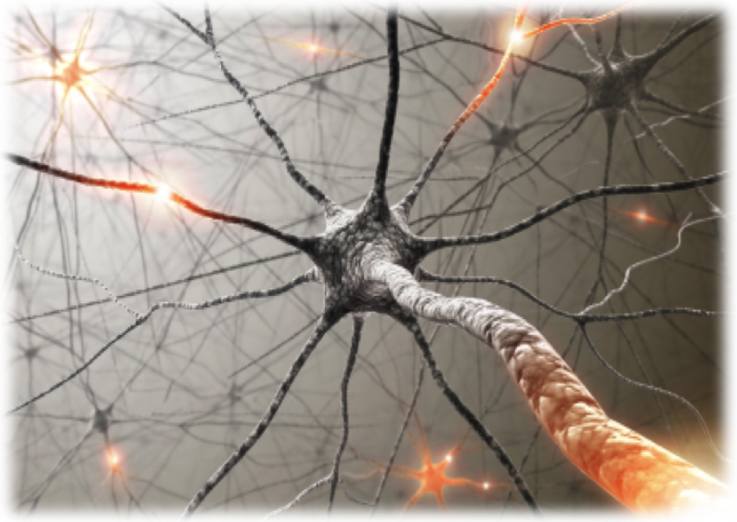
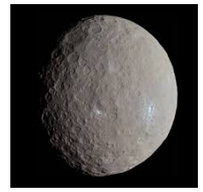
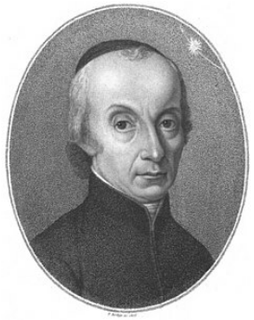
Machine learning: linear regression



- In this module, we will cover the basics of linear regression.

The discovery of Ceres





1801: astronomer Piazzi discovered Ceres, made 19 observations of location before it was obscured by the sun

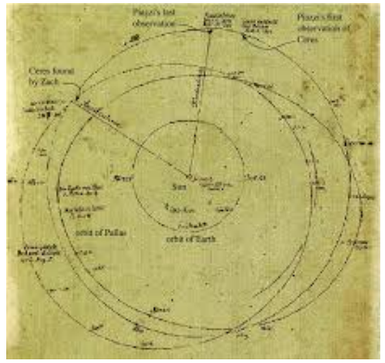
Time	**Right ascension**	**Declination**
Jan 01, 20:43:17.8	50.91	15.24
Jan 02, 20:39:04.6	50.84	15.30
...	...	...
Feb 11, 18:11:58.2	53.51	18.43

When and where will Ceres be observed again?

- Our story of linear regression starts on January 1, 1801, when an Italian astronomer Giuseppe Piazzi noticed something in the night sky while looking for stars, which he named Ceres. Was it a comet or a planet? He wasn't sure.
- He observed Ceres over 42 days and wrote down 19 data points, where each one consisted of a timestamp along with the right ascension and declination, which identifies the location in the sky.
- Then Ceres moved too close to the sun and was obscured by its glare. Now the big question was when and where will Ceres come out again?
- It was now a race for the top astronomers at the time to answer this question.

Gauss's triumph

September 1801: Gauss took Piazzi's data and created a model of Ceres's orbit, makes prediction

