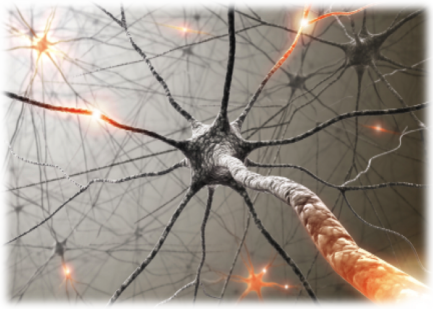
Linear regression



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

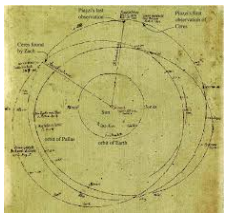
When and where will Ceres be observed again?

CS221 2

- 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.
- I now encourage you to pause the video and think about how you would go about solving this problem.

Gauss's triumph

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



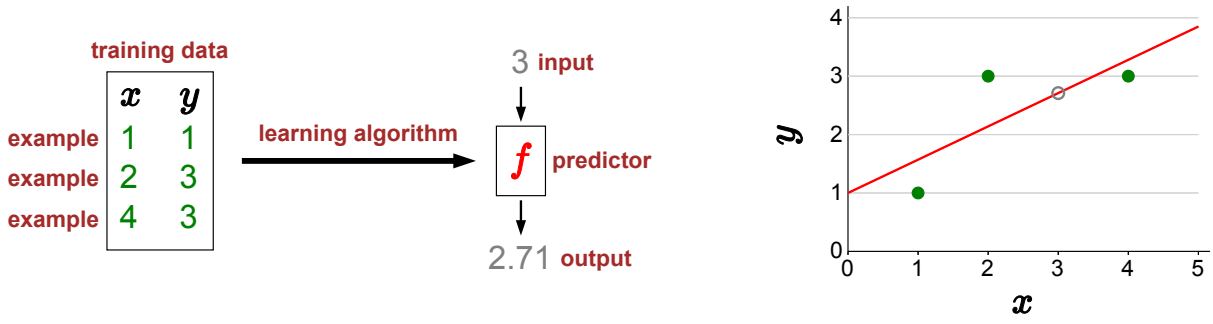
December 7, 1801: Ceres located within 1/2 degree of Gauss's prediction, much more accurate than other astronomers

Method: least squares linear regression

CS221 4

- 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, ten years prior.
- 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, such as neural networks and deep learning.

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**

CS221 6

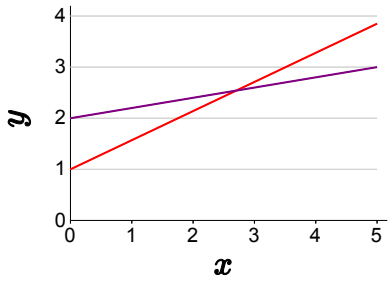
- Let us now present the linear regression framework.
- Suppose we are given **training data**, which consists of a set of examples. Each **example** (also known as data point, instance, case) consists of an input x and an output y. We can visualize the training set by plotting y against x.

- 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 a machine learning algorithm:
- First, which predictors f is the learning algorithm allowed to produce? Do we draw a straight line, or can we draw a squiggly line that passes through all the points? This choice is what we call the **hypothesis class**, since it is a hypothesis about possible relationships between x and y. The hypothesis class is the set of predictors or functions that we will be considering.
- Second, how does the learning algorithm judge which predictor in this set is good? In this case, you and I are probably eyeballing the distance between the point and the line but we need to express this more formally. This is the **loss function** which measures how well a predictor fits an example
- Finally, how can we find the lowest loss predictor in our hypothesis class? Here, we might as for some explicit algorithm that gives us the line that minimizes the vertical distance to the points. We call this 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:

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

CS221 8

- 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.

- We are going to write this down in a slightly more general, vector notation which will be handy later. Let's first pack the intercept and slope into a single vector, which we will call the **weight vector** w (more generally we are going to call this 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** with two elements. The first element is a constant one, and the second is the value of x itself

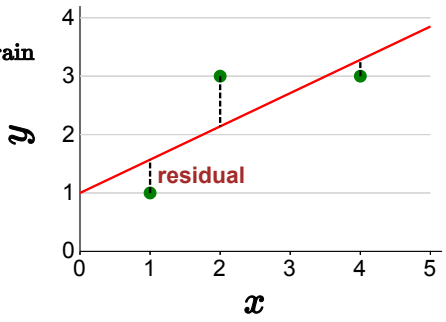
- Now we can succinctly write the predictor $f_{\mathbf{w}}$ to be the dot product between the weight vector and the feature vector, which we call the **score**.
- To see this predictor in action, let us feed $x = 3$ as the input and take the dot product.
- We now have a way of writing down predictors We can use this to define the hypothesis class $\mathcal{F}$ as the set of all possible predictors $f_{\mathbf{w}}$ as we range over all possible weight vectors $\mathbf{w}$.
  Our goal will be to find the best predictor within this class of linear functions

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

$$\text{Loss}(1, 1, [1, 0.57]) = ([1, 0.57] \cdot [1, 1] - 1)^2 = 0.32$$
$$\text{Loss}(2, 3, [1, 0.57]) = ([1, 0.57] \cdot [1, 2] - 3)^2 = 0.74$$
$$\text{Loss}(4, 3, [1, 0.57]) = ([1, 0.57] \cdot [1, 4] - 3)^2 = 0.08$$

