Linear Regression*

*Based on Chapter 5, “Machine Learning Refined: Foundations, Algorithms, and Applications”, Jeremy Watt, Reza Borhani, Aggelos K. Katsaggelos, 2nd Ed., Cambridge University Press, 2020

Least Squares Linear Regression

- In this lecture, we formally describe the problem of *linear regression.*
- That is, *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 the data.
  - Emphasize a particular point about the data under study.
  - Learn a model so that precise predictions can be made regarding output values in the future.

Notation and Modeling

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

$$(\boldsymbol{x}_1, y_1),\ (\boldsymbol{x}_2, y_2),\ \ldots,\ (\boldsymbol{x}_P, y_P)$$

- More compactly, this is $\{(\boldsymbol{x}_p, y_p)\}_{p=1}^{P}$, where $\boldsymbol{x}_p$ and y_p denote the p^{th} input and output, respectively.
- In simple instances, the input is scalar-valued.
- Then, the linear regression problem is geometrically equivalent of fitting a line to the associated scatter of data points in 2-dimensional space.
- However, each input $\boldsymbol{x}_p$ generally may be a column vector of length N:

$$\boldsymbol{x}_p = \begin{bmatrix} x_{1,p} & x_{2,p} & \cdots & x_{N,p} \end{bmatrix}^T$$

- Now, the linear regression problem is analogous to fitting a hyperplane to a scatter of points in $(N+1)$-dimensional space.

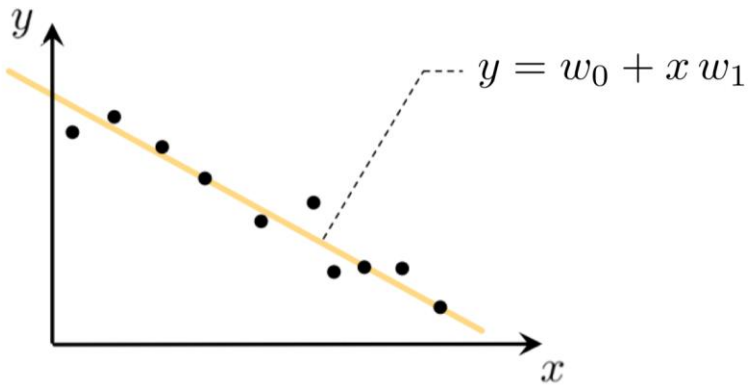
Notation and Modeling (cont'd)

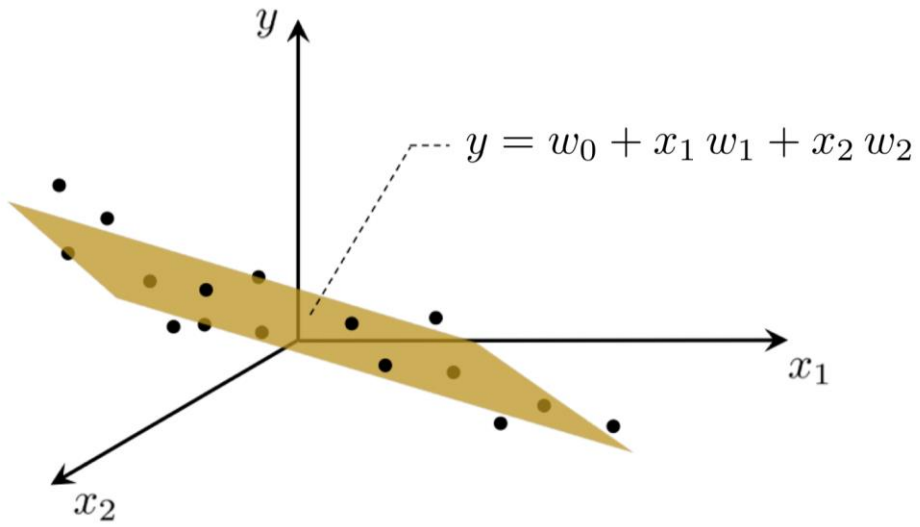
- With scalar input, we must determine appropriate vertical (y-) intercept w_0 and slope w_1 so that the following holds:

$$\hat{y}_p = w_0 + x_p w_1 \approx y_p, \qquad p = 1, \ldots, P.$$

- The approximately equal sign ($\approx$) is used *because we cannot be sure that all data lies completely on a line*.
- When dealing with N dimensional input, we have a bias w_0 and N associated slope weights w_i to tune appropriately:

$$\hat{y}_p = w_0 + x_{1,p} w_1 + x_{2,p} w_2 + \cdots + x_{N,p} w_N \approx y_p, \qquad p = 1, \ldots, P.$$





Notation and Modeling (cont'd)

- Note:
  - Each dimension of the *input* is referred to in the jargon of machine learning as a *feature* or *input feature*.
  - So, $w_{1,p}$, $w_{2,p}, \ldots, w_{N,p}$ are often referred to as the *feature-touching weights.*
  - The only weight *not* touching a feature is the bias w_0.
- For any N, we can write the previous equation more compactly as:

$$\dot{\boldsymbol{x}}_p^T \boldsymbol{w} \approx y_p, \quad p = 1, \ldots, P$$

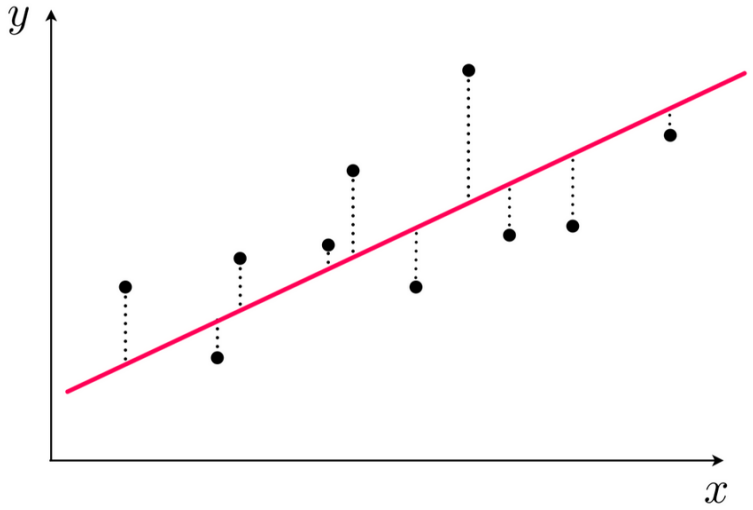
where:

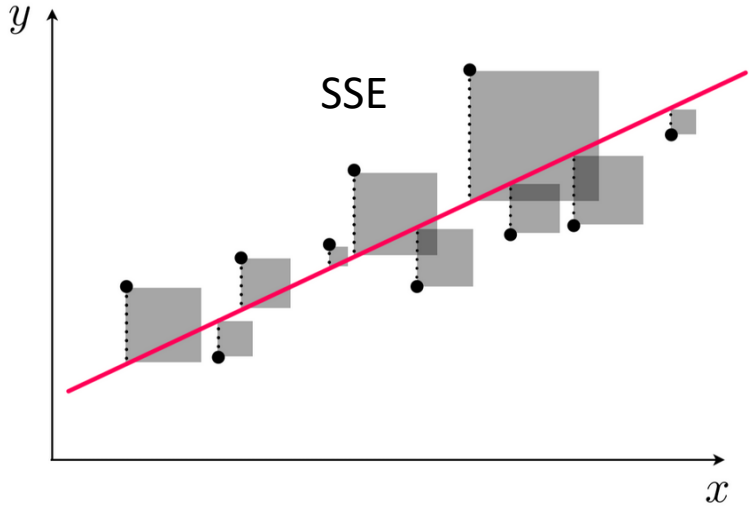
$$\boldsymbol{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N-1} \\ w_N \end{bmatrix} \qquad \dot{\boldsymbol{x}}_p = \begin{bmatrix} 1 \\ x_{1,p} \\ x_{2,p} \\ \vdots \\ x_{N,p} \end{bmatrix}$$

and notation $\dot{\boldsymbol{x}}$ denotes an input $\boldsymbol{x}$ with a 1 placed on top of it.

The Least Squares cost function

- To find the parameters of the hyperplane that best fits a regression dataset, we must use a cost function to measure how well a linear model fits it.
- A popular choice is to employ the *Least Squares cost function*.
- For a given set of parameters $\boldsymbol{w}$, this cost function computes the total squared error SSE between the associated hyperplane and the data, as illustrated in the figure below.
- Then, the best fitting hyperplane is the one whose parameters minimize this error.