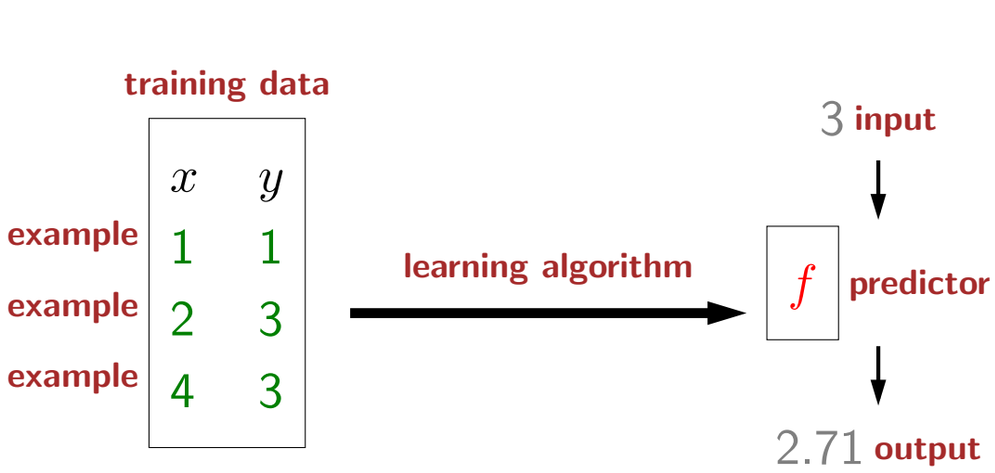


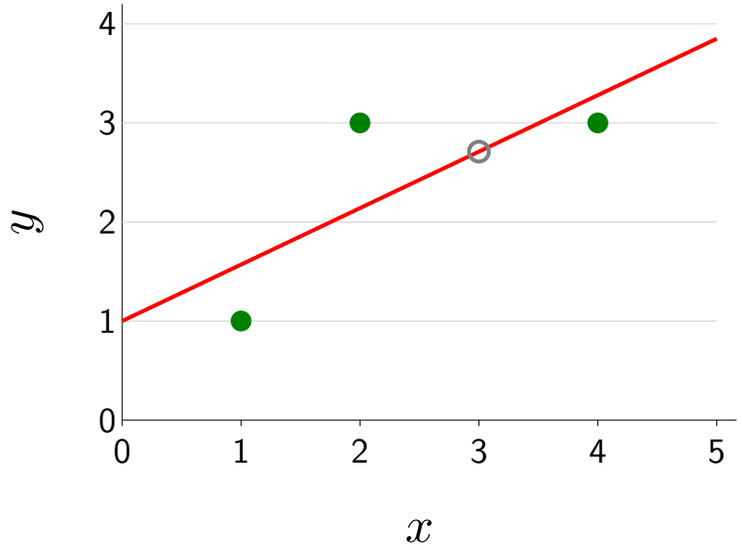
December 7, 1801: Ceres located within 1/2 degree of Gauss's prediction, much more accurate than other astronomers

Method: least squares linear regression

- Carl Friedrich Gauss, the famous German mathematician, took the data and developed a model of Ceres's orbit and used it to make a prediction.
- Clearly without a computer, Gauss did all his calculations by hand, taking over 100 hours.
- This prediction was actually quite different than the predictions made by other astronomers, but in December, Ceres was located again, and Gauss's prediction was by far the most accurate.
- Gauss was very secretive about his methods, and a French mathematician Legendre actually published the same method in 1805, though Gauss had developed the method as early as 1795.
- The method here is least squares linear regression, which is a simple but powerful method used widely today, and it captures many of the key aspects of more advanced machine learning techniques.

Linear regression framework





Design decisions:

- Which predictors are possible? **hypothesis class**
- How good is a predictor? **loss function**
- How do we compute the best predictor? **optimization algorithm**

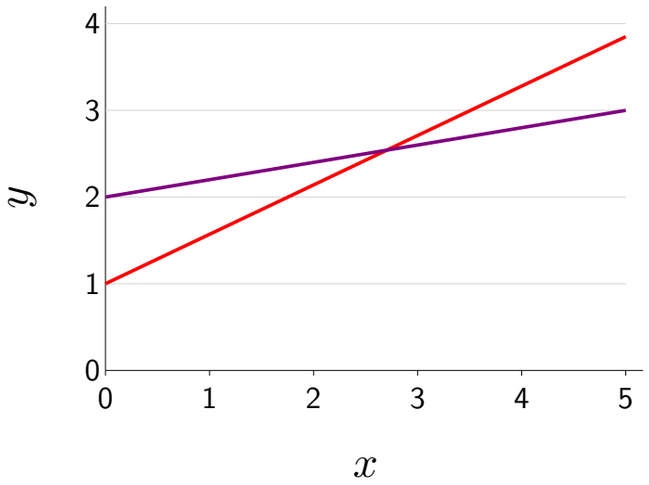
- Let us now present the linear regression framework.
- Suppose we are given **training data**, which consists of a set of examples. Each **example** (also known as data point, instance, case) consists of an input x and an output y. We can visualize the training set by plotting y against x.
- A learning algorithm takes the training data and produces a model f, which we will call a **predictor** in the context of regression. In this example, f is the red line.
- This predictor allows us to make predictions on new inputs. For example, if you feed 3 in, you get $f(3)$, corresponding to the gray circle.
- There are three design decisions to make to fully specify the learning algorithm:
- First, which predictors f is the learning algorithm allowed to produce? Only lines or curves as well? In other words, what is the **hypothesis class**?
- Second, how does the learning algorithm judge which predictor is good? In other words, what is the **loss function**?
- Finally, how does the learning algorithm actually find the best predictor? In other words, what is the **optimization algorithm**?