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

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

CS221 10

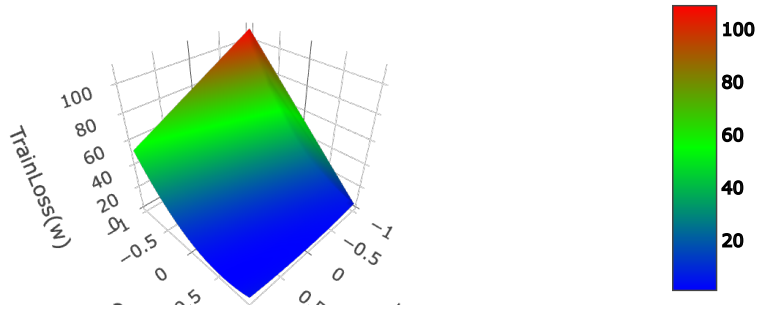
- The next design decision is how to define what we mean when we say a predictor is good or bad
- Going back to our running example, let's consider the red predictor defined by the weight vector [1, 0.57], and the three training examples.
- Intuitively, a predictor is good if it can fit the training 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.

- If this is your first time seeing regression, you might be wondering why we are squaring the residual. There are somewhat deeper reasons from statistical and optimization perspectives for why squaring would be the natural thing to do I will discuss this briefly later, but if you are interested, I encourage you to ask in office hours or Q and A.
- Returning to our loss, we now 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.

tion

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



CS221 12

- 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.

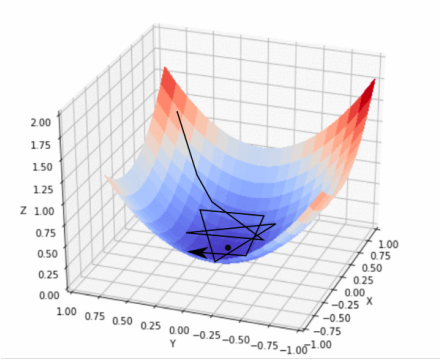
- It is now clear that the best predictor is simply the one with the lowest training loss, which is somewhere down in the blue region. Formally, we are looking for the predictor that minimizes the training error, or empirical risk..

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

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

CS221 14

- Now the third design decision: how do we compute the best predictor, i.e., the solution to the optimization problem?
- The dominant approach in machine learning is to abstract away the optimization problem. Often, we can ignore the fact that we are doing linear least squares, or even machine learning and simply treat the objective function $\text{TrainLoss}(\mathbf{w})$ as a black-box to optimize.
- In this case, an effective approach is **iterative optimization**, which resembles a form of trial-and-error. We start with some $\mathbf{w}$ and change it slightly to make the objective function smaller.
- But how should we be changing the function? trying random small changes to $\mathbf{w}$ is wildly inefficient. Ideally, we can find the direction to modify $\mathbf{w}$ so that it makes the loss go down as fast as possible. It turns out that we can identify this **exactly** since it is the negative of the gradient of the loss

- Formally, this iterative optimization procedure is called **gradient descent**. We first initialize $\mathbf{w}$ to some value.
- Then perform the following update T times, where T is called 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, initialization, and the number of epochs T are **hyperparameters** of the optimization algorithm, which we will discuss later.

- Finally, we are going to be discussing gradient descent, or forms of it, because of its ease of use and popularity but gradient descent is not necessarily the best optimizer, especially for simple problems like linear regression. In our case, we can **exactly** solve the least-squares regression problem using either singular value decomposition or a pseudoinverse.

Computing the gradient

Objective function:

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

Gradient (use chain rule):

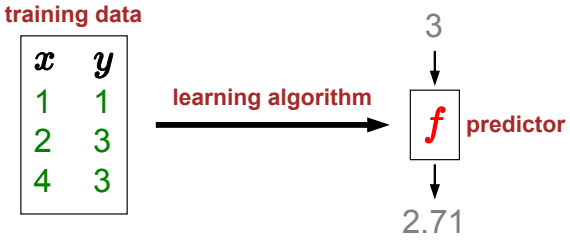
$$\nabla_{\mathbf{w}} \textbf{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} 2(\underbrace{\mathbf{w} \cdot \phi(x) - y}_{\text{prediction}-\text{target}})\phi(x)$$

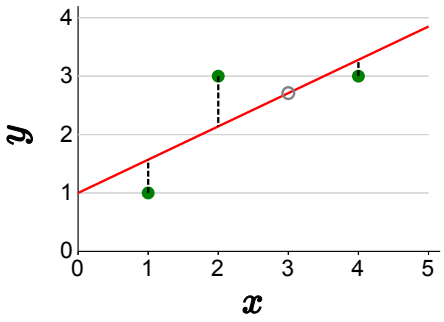
CS221 16

- To apply gradient descent, we need to compute the gradient of our objective function $\textbf{TrainLoss}(\mathbf{w})$.
- 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.
- Gradients are a lot like derivatives -- they are linear and therefore the gradient of a sum is the sum of the gradient. To take the gradient, we can apply the chain rule, which means that we first take the gradient of the square, and then gradient inside the square Finally, the gradient of the dot product $\mathbf{w} \cdot \phi(x)$ is simply $\phi(x)$.

- If this doesn't sound familiar, then I'd encourage you to brush up on your single-variable calculus.
- Note that the gradient has a nice interpretation here. For the squared loss, it is the residual (prediction - target) times the feature vector $\phi(x)$.
- This is a very natural update function: take the prediction error and try to map the error back to the weights. Going back to our earlier discussion about loss function, this is related to the optimization reason for choosing the squared loss, when a model makes very small error, its gradient is also small. The same is not true of using the absolute value.

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

CS221 18

- 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.