Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2024

Contents

CHAPTER 2

Regression

Machine learning, as demonstrated in the first chapter, is an incredibly broad subject.

- The best way to start, then, is with an example.

We'll start with a simple model: **regression**.

- As we go through, we'll develop some broader **concepts**: we'll use these throughout the rest of the class.

2.1 Problem Formulation

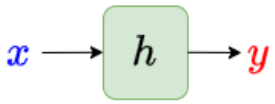
2.1.1 Hypothesis (Review)

In last chapter, we broke up our model into two parts: **problem** and **solution**.

- Our problem: using our input x to **predict** output y.

$$x \to \boxed{?} \to y$$

- Our solution: we use a function, called a **model**.
  - This is also called our **hypothesis** h.



Our hypothesis reads an input x, and predicts the output y.

Definition 1

A **hypothesis** is a **function** we use to predict y, based on x.

$$y = h(x)$$

This is also called a **model** in machine learning.

There are many models we could use: this makes it hard to search for a good one.

- One solution is to restrict ourselves to only a certain **class** of model.

  Each of these is a different kind of model we could try:

$$h_A(x) = \theta_1 x + \theta_0 \qquad h_B(x) = \theta_2 x^2 + \theta_1 x + \theta_0 \qquad h_C(x) = \theta_2 \sin(\theta_1 x) + \theta_0$$

Each of these formats represents a **hypothesis class**, or a "model class".

Definition 2

A **hypothesis class** $\mathcal{H}$ is a **set** of possible hypotheses.

- Typically, we include all of the hypotheses with the **same equation format**.

Example: Let's consider the hypothesis class, represented by h_A:

$$\mathcal{H}_A = \left\{ h(x) \quad : \quad h(x) = \theta_1 x + \theta_0 \right\} \tag{2.1}$$

Not familiar with set notation? $\mathcal{H}_A$ is:

"the set of every function that looks like $\theta_1 x + \theta_0$".

Here are a few example functions from $\mathcal{H}_A$:

$$h_1(x) = 1x + 5 \qquad h_2(x) = 3x - 9 \qquad h_3(x) = -10x + 10 \qquad h_4(x) = e^2 x - \pi$$

This makes things easier: rather than searching all possible hypotheses, we're searching $\mathcal{H}$.

Concept 3

Our goal is to **search** $\mathcal{H}$, to find a good **hypothesis** h.

Within a hypothesis class, every hypothesis has the **same structure**.

- What makes a hypothesis different? The **constants** θ_i.
- We call these **parameters.**

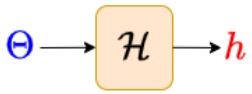
Definition 4

A **parameter** θ_i is a **number** that is plugged into the hypothesis.

- Each **hypothesis** $h \in \mathcal{H}$ has a **unique list of parameters** Θ.

Typically, θ_i is a real number, for our purposes.

If we know our model class $\mathcal{H}$, Θ **fully defines** our hypothesis h.



By plugging in our Θ values into the formula for h, we get a particular hypothesis.

For example:

$$\begin{bmatrix} 2 \\ -7 \end{bmatrix} \longrightarrow h(x) = \theta_1 x + \theta_0 \longrightarrow h_1(x) = 2x - 7$$

This Θ is what we'll modify, and use to find a **better model**.

Concept 5

Often, we already know which hypothesis class $\mathcal{H}$ we're using.

- So, it's very easy to convert back-and-forth between Θ and h.

So, we often consider the **parameters** and **hypothesis** interchangeable, or equivalent.

- "Finding a good hypothesis" and "finding good parameters" are, more or less, the same problem.

Notice that we have **two separate steps** of plugging in, when we use a model h:

- When choosing our model h, we "plug in" **parameters** Θ to create our equation.

$$h(x) = \theta_1 x + \theta_0 \xrightarrow{\Theta = \begin{bmatrix} 2 \\ -7 \end{bmatrix}} h_1(x) = 2x - 7 \tag{2.2}$$

- When we want to use our model to predict y, we "plug in" our **input value** x.

$$h_1(x) = 2x - 7 \xrightarrow{x=5} h_1(5) = 2(10) - 7 = 13 \tag{2.3}$$

We'll introduce some new notation to keep the difference clear.

Notation 6

We can write the same hypothesis h **two different ways**:

$$h(x) \qquad h(x; \Theta)$$

- The first notation is denser and **simpler** to read.
- The second notation includes Θ, acknowledging that we had to "plug in" **parameters** to create h.

We **distinguish** between "input variables" and "parameters" by separating them with a **semicolon** ;.

2.1.2 The Problem of Regression

Our hypothesis is a function that solves a **problem**. What kind of problem are we dealing with?

We distinguish different types of problems based on two things:

- **Inputs**: what kind of data do we have to work with?
- **Outputs**: what are we trying to predict?

The notation for functions reflects this idea: what matters most is, "what comes in, and what goes out".

Functions, of course, weren't specifically designed for machine learning. But the same idea applies to other STEM disciplines.

Notation 7

A **function** is notated based on what sorts of **inputs** it can take, and the **outputs** it can return.

A function f is written like this:

$$