Hypothesis class: which predictors?

$$f(x) = 1 + 0.57x$$

$$f(x) = 2 + 0.2x$$

$$f(x) = w_1 + w_2 x$$



Vector notation:

weight vector $\mathbf{w} = [w_1, w_2]$ feature extractor $\phi(x) = [1, x]$ feature vector

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) \text{ score}$$

$$f_{\mathbf{w}}(3) = [1, 0.57] \cdot [1, 3] = 2.71$$

Hypothesis class:

$$\mathcal{F} = \{f_{\mathbf{w}} : \mathbf{w} \in \mathbb{R}^2\}$$

- Let's consider the first design decision: what is the hypothesis class? One possible predictor is the red line, where the intercept is 1 and the slope is 0.57, Another predictor is the purple line, where the intercept is 2 and the slope is 0.2.
- In general, let's consider all predictors of the form $f(x) = w_1 + w_2 x$, where the intercept w_1 and and the slope w_2 can be arbitrary real numbers.
- Now let us generalize this further using vector notation. Let's pack the intercept and slope into a single vector, which we will call the **weight vector** (more generally called the parameters of the model).
- Similarly, we will define a **feature extractor** (also called a feature map) ϕ, which takes x and converts it into the **feature vector** $[1, x]$.
- Now we can succinctly write the predictor $f_{\mathbf{w}}$ to be the dot product between the weight vector and the feature vector, which we call the **score**.
- To see this predictor in action, let us feed $x = 3$ as the input and take the dot product.
- Finally, define the hypothesis class $\mathcal{F}$ to be simply the set of all possible predictors $f_{\mathbf{w}}$ as we range over all possible weight vectors $\mathbf{w}$. This is the possible functions that we want our learning algorithm to consider.

Loss function: how good is a predictor?

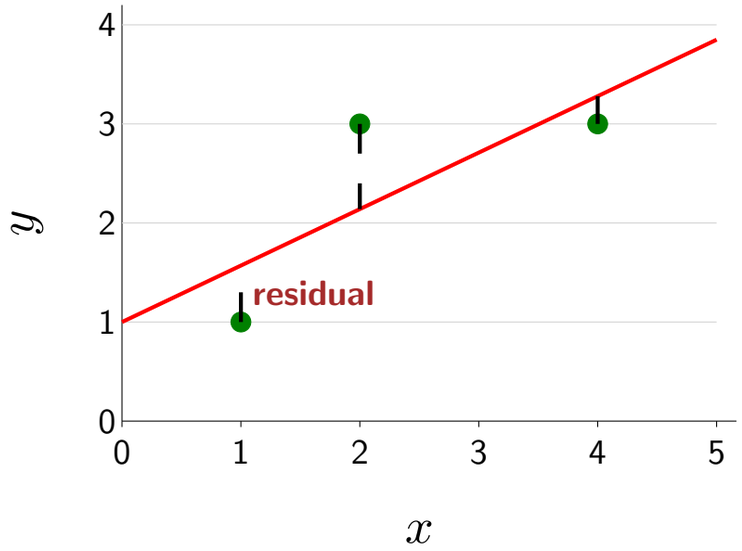
$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

