. In the right panel the corresponding cost function value which plots the evaluation of each step up to the current one. Pushing the slider from left to right animates the run from start to finish - updating corresponding hyperplane in the left panel as well as cost function value in the right at each step (both of which simultaneously colored green at the start of the run, and gradually fade to red as the run ends).

As can be seen while pushing the slider to the right, as the minimum of the cost function is neared the corresponding weights 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 minimum).

In [20]:

In [21]:

Out[21]:

Example 5: Completely minimizing the Least Squares cost function using a single Newton step

In Example 6 of Section 4.4 (https://jermwatt.github.io/machine_learning_refined/notes/4_Second_order_methods/4_4_Newtons.html) we described how Newton's method perfectly minimizes any convex quadratic function in a single step since it is built on successively minimizing quadratic approximations to a given function. As we show formally in the next Subsection, the Least Squares cost function is a convex quadratic *for any dataset*. We show this by repeating the experiment in the previous example using a single Newton step.

In [26]:

In [27]:

Out[27]:

Technical details: proof of Least Squares convexity and Lipschitz constants

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 oepration) we have

$$\left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2 = \left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)\left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right) = y_p^2 - 2\mathring{\mathbf{x}}_p^T\mathbf{w}y_p + \mathring{\mathbf{x}}_p^T\mathbf{w}\mathring{\mathbf{x}}_p^T\mathbf{w} \tag{19}$$

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

$$= y_p^2 - 2\mathring{\mathbf{x}}_p^T\mathbf{w}y_p + \mathbf{w}^T\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T\mathbf{w} \tag{20}$$

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

$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\left(y_p^2 - 2\mathring{\mathbf{x}}_p^T\mathbf{w}y_p + \mathbf{w}^T\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}y_p^2 - \frac{2}{P}\sum_{p=1}^{P}y_p\mathring{\mathbf{x}}_p^T\mathbf{w} + \frac{1}{P}\sum_{p=1}^{P}\mathbf{w}^T\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T\mathbf{w} \tag{21}$$

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

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

we can write the Least Squares cost equivalently as

$$g\left(\mathbf{w}\right) = a \ + \mathbf{b}^T\mathbf{w} \ + \mathbf{w}^T\mathbf{C}\,\mathbf{w} \tag{22}$$

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 nonnegative.

Computation of the Lipschitz constant

Since we have just seen that the cost function is convex in order to compute a Lipschitz constant we can simply compute the largest eigenvalue of the matrix $\mathbf{C} = \frac{1}{P}\sum_{p=1}^{P}\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T$. This is precisely given as the *2-norm* of this matrix, squared

$$L = \|\mathbf{C}\|_2^2 \tag{23}$$

which one can compute via e.g., the *power method*.

For a larger but easier to compute Lipschitz constant one can use the trace of the matrix $\mathbf{C}$, since this equals the sum of all eigenvalues, which in this case must be larger than its maximum value since all eigenvalues are non-negative.