The Least Squares cost function (cont'd)

- Where does this "Least Squares" cost come from?
- Remember: We want to find a weight vector $\boldsymbol{w}$ so that each of the P approximate equalities below holds as tightly as possible:

$$\hat{y}_p = \dot{\boldsymbol{x}}_p^T \boldsymbol{w} \approx y_p, \quad p = 1, \ldots, P.$$

- Equivalently, the *error (i.e., difference)*:

$$\epsilon_p = \hat{y}_p - y_p$$

  between $\dot{\boldsymbol{x}}_p^T \boldsymbol{w}$ (approximation) and y_p (true value) must be small.
- One natural way to measure the error between two quantities like this is to measure its *square* as:

$$g_p(\mathrm{w}) = \left(\dot{\boldsymbol{x}}_p^T \boldsymbol{w} - y_p\right)^2$$

- This means that *both negative and positive errors are treating equally*.

The Least Squares cost function (cont'd)

- This squared error $g_p(\cdot)$ is one example of a *point-wise cost* or *loss* that measures the error of a model on the point $\{\boldsymbol{x}_p, y_p\}$.
- Since we want all P such values to be small, we can take their average*, forming the *Least Squares* cost function for linear regression:

$$g(\boldsymbol{w}) = \frac{1}{P}\sum_{p=1}^{P} g_p(\boldsymbol{w}) = \underbrace{\frac{1}{P}\sum_{p=1}^{P} \overbrace{\left(\dot{\boldsymbol{x}}_p^T\boldsymbol{w} - y_p\right)^2}^{\text{loss}}}_{\text{SSE}}$$

- Observation: The Least Squares cost function is a function of both the weights $\boldsymbol{w}$ and the data.
- However, when we express the function in mathematical shorthand as $g(\boldsymbol{w})$ (as we do on the left-hand side above), we only show dependency on the weights $\boldsymbol{w}$.

*Another option followed: apply minimization to SSE, which is equivalent.

The Least Squares cost function (cont'd)

- Why do we do this?
  1. Foremost, for notational simplicity and convenience.
  2. For a given dataset, the weights $\boldsymbol{w}$ are the important input, *since this is what we need to tune to produce a good fit*.
  3. From an optimization perspective, *the dataset itself is considered fixed*.
- In overall:
  We want to tune our parameters $\boldsymbol{w}$ 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 so, the poorer we represent the given dataset using a linear model.
- In other words:
  We want to determine a value for weights $\boldsymbol{w}$ that *minimizes* $g(\boldsymbol{w})$, written formally as:

$$\text{minimize}_{\boldsymbol{w}} \frac{1}{P}\sum_{p=1}^{P}\left(\dot{\boldsymbol{x}}_p^T\boldsymbol{w} - y_p\right)^2$$

Least Squares method: Simple or bivariate regression

- Taking the partial derivatives:

$$\frac{\partial \sum(y_i-(w_0+w_1x_i))^2}{\partial w_0} = -2\sum\big(y_i-(w_0+w_1x_i)\big)=0$$

$$\frac{\partial \sum(y_i-(w_0+w_1x_i))^2}{\partial w_1} = -2\sum x_i\big(y_i-(w_0+w_1x_i)\big)=0$$

we find

(a) the slope w_1:

$$w_1=\frac{\sum(x_i-\mu_X)(y_i-\mu_Y)}{\sum(x_i-\mu_X)^2}=\frac{\sum x_iy_i-P\mu_X\mu_Y}{\sum x_i^2-P\mu_X^2}=\frac{S_{XY}}{S_{XX}}=\frac{Cov(X,Y)}{Var(X)}$$

where:

$$\mu_X=\frac{\sum x_i}{n} \quad \text{and} \quad \mu_Y=\frac{\sum y_i}{n}$$

Least Squares method: Simple or bivariate regression (cont'd)

and

(b) the y-intercept w_0:

$$w_0 = \mu_Y - w_1 \mu_X$$

- So:

$$\hat{y} = \mu_Y + w_1 (x - \mu_X)$$

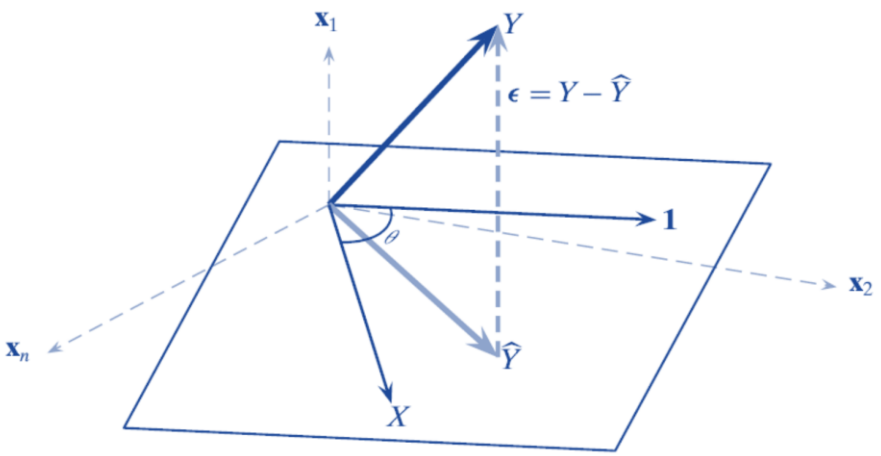
- Observation:
  The point (μ_X, μ_Y) *always belongs to the regression line.*

Least Squares method: Geometry of simple or bivariate regression

- Note that the predictions can be written as:

$$\widehat{\boldsymbol{Y}} = w_0\mathbf{1} + w_1\boldsymbol{X}$$

- This equation indicates that the predicted vector $\widehat{\boldsymbol{Y}}$ is a *linear combination* of $\mathbf{1}$ and $\boldsymbol{X}$.
- That is, it must lie in the column space spanned by $\mathbf{1}$ and $\boldsymbol{X}$, given as span($\{\mathbf{1}, \boldsymbol{X}\}$).
- The figure on the right depicts that the optimal $\widehat{\boldsymbol{Y}}$ minimizing the error $\boldsymbol{\epsilon}$ is the orthogonal projection of $\boldsymbol{Y}$ onto the subspace span($\{\mathbf{1}, \boldsymbol{X}\}$)
- The residual error vector $\boldsymbol{\epsilon}$ is *thus orthogonal* to *the subspace* span($\{\mathbf{1}, \boldsymbol{X}\}$), *and its squared length (or magnitude) equals the SSE value*.



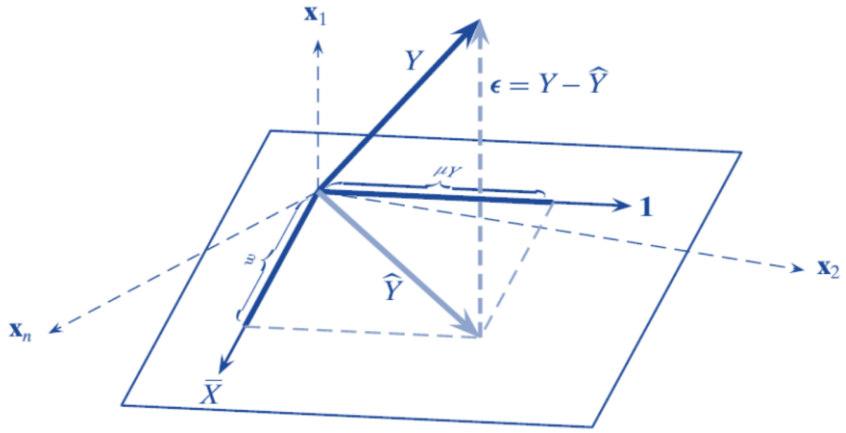
Least Squares method: Geometry of simple or bivariate regression (cont'd)

- We can create an orthogonal basis by decomposing $\boldsymbol{X}$ into a component along $\mathbf{1}$, and a component orthogonal to $\mathbf{1}$, as shown in the figure below, where:
  1. The projection of $\boldsymbol{X}$ onto $\mathbf{1}$ equals:

$$\left(\frac{\boldsymbol{X}^T\mathbf{1}}{\mathbf{1}^T\mathbf{1}}\right)\mathbf{1} = \left(\frac{\sum_{i=1}^{n} x_i}{n}\right)\mathbf{1} = \mu_{\boldsymbol{X}}\mathbf{1}$$

  2. The orthogonal (demeaned) component $\overline{\boldsymbol{X}}$ equals:

$$\boldsymbol{X} - \mu_{\boldsymbol{X}}\mathbf{1}$$



- Since $\mathbf{1}$ and $\overline{\boldsymbol{X}}$ form an orthogonal basis, we can obtain the predicted vector $\widehat{\boldsymbol{Y}}$ by projecting $\boldsymbol{Y}$ onto them.
- Summing up these two components, we have:

$$\widehat{\boldsymbol{Y}} = \mu_{\boldsymbol{Y}}\mathbf{1} + \left(\frac{\boldsymbol{Y}^T\overline{\boldsymbol{X}}}{\overline{\boldsymbol{X}}^T\overline{\boldsymbol{X}}}\right)\overline{\boldsymbol{X}} = \left(\mu_{\boldsymbol{Y}} - \left(\frac{\boldsymbol{Y}^T\overline{\boldsymbol{X}}}{\overline{\boldsymbol{X}}^T\overline{\boldsymbol{X}}}\right)\mu_{\boldsymbol{X}}\right)\mathbf{1} + \left(\frac{\boldsymbol{Y}^T\overline{\boldsymbol{X}}}{\overline{\boldsymbol{X}}^T\overline{\boldsymbol{X}}}\right)\boldsymbol{X}$$

- Since $\widehat{\boldsymbol{Y}} = w_0\mathbf{1} + w_1\boldsymbol{X}$, equating the two expressions, we arrive at the same w_0, w_1 values.