$\mathbf{w} = [1, 0.57]$

$\phi(x) = [1, x]$

training data $\mathcal{D}_{\text{train}}$

x	y
1	1
2	3
4	3



$$\text{Loss}(x, y, \mathbf{w}) = (f_{\mathbf{w}}(x) - y)^2$$ **squared loss**

$$\text{Loss}(1, 1, [1, 0.57]) = ([1, 0.57] \cdot [1, 1] - 1)^2 = 0.32$$

$$\text{Loss}(2, 3, [1, 0.57]) = ([1, 0.57] \cdot [1, 2] - 3)^2 = 0.74$$

$$\text{Loss}(4, 3, [1, 0.57]) = ([1, 0.57] \cdot [1, 4] - 3)^2 = 0.08$$

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

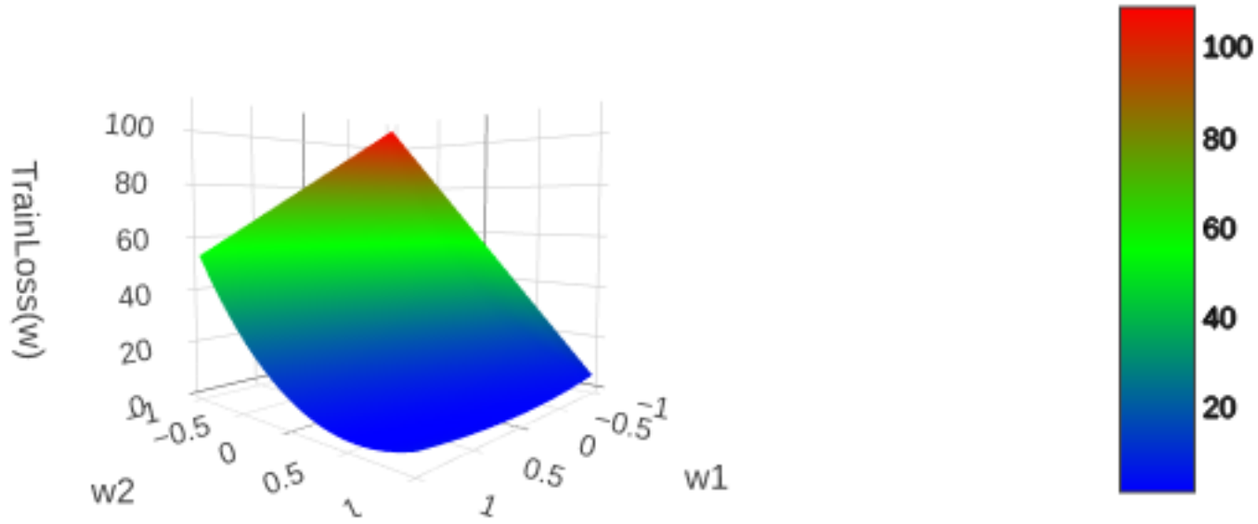
$$\text{TrainLoss}([1, 0.57]) = 0.38$$

- The next design decision is how to judge each of the many possible predictors.
- Going back to our running example, let's consider the red predictor defined by the weight vector [1, 0.57], and the three training examples.
- Intuitively, a predictor is good if it can fit the training data. For each training example, let us look at the difference between the predicted output $f_{\mathbf{w}}(x)$ and the actual output y, known as the **residual**.
- Now define the **loss function** on given example with respect to $\mathbf{w}$ to be the residual squared (giving rise to the term least squares). This measures how badly the function f screwed up on that example.
- Aside: You might wonder why we are taking the square of the residual as opposed to taking an absolute value of the residual (known as the absolute deviation): the answer for now is both mathematical and computational convenience, though if your data potentially has outliers, it is beneficial to use the absolute deviation.
- For each example, we have a per-example loss computed by plugging in the example and the weight vector. Now, define the **training loss** (also known as the training error or empirical risk) to be simply the average of the per-example losses of the training examples.
- The training loss of this red weight vector is 0.38.
- If you were to plug in the purple weight vector or any other weight vector, you would get some other training loss.

Loss function: visualization

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} (f_{\mathbf{w}}(x) - y)^2$$

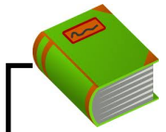
$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$



- We can visualize the training loss in this case because the weight vector $\mathbf{w}$ is only two-dimensional. In this plot, for each w_1 and w_2, we have the training loss. Red is higher, blue is lower.
- It is now clear that the best predictor is simply the one with the lowest training loss, which is somewhere down in the blue region. Formally, we wish to solve the optimization problem.

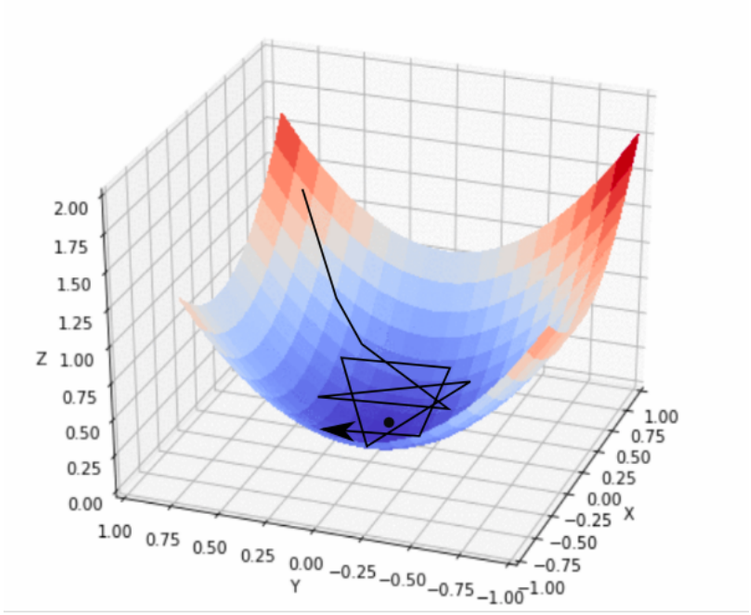
Optimization algorithm: how to compute best?

Goal: $\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$



Definition: gradient

The gradient $\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$ is the direction that increases the training loss the most.



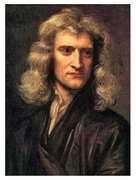
Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$: **epochs**

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})}_{\text{gradient}}$$

- Now the third design decision: how do we compute the best predictor, i.e., the solution to the optimization problem?
- To answer this question, we can actually forget that we're doing linear regression or machine learning. We simply have an objective function TrainLoss($\mathbf{w}$) that we wish to minimize.
- We will adopt the "follow your nose" strategy, i.e., **iterative optimization**. We start with some $\mathbf{w}$ and keep on tweaking it to make the objective function go down.
- To do this, we will rely on the gradient of the function, which tells us the direction to move in that will decrease the objective function the most.
- Formally, this iterative optimization procedure is called **gradient descent**. We first initialize $\mathbf{w}$ to some value (say, all zeros).
- Then perform the following update T times, where T is the number of **epochs**: Take the current weight vector $\mathbf{w}$ and subtract a positive constant η times the gradient. The **step size** η specifies how aggressively we want to pursue a direction.
- The step size η and the number of epochs T are two **hyperparameters** of the optimization algorithm, which we will discuss later.



Computing the gradient

Objective function:

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} (\mathbf{w} \cdot \phi(x) - y)^2$$

Gradient (use chain rule):

$$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} 2(\underbrace{\mathbf{w} \cdot \phi(x) - y}_{\text{prediction} - \text{target}})\phi(x)$$

- To apply gradient descent, we need to compute the gradient of our objective function TrainLoss($\mathbf{w}$).
- You could throw it into TensorFlow or PyTorch, but it is pedagogically useful to do the calculus, which can be done by hand here.
- The main thing here is to remember that we're taking the gradient with respect to $\mathbf{w}$, so everything else is a constant.
- The gradient of a sum is the sum of the gradient, the gradient of an expression squared is twice that expression times the gradient of that expression, and the gradient of the dot product $\mathbf{w} \cdot \phi(x)$ is simply $\phi(x)$.
- Note that the gradient has a nice interpretation here. For the squared loss, it is the residual (prediction - target) times the feature vector $\phi(x)$.
- Aside: no matter what the loss function is, the gradient is always something times $\phi(x)$ because $\mathbf{w}$ only affects the loss through $\mathbf{w} \cdot \phi(x)$.

Gradient descent example

training data $\mathcal{D}_{\text{train}}$

x	y
1	1
2	3
4	3

$$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} 2(\mathbf{w} \cdot \phi(x) - y)\phi(x)$$

Gradient update: $\mathbf{w} \leftarrow \mathbf{w} - 0.1 \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

t	$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$	$\mathbf{w}$
		$[0, 0]$
1	$\underbrace{\frac{1}{3}(2([0,0] \cdot [1,1] - 1)[1,1] + 2([0,0] \cdot [1,2] - 3)[1,2] + 2([0,0] \cdot [1,4] - 3)[1,4])}_{=[-4.67, -12.67]}$	$[0.47, 1.27]$
2	$\underbrace{\frac{1}{3}(2([0.47,1.27] \cdot [1,1] - 1)[1,1] + 2([0.47,1.27] \cdot [1,2] - 3)[1,2] + 2([0.47,1.27] \cdot [1,4] - 3)[1,4])}_{=[2.18, 7.24]}$	$[0.25, 0.54]$
...	...	...
200	$\underbrace{\frac{1}{3}(2([1,0.57] \cdot [1,1] - 1)[1,1] + 2([1,0.57] \cdot [1,2] - 3)[1,2] + 2([1,0.57] \cdot [1,4] - 3)[1,4])}_{=[0,0]}$	$[1, 0.57]$