Least Squares method: Multiple regression

- Let $\dot{\boldsymbol{X}}$ be the matrix, where:
  1. The first column $\dot{\boldsymbol{X}}_0$ is equal to $[1 \;\; 1 \cdots \;\; 1]^T$.
  2. The other columns $\dot{\boldsymbol{X}}_j$ contain the data of the jth variable (factor).
- $\dot{\boldsymbol{X}}$ is also known as the "*model matrix*".
- Setting:

$$\nabla_{\boldsymbol{w}} \left(\left\| \boldsymbol{Y} - \dot{\boldsymbol{X}}\boldsymbol{w} \right\|^2 \right) = -2\dot{\boldsymbol{X}}^T \left(\boldsymbol{Y} - \dot{\boldsymbol{X}}\boldsymbol{w} \right) = \boldsymbol{0}$$

  we have:

$$\dot{\boldsymbol{X}}^T \boldsymbol{Y} = \dot{\boldsymbol{X}}^T \dot{\boldsymbol{X}} \boldsymbol{w}$$

- This expression defines a set of equations known as the *normal equations.*
- Given that $\dot{\boldsymbol{X}}$ has linear independent columns (and so does the Gram matrix $\dot{\boldsymbol{X}}^T \dot{\boldsymbol{X}}$), we have:

$$\boldsymbol{w} = \underbrace{\left(\dot{\boldsymbol{X}}^T \dot{\boldsymbol{X}} \right)^{-1} \dot{\boldsymbol{X}}^T}_{\text{Pseudoinverse of } \dot{\boldsymbol{X}}} \boldsymbol{Y} \qquad \text{and} \qquad \hat{\boldsymbol{Y}} = \dot{\boldsymbol{X}}\boldsymbol{w} = \underbrace{\dot{\boldsymbol{X}} \left(\dot{\boldsymbol{X}}^T \dot{\boldsymbol{X}} \right)^{-1} \dot{\boldsymbol{X}}^T}_{\boldsymbol{H}} \boldsymbol{Y}$$

- $\boldsymbol{H}$ is called the "*hat matrix*", since it 'puts' a hat ($\hat{}$) on $\boldsymbol{Y}$.

Least Squares method: Geometry of multiple regression

- As in the simple case:
  - $\widehat{\boldsymbol{Y}}$ is the orthogonal projection of $\boldsymbol{Y}$ onto the subspace spanned by the column vectors $\dot{\boldsymbol{X}}_j$.
  - The error $\boldsymbol{\epsilon} = \boldsymbol{Y} - \dot{\boldsymbol{X}}\boldsymbol{w}$ is orthogonal to this subspace and so to each $\dot{\boldsymbol{X}}_j$.
- Thus:

$$\dot{\boldsymbol{X}}_j^T \boldsymbol{\epsilon} = 0, \quad \forall j$$

- From the above equations, we can derive again the *normal equations*.

Least Squares method: Geometry of multiple regression (cont'd)

- Since $\dot{\boldsymbol{X}}_j$ are linear independent, we can use the *Gram–Schmidt* orthogonalization to construct a set of orthogonal basis vectors $\boldsymbol{U}_0, \boldsymbol{U}_1, \ldots, \boldsymbol{U}_d$:

$$\begin{aligned}
\boldsymbol{U}_0 &= \dot{\boldsymbol{X}}_0 \\
\boldsymbol{U}_1 &= \dot{\boldsymbol{X}}_1 - p_{01}\boldsymbol{U}_0 \\
\boldsymbol{U}_2 &= \dot{\boldsymbol{X}}_2 - p_{02}\boldsymbol{U}_0 - p_{12}\boldsymbol{U}_1 \\
&\vdots \\
\boldsymbol{U}_d &= \dot{\boldsymbol{X}}_d - p_{0d}\boldsymbol{U}_{\boldsymbol{0}} - p_{1d}\boldsymbol{U}_1 - \cdots - p_{d-1d}\boldsymbol{U}_{d-1}
\end{aligned}$$

  where:

  $p_{ij} = \frac{\dot{\boldsymbol{X}}_j^T \boldsymbol{U}_i}{\boldsymbol{U}_i^T \boldsymbol{U}_i}$ is the scalar projection of attribute $\dot{\boldsymbol{X}}_j$ onto the basis vector $\boldsymbol{U}_i$.

- From the above expression, with rearrangement, we can write $\dot{\boldsymbol{X}}$, also known as *QR-factorization*, as:

$$\dot{\boldsymbol{X}} = \boldsymbol{Q}\boldsymbol{R},$$

  where $\boldsymbol{Q}$ is the orthogonal matrix of orthogonal basis vectors $\boldsymbol{U}_0, \boldsymbol{U}_1, \ldots, \boldsymbol{U}_d$ (note: $\boldsymbol{Q}^T\boldsymbol{Q} = \text{diag}(\|\boldsymbol{U}_i\|^2) = \boldsymbol{\Delta}$) and $\boldsymbol{R}$ the *upper triangular* matrix, with 1 in the main diagonal and the scalar projections p_{ij} in the rest cells.

- Then:

$$\dot{\boldsymbol{X}}^T\boldsymbol{Y} = \dot{\boldsymbol{X}}^T\dot{\boldsymbol{X}}\boldsymbol{w} \Rightarrow \boldsymbol{R}^T\boldsymbol{Q}^T\boldsymbol{Y} = \boldsymbol{R}^T\boldsymbol{Q}^T\boldsymbol{Q}\boldsymbol{R}\boldsymbol{w} \text{ or } \boldsymbol{R}^T\boldsymbol{Q}^T\boldsymbol{Y} = \boldsymbol{R}^T\boldsymbol{\Delta}\boldsymbol{R}\boldsymbol{w} \text{ or } \boldsymbol{w} = \boldsymbol{R}^{-1}\boldsymbol{\Delta}^{-1}\boldsymbol{Q}^T\boldsymbol{Y}$$

Least Squares method: Geometry of multiple regression (cont'd)

- The above discussion can be summarized as follows:

MULTIPLE-REGRESSION $(\boldsymbol{X}, \boldsymbol{Y})$

1. $\dot{\boldsymbol{X}} = [\mathbf{1}\ \boldsymbol{X}]$ // augmented data with $\boldsymbol{X}_0 = \mathbf{1} \in \mathbb{R}^n$
2. $\{\boldsymbol{Q}, \boldsymbol{R}\} = \text{QR–factorization}(\dot{\boldsymbol{X}})$ // $\boldsymbol{Q} = [\boldsymbol{U}_0\ \boldsymbol{U}_1 \cdots \boldsymbol{U}_d]$
3. $\boldsymbol{\Delta}^{-1} = \begin{bmatrix} \frac{1}{\|\boldsymbol{U}_0\|^2} & 0 & \cdots & 0 \\ 0 & \frac{1}{\|\boldsymbol{U}_1\|^2} & \cdots & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \cdots & \frac{1}{\|\boldsymbol{U}_d\|^2} \end{bmatrix}$ // reciprocal squared norm
4. Calculate $\boldsymbol{\Delta}^{-1}\boldsymbol{Q}^T\boldsymbol{Y}$
5. Solve for $\boldsymbol{w}$ by back-substitution, using the fact that $\boldsymbol{R}\boldsymbol{w} = \boldsymbol{\Delta}^{-1}\boldsymbol{Q}^T\boldsymbol{Y}$
6. $\hat{\boldsymbol{Y}} = \boldsymbol{Q}\boldsymbol{\Delta}^{-1}\boldsymbol{Q}^T\boldsymbol{Y}$ // $\hat{\boldsymbol{Y}} = \boldsymbol{Q}\boldsymbol{R}\boldsymbol{w}$