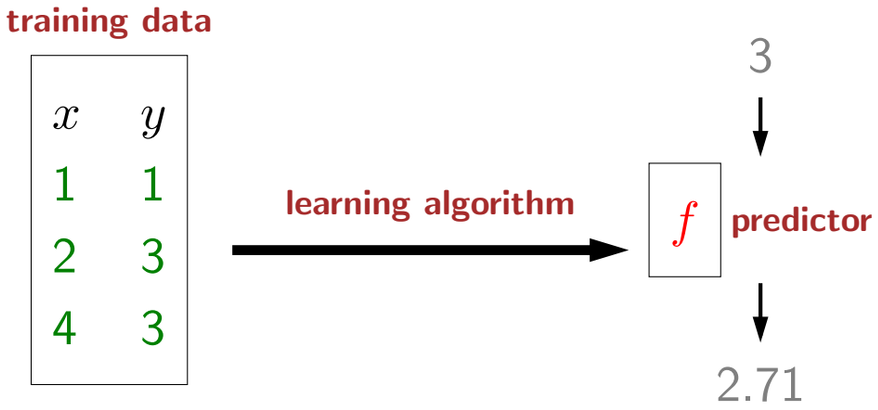
- Let's step through an example of running gradient descent.
- Suppose we have the same dataset as before, the expression for the gradient that we just computed, and the gradient update rule, where we take the step size $\eta = 0.1$.
- We start with the weight vector $\mathbf{w} = [0, 0]$. Let's then plug this into the expression for the gradient, which is an average over the three training examples, and each term is the residual (prediction - target) times the feature vector.
- That vector is multipled by the step size (0.1 here) and subtracted out of the weight vector.
- We then take the new weight vector and plug it in again to the expression for the gradient. This produces another gradient, which is used to update the weight vector.
- If you run this procedure for long enough, you eventually get the final weight vector. Note that the gradient at the end is zero, which indicates that the algorithm has converged and running it longer will not change anything.

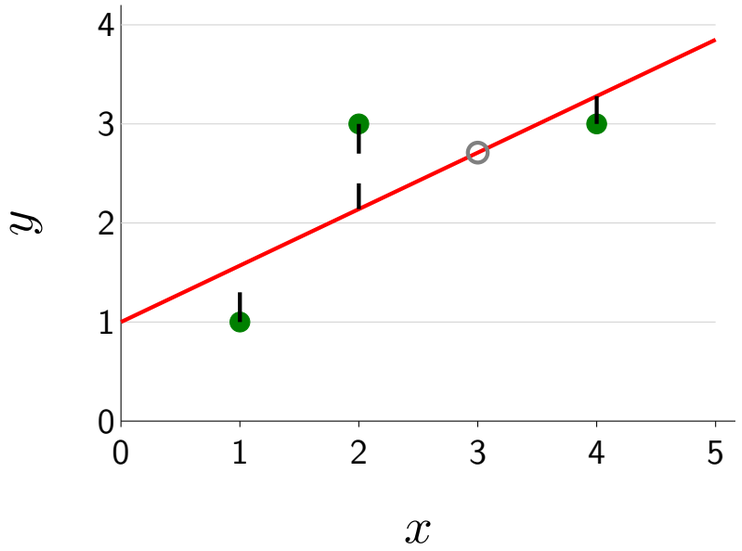
Gradient descent in Python

[code]

- To make things even more concrete, let's code up gradient descent in Python.
- In practice, you would probably use TensorFlow or PyTorch, but I will go with a very bare bones implementation on top of NumPy just to emphasize how simple the ideas are.
- Note that in the code, we are careful to separate out the **optimization problem** (what to compute), that of minimizing training loss on the given data, from the **optimization algorithm** (how to compute it), which works generically with the weight vector and gradients of the objective function.

Summary





Which predictors are possible?

Hypothesis class

Linear functions

$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)\}, \phi(x) = [1, x]$

How good is a predictor?

Loss function

Squared loss

$\text{Loss}(x, y, \mathbf{w}) = (f_{\mathbf{w}}(x) - y)^2$

How to compute best predictor?

Optimization algorithm

Gradient descent

$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \textbf{TrainLoss}(\mathbf{w})$

- In this module, we have gone through the basics of linear regression. A learning algorithm takes training data and produces a predictor f, which can then be used to make predictions on new inputs.
- Then we addressed the three design decisions:
- First, what is the hypothesis class (the space of allowed predictors)? We focused on linear functions, but we will later see how this can be generalized to other feature extractors to yield non-linear functions, and beyond that, neural networks.
- Second, how do we assess how good a given predictor is with respect to the training data? For this we used the squared loss, which gives us least squares regression. We will see later how other losses allow us to handle problems such as classification.
- Third, how do we compute the best predictor? We described the simplest procedure, gradient descent. Later, we will see how stochastic gradient descent can be much more computational efficient.
- And that concludes this module.