Least Squares method: Gradient descent

- Instead of using the QR-factorization approach to exactly solve the multiple regression problem, we can also employ the *simpler* gradient descent algorithm.
- Since:

$$\nabla_{\boldsymbol{w}}\left(\frac{1}{p}\left\|\boldsymbol{Y}-\dot{\boldsymbol{X}}\boldsymbol{w}\right\|^{2}\right)=-\frac{2}{p}\dot{\boldsymbol{X}}^{T}\left(\boldsymbol{Y}-\dot{\boldsymbol{X}}\boldsymbol{w}\right)$$

we have the following update rule:

$$\boldsymbol{w}^{k+1}=\boldsymbol{w}^{k}-\alpha\nabla_{\boldsymbol{w}}=\boldsymbol{w}^{k}+\alpha\frac{2}{p}\underbrace{\dot{\boldsymbol{X}}^{T}\left(\boldsymbol{Y}-\dot{\boldsymbol{X}}\boldsymbol{w}^{k}\right)}_{\widetilde{\nabla}_{\boldsymbol{w}}}=\boldsymbol{w}^{k}+\alpha'\widetilde{\nabla}_{\boldsymbol{w}}$$

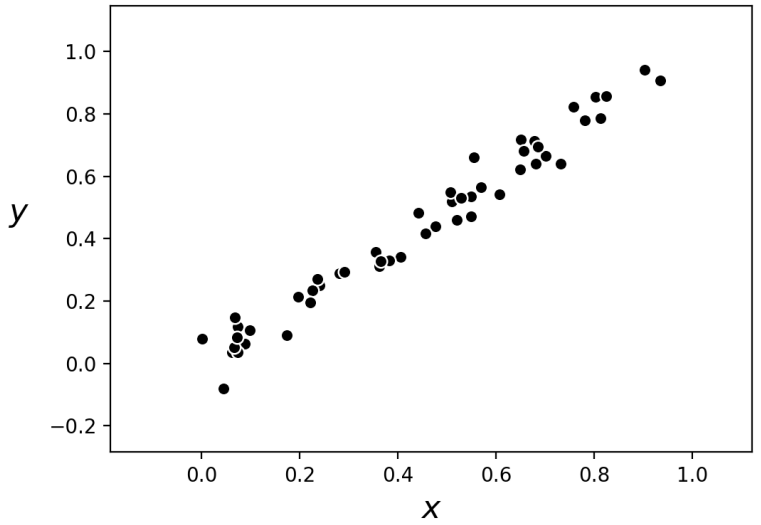
Minimization of the Least Squares cost function

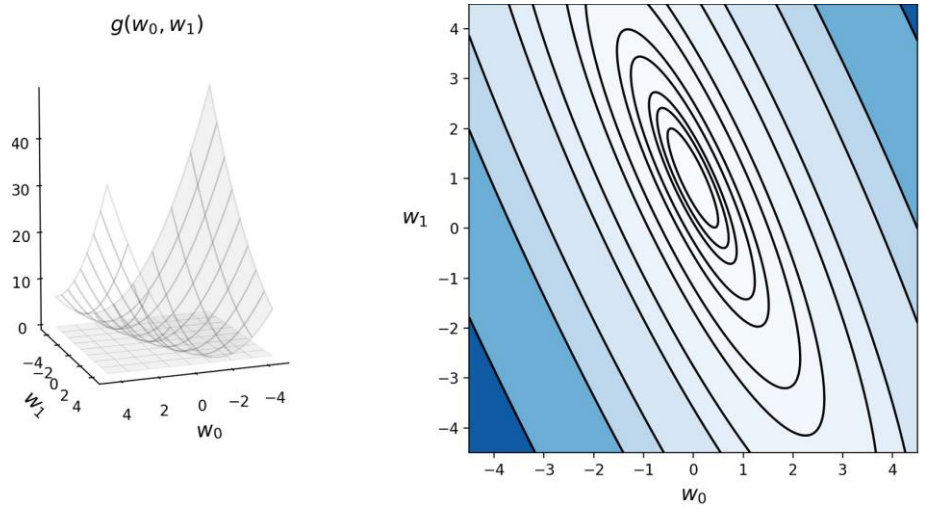
- Remember:
  Determining the overall shape of a function (i.e., the function is convex)…
  ...helps determine the appropriate optimization method(s) we can apply to efficiently determine the ideal parameters.
- In the case of the Least Squares cost function for linear regression, it is easy to check that *it is always a convex quadratic function 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 the following example demonstrates.

* See Section 5.9 of the textbook.

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

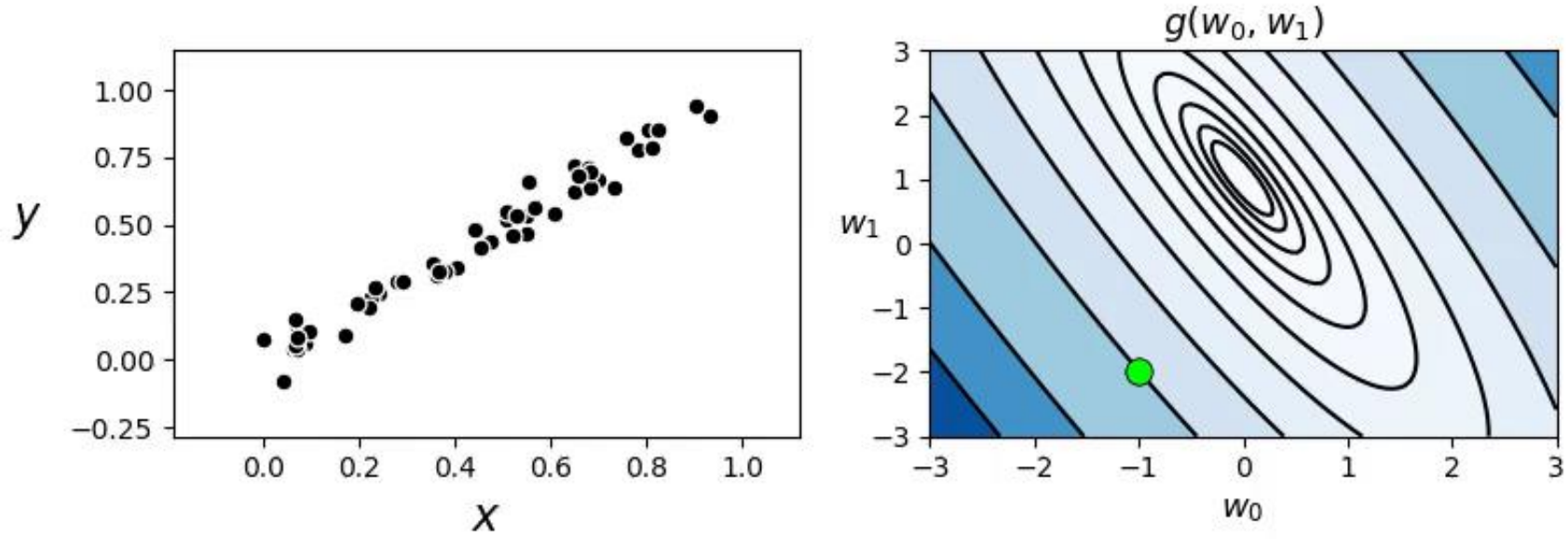
- In this example, we plot the contour and surface plot for the Least Squares cost function for linear regression for a toy dataset.
- This toy dataset consists of 50 points randomly selected from the line $y = x$, adding a small amount of Gaussian noise to each.
- The contour plot and corresponding surface generated by the Least Squares cost function using this data are shown in the bottom panel of the figure.





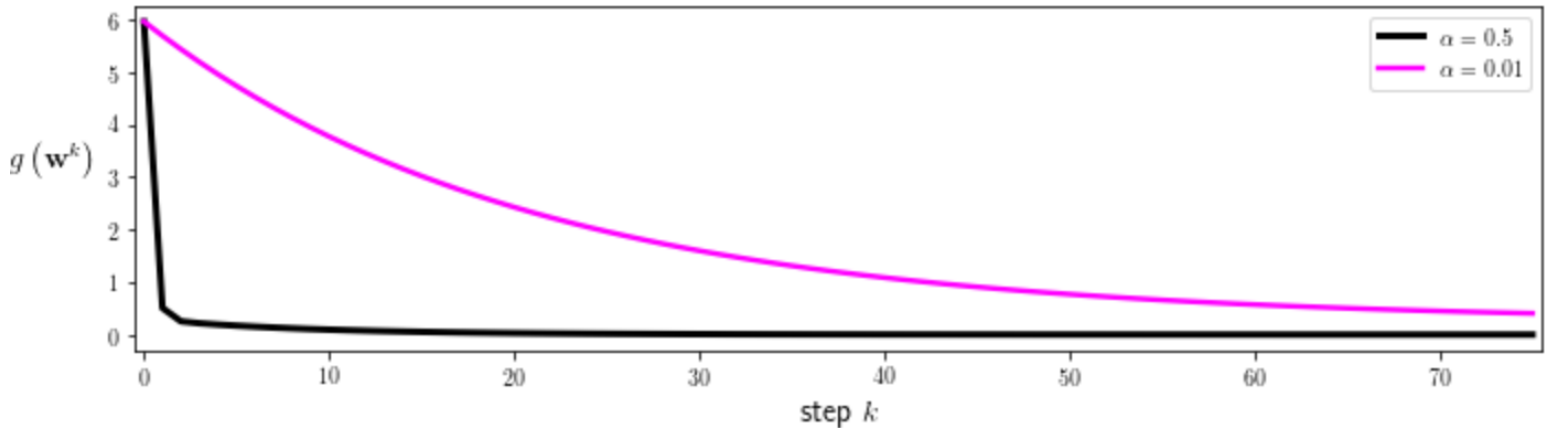
Example. Using gradient descent to minimize the Least Squares cost on a toy dataset

- Below, the minimization of the Least Squares cost is animated with the toy dataset presented in the previous example.
- More specifically, gradient descent is used, employing a fixed steplength value $\alpha = 0.5$ for all 75 steps, until approximately reaching the minimum of the function.



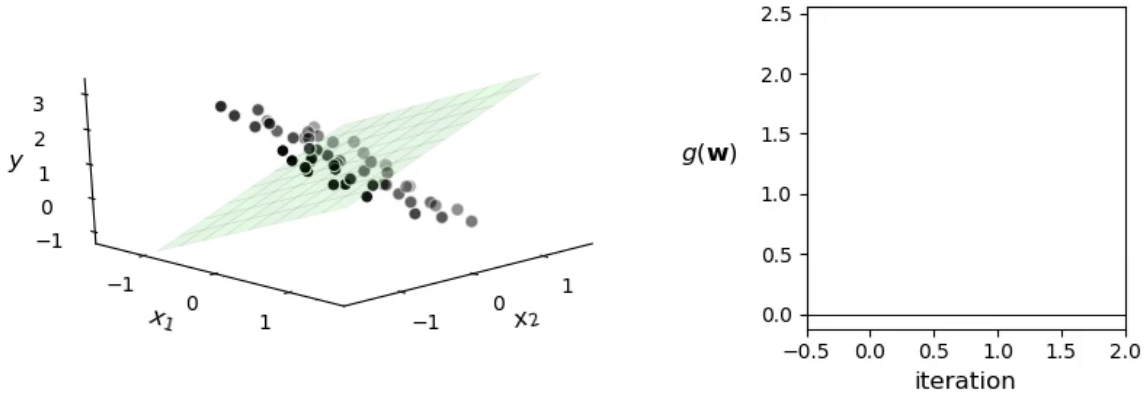
Example. Using gradient descent to minimize the Least Squares cost on a toy dataset (cont'd)

- Remember: Whenever we use a local optimization method like gradient descent, we must properly tune the steplength parameter α.
- This was done for the previous run, with several fixed steplength values.
- Below, those runs are re-created, using $\alpha = 0.5$, $\alpha = 0.01$ and showing 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.



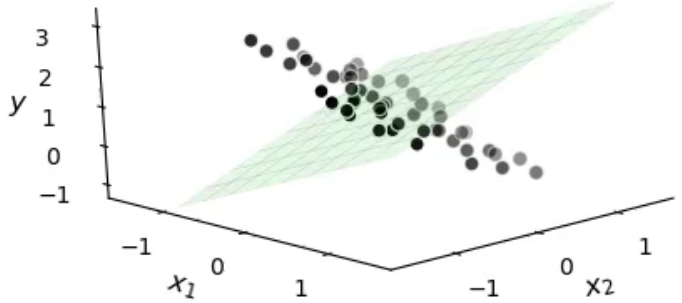
Example. Completely minimizing the Least Squares cost function using a single Newton step

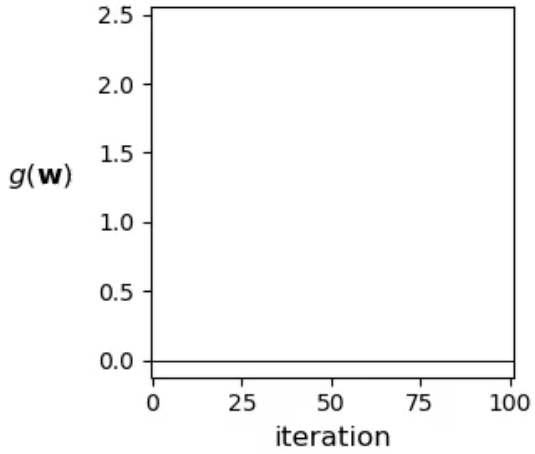
- In the optional reading of the previous topic, we described how Newton's method perfectly minimizes any convex quadratic function in a single step.
- As mentioned, one can easily show that the Least Squares cost function is *a convex quadratic* function *for any dataset*.
- Thus, a *single Newton step* will completely minimize it, *regardless of the dataset used.*
- We animate this for a toy dataset with $N = 2$ inputs.



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

- The same minimization using gradient descent





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

$$\text{model}\left(\boldsymbol{x}_p, \boldsymbol{w}\right) = \dot{\boldsymbol{x}}_p^T \boldsymbol{w}.$$

- If we were to go back then, and use this modeling notation, we could equally well express our ideal settings of the weights our original approximation as

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

and, likewise, our Least Squares cost function itself as

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

Implementing the Least Squares cost in `Python` (cont'd)

- To write this in `Python` we un-ravel the bias from our weights, using slicing and express the model computation as `a = w[0] + np.dot(x_p.T,w[1:])`
- The linear combination $\boldsymbol{x}_p^T\boldsymbol{w}_{[1:]}$ - implemented using `np.dot` - is very efficient (far more than an explicit `for` loop in `Python`).

```
# 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
```

- Next, in writing out the Least Squares function, we could explicitly loop over our P points, one at-a-time.

```
# 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)
```

Implementing the Least Squares cost in `Python` (cont'd)

- However, when such `for` loops can be equivalently written using `numpy` operations, this is typically far more efficient.
- *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*).
- 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`.
- In fact, Python code, employing heavy use of `numpy` functions, can often execute almost as fast a raw `C` implementation itself.

Implementing the Least Squares cost in `Python` (cont'd)

- Below, we show compact implementations of both our model and Least Squares cost in `Python` leveraging `numpy`.

```
# 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 that, for simplicity, we write 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, 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.

Least Absolute Deviations

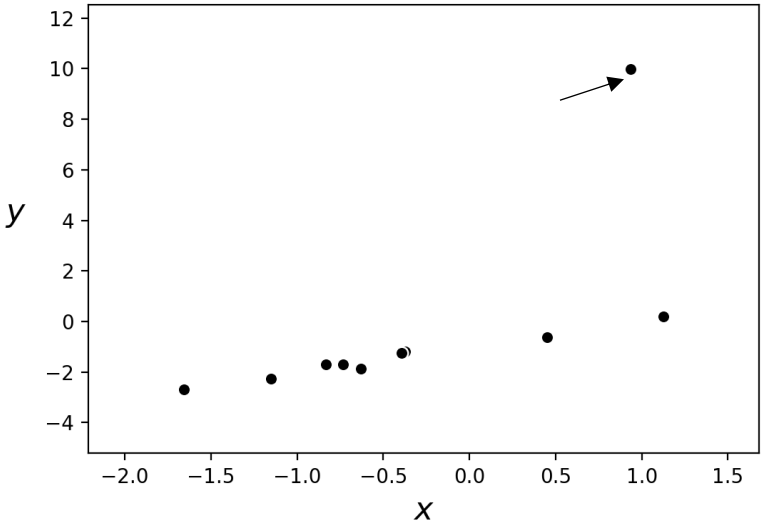
- In the sequel, we consider a slight twist on the derivation of the Least Squares cost function that leads to an alternative cost for linear regression.
- This cost is called *Least Absolute Deviations*.
- This alternative cost function is much more *robust* to outliers in a dataset than the original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

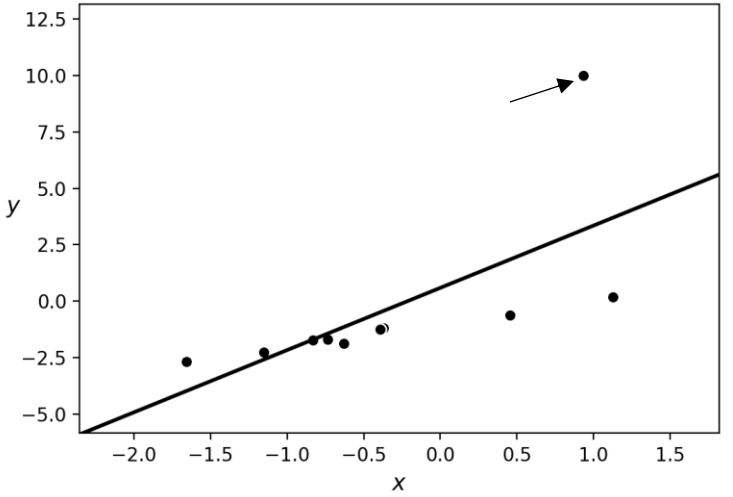
The susceptibility of the Least Squares cost to outliers

- One downside of using the squared error in the Least Squares cost is that *squaring the error increases the importance of larger errors*.
- Specifically, *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.
- This is often 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 the following example.

Example. Least Squares overfits to outliers

- In this example, we use the dataset plotted on the right.
- This dataset can largely be represented by a proper linear model, except for a single large outlier.
- This example shows how the Least Squares cost function for linear regression tends to create linear models that *overfit* to outliers.
- First, the parameters of a linear regressor to this dataset are tuned by minimizing the Least squares cost via gradient descent.
- Next, the linear model associated with those weights achieving the smallest cost function values is plotted.
- This fit, shown in black, does not fit most of the data points well, bending upward.
- This occurred with the aim of minimizing the large squared error on the singleton outlier point.





Replacing Squared Error with Absolute Error

- How can we make our linear regression framework more robust to outliers?
- An alternative to *squared* error for our point-wise cost is the *absolute error*, defined as:

$$g_p(\boldsymbol{w}) = \left|\dot{\boldsymbol{x}}_p^T \boldsymbol{w} - y_p\right|, \quad p = 1, \ldots, P.$$

- By using absolute error, *we still treat negative and positive errors equally*.
- However, *we do not exaggerate the importance of large errors greater than 1.*
- Why?
- Because *we do not square them.*

Replacing Squared Error with Absolute Error (cont'd)

- If we form the average of these absolute error point-wise costs (losses), we have the so-called *Least Absolute Deviations* cost function:

$$g(\boldsymbol{w}) = \frac{1}{P}\sum_{p=1}^{P} g_p(\boldsymbol{w}) = \frac{1}{P}\sum_{p=1}^{P} \left|\dot{\boldsymbol{x}}_p^T \boldsymbol{w} - y_p\right|$$

- This cost function is also *always convex, regardless of the input dataset* (like Least Squares).
- But *we cannot use Newton's method to minimize this in a single step, since it doesn't have a useful second derivative as it is zero almost everywhere.*
- Even though it is not differentiable everywhere, we can use gradient descent to minimize it.
- Technically speaking when doing this, we are using 'sub-gradient* descent', but it works all the same.

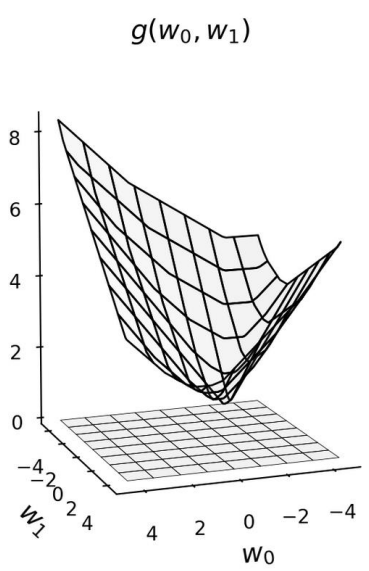
*$\boldsymbol{v}$: $f(\boldsymbol{x}) - f(\boldsymbol{x_0}) \geq \boldsymbol{v} \cdot (\boldsymbol{x} - \boldsymbol{x_0})$

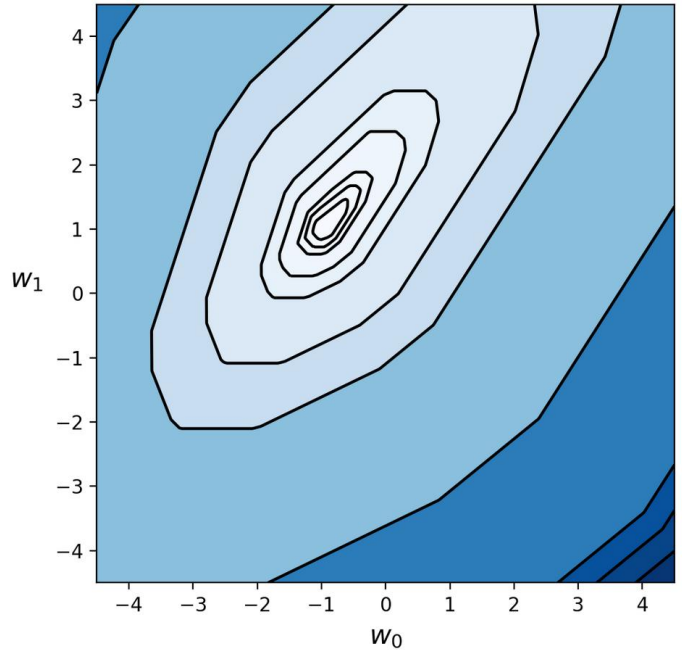
Least absolute deviations versus Least Squares

- In implementing the cost in `Python`, we can employ the model function we used with our Least Squares implementation shown previously.
- All we then need to do is slightly alter the cost function itself to get our desired implementation, as shown below.

```
# a compact least absolute deviations cost function
def least_absolute_deviations(w):
    cost = np.sum(np.abs(model(x,w) - y))
    return cost/float(np.size(y))
```

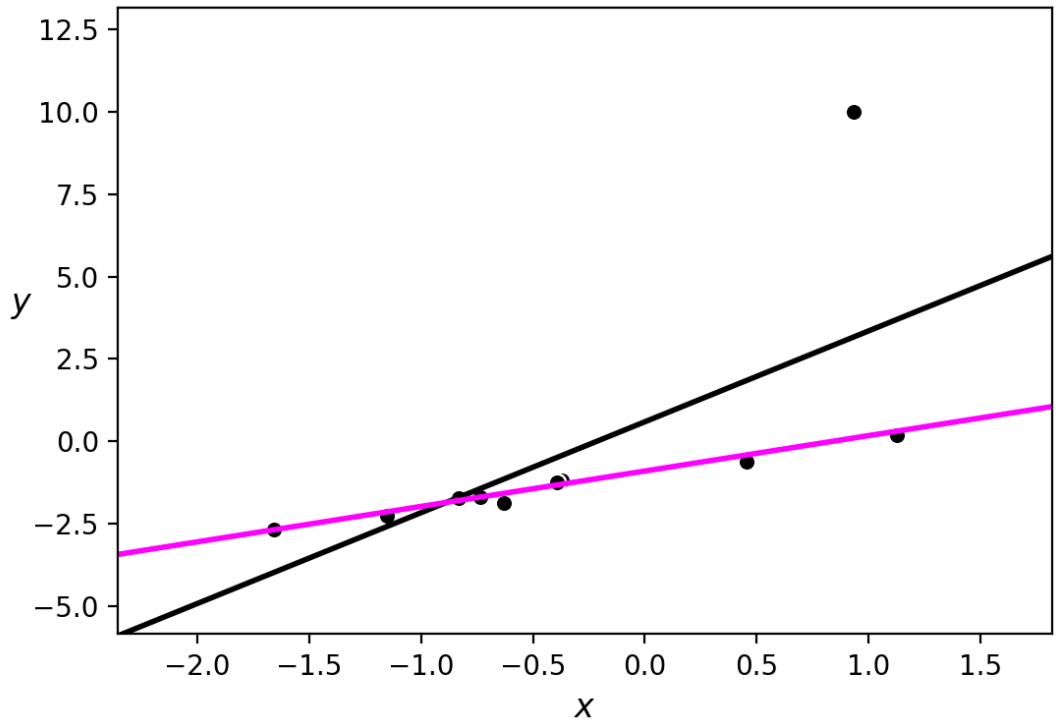
- On the right, we plot the surface/ contour plot of this cost function using the previously shown dataset - indeed it is convex.





Example. Least absolute deviations versus Least Squares

- Let us plot and compare the best fit found via gradient descent for both Least Squares and Least Absolute Deviations cost functions on the previous dataset.
- The Least Squares fit is shown in black, while the Least Absolute Deviation fit is depicted in magenta.
- The latter fit is considerably better, since it does not exaggerate the large error produced by the single outlier.



Regression Quality Metrics

- Next, we discuss how to make predictions using a trained model.
- We also describe simple metrics for judging the quality of a trained regression model.

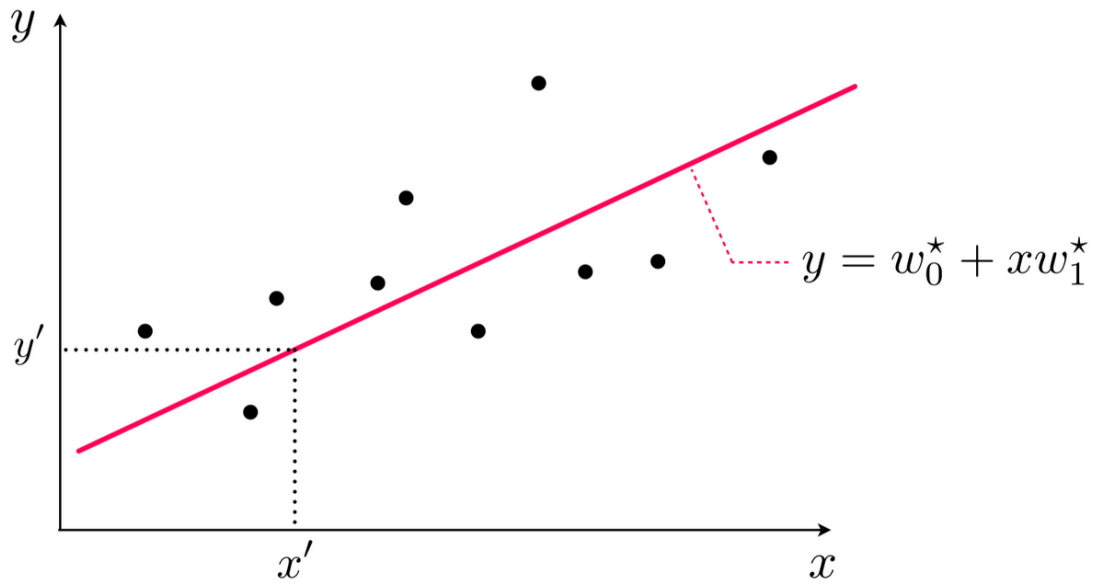
Making predictions using a trained model

- If we denote the optimal set of weights found by minimizing a regression cost function by $\boldsymbol{w}^\star$, then we can write our fully tuned linear model as:

$$\text{model}(\boldsymbol{x}, \boldsymbol{w}^\star) = \dot{\boldsymbol{x}}^T \boldsymbol{w}^\star = w_0^\star + x_1 w_1^\star + x_2 w_2^\star + \cdots + x_N w_N^\star.$$

- Given an input $\boldsymbol{x}'$, whether one from our training dataset or a new input, we predict its output y' by passing it along with our trained weights into our model as:

$$\text{model}(\boldsymbol{x}', \boldsymbol{w}^\star) = y'$$



Judging the quality of a trained model

- Once we have successfully minimized the adopted cost function for linear regression, it is an easy matter to determine the quality of our regression model.
- We must simply evaluate a cost (error) function using our optimal weights.
- For example, we can evaluate the quality of this trained model using a Least Squares cost.
- This is especially natural to use, when we employ this cost in training.
- To do this, we plug in our trained model and dataset into the Least Squares cost, giving the *Mean Squared Error* (or MSE for short) of our trained model:

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

- The name for this quality measurement describes precisely what the Least Squares cost computes: the *average* or *mean squared error*.

Judging the quality of a trained model (cont'd)

- In the same way, we can employ the Least Absolute Deviations cost to determine the quality of our trained model.
- Plugging in our trained model and dataset into this cost computes the *Mean Absolute Deviations* (or MAD for short).
- This is precisely what this cost function computes:

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

Judging the quality of a trained model (cont'd)

- These two measurements differ in precisely the ways we have seen their respective cost functions differ:
  the MSE measure is far more sensitive to outliers.
- Which measure one employs in practice can therefore depend on personal and/or occupational preference, or the nature of a problem at hand.
- Of course, in general, the *lower* one can make a quality measure, 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.

Weighted Regression

- Regression cost functions are *summable losses over individual points.*
- Hence, we can weight individual points in order to emphasize or de-emphasize their importance to a regression model.
- This is called *weighted regression*.

Dealing with duplicates

- Imagine we have a linear regression dataset $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_P, y_P)$ that contains multiple copies of the same point.
- This can appear in a variety of contexts including:
  - Experimental data (e.g., in physics, medicine, etc.): If a repeated experiment produces the same result.
  - Metadata-type datasets (e.g., census, customer databases): Due to necessary/ helpful pre-processing that quantizes (bins) input features to make human-in-the-loop analysis of the data and/or modeling easier.
- In such instances, *'duplicate' datapoints should not be thrown away*:

  They accurately represent the true phenomenon under study.
- If we examine any regression cost function over such a dataset (i.e., one with repeated entries), we can see that it naturally *collapses* into a weighted version itself.

Dealing with duplicates (cont'd)

- For example, let us examine the Least Squares cost.
- Suppose that our first two datapoints $(\boldsymbol{x}_1,\, y_1)$ and $(\boldsymbol{x}_2,\, y_2)$ are *identical*.
- In this instance, using our model notation, the first two summands of our cost function (in the first two datapoints) can be combined since they will always take on the same value:

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

$$= \frac{1}{P}\left(\text{model}(\boldsymbol{x}_1, \boldsymbol{w}) - y_1\right)^2 + \frac{1}{P}\left(\text{model}(\boldsymbol{x}_2, \boldsymbol{w}) - y_2\right)^2 + \frac{1}{P}\sum_{p=3}^{P}\left(\text{model}\left(\boldsymbol{x}_p, \boldsymbol{w}\right) - y_p\right)^2$$

$$= \frac{1}{P}\,2\left(\text{model}(\boldsymbol{x}_1, \boldsymbol{w}) - y_1\right)^2 + \frac{1}{P}\sum_{p=3}^{P}\left(\text{model}\left(\boldsymbol{x}_p, \boldsymbol{w}\right) - y_p\right)^2$$

- Here, we can see that the cost function naturally collapses, so that a repeated point in a dataset is represented *in the cost function* by a single weighted summand.

Dealing with duplicates (cont'd)

- Of course, this holds more generally as well.
- If a dataset has any number of identical points, then we can collapse the summands of the regression cost for each set of identical points just as we have seen previously.
- In general, this leads to the notion that each term in a regression cost can be *weighted* to reflect repeated points.

Dealing with duplicates (cont'd)

- We can write such a *weighted regression* Least Squares as:

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

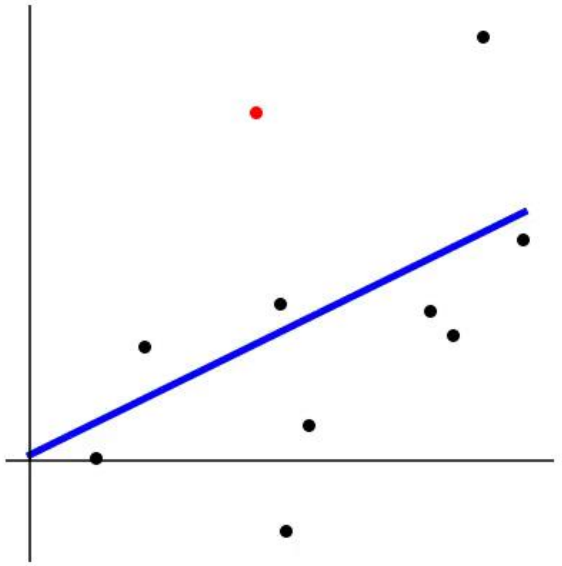
  where $\beta_1, \beta_2, \ldots, \beta_P$ are *point-wise* weights.
- That is, a unique point $(\boldsymbol{x}_p, y_p)$ in the dataset has weight $\beta_p = 1$.
- If a point is repeated R times in the dataset, then one instance of it will have weight $\beta_p = R$, while the others will have weight $\beta_p = 0$.
- Since these weights are fixed (i.e., they are *not* parameters that need to be tuned, like $\boldsymbol{w}$), *we can minimize a weighted regression cost precisely as we would any other*.

Weighting points by confidence

- Weighted regression can also be employed when we want to weight each point based on our *confidence on the trustworthiness* of each datapoint.
- That assumes knowledge of the process generating the dataset.
- For example, if our dataset came in two batches, one from a trustworthy 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.
- We can do this very easily, using precisely the weighted regression paradigm introduced above.
- So, now we set the weights $\beta_1, \beta_2, \dots, \beta_P$ 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 closer to 1.
- The less trustworthy a point is the smaller we set β_p in the range $[0,1]$, where the zero value implies that *we do not trust the point at all*.
- Making these weight selections, effectively we determine *how important each datapoint is* in the training of the model.

Weighting points by confidence (cont'd)

- Below, the result of increasing the confidence/weight β_p on a *single point* in a toy dataset and how this effects a fully trained regression model on a toy linear regression dataset is animated.
- This single point is colored red, and its diameter is shown increasing as the corresponding weight β_p increases.
- With each weighting, a weighted Least Squares cost is completely minimized over the entire dataset, and the resulting line fit to data is shown in blue.
- Observe: The higher the weighting of this single point, the more a linear regressor is incentivized to fit it.
- If the weight is increased enough, the fully trained regression model naturally starts fitting to this single datapoint alone, *disregarding all other points*.



Multi-Output Regression

- Thus far we have assumed that datapoints for linear regression consist of N dimensional vector valued inputs and scalar-valued outputs.
- So, a prototypical datapoint takes the form $(\boldsymbol{x}_p, y_p)$, where $\boldsymbol{x}_p$ is an N dimensional input vector and a y_p is a scalar output.
- While this configuration covers most of the regression cases one may well encounter in practice,…
  …it is possible to perform (linear) regression where both input and output are vector-valued.
- This is often called *multiple-output regression*.

Notation and Modeling

- Suppose now that the output of a regression dataset is *vector-valued*.
- That is, our datapoints $(\boldsymbol{x}_1, \boldsymbol{y}_1)$, $(\boldsymbol{x}_2, \boldsymbol{y}_2)$, …, $(\boldsymbol{x}_P, \boldsymbol{y}_P)$ have both *vector-valued* input and output.
- The pth point $(\boldsymbol{x}_p, \boldsymbol{y}_p)$ has N dimensional input $\boldsymbol{x}_p$ and associated C dimensional output $\boldsymbol{y}_p$.
- While, in principle, we can treat $\boldsymbol{y}_p$ as a $C \times 1$ column vector,…
  …to keep the formulas that follow looking similar to what we have already seen in the scalar case,…
  …the input is treated as a $N \times 1$ column vector and the output as a $1 \times C$ row vector.

Notation and Modeling (cont'd)

- So, if we assume that a linear relationship holds between the input $\boldsymbol{x}_p$ and just the cth dimension of the output $y_{c,p}$, we are using precisely the sort of regression framework discussed thus far.
- So, we can write:

$$\dot{\boldsymbol{x}}_p^T \boldsymbol{w}_c \approx y_{c,p}, \qquad p = 1, \ldots, P$$

where $\boldsymbol{w}_c$ is a vector of weights and $\dot{\boldsymbol{x}}_p$ is the vector formed by stacking 1 on top of $\boldsymbol{x}_p$.

- 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 $\boldsymbol{w}_c$ into the cth column of an $(N + 1) \times C$ weight matrix $\boldsymbol{W}$ like this:

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

Notation and Modeling (cont'd)

- Then, the entire set of C linear models can be written compactly as $1 \times C$ vector-matrix product:

$$\dot{\boldsymbol{x}}_p^T \boldsymbol{W} = [\dot{\boldsymbol{x}}_p^T \boldsymbol{w}_0 \;\; \dot{\boldsymbol{x}}_p^T \boldsymbol{w}_1 \;\cdots\; \dot{\boldsymbol{x}}_p^T \boldsymbol{w}_{c-1}]$$

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

$$\hat{\boldsymbol{y}}_p = \dot{\boldsymbol{x}}_p^T \boldsymbol{W} \approx \boldsymbol{y}_p, \qquad p = 1, \ldots, P$$

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.
- For example, to derive a Least Squares cost function, we begin by taking the difference of both sides of $\boldsymbol{x}_p^T \boldsymbol{W} \approx \boldsymbol{y}_p$.
- However, now the error $\boldsymbol{\epsilon}_p = \dot{\boldsymbol{x}}_p^T \boldsymbol{W} - \boldsymbol{y}_p$ associated with the pth point has C values.
- The Least Squares cost function in this case is then the *average squared* ℓ_2 *norm* of each point's error:

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

- Likewise, the Least Absolute Deviations cost for our present case, using the ℓ_1 *norm* of each point's error, takes the form:

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

Cost Functions (cont'd)

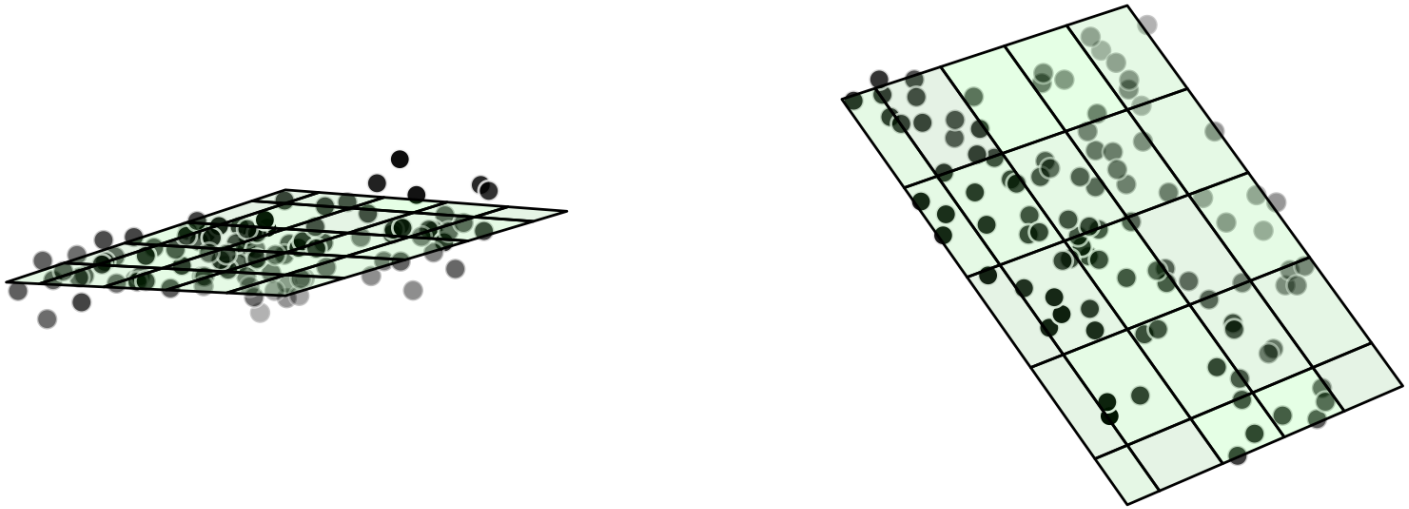
- Just like their scalar-valued versions, these cost functions *are always convex regardless of the dataset used*.
- They also decompose over the weights $\boldsymbol{w}_c$ associated with each output dimension.
- For example, we can rewrite the Least Absolute Deviations cost by swapping the summands over P and C, as:

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

- Since the weights from each of the C subproblems do not interact,…
  …if desired, we can minimize each $g_c(\cdot)$ for an optimal setting of $\boldsymbol{w}_c$ *independently*,…
  …and then take their sum to form the full cost function g.

Example. Fitting a linear model to a multi-output regression dataset

- Below, an example of multi-output linear regression is shown, using a toy dataset with input dimension $N = 2$ and output dimension $C = 2$.
- In each of the two panels of the figure, the input and one output value are plotted.
- The parameters of an appropriate linear model are tuned via minimizing the Least Squares cost, using gradient descent, with 200 steps and steplength $\alpha = 1$.



- The fully trained model is shown in green in each panel, by evaluating a fine mesh of points in the input region of the dataset.

Python implementation

- Multi-output regression cost functions can also be implemented in Python 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))
```

- Implementing the model using np.dot as np.dot(x.T,w[1:]) is especially efficient since NumPy's np.dot operation is far more efficient than constructing a linear combination in Python via an explicit for loop.

Resources

- Ch. 5: 5.1 – 5.7. Optional sections: 5.2.4, 5.6.3, 5.9.1.
- Animations
- Optional Reading: Zaki and Meira (2020) Ch. 23: 23.1 – 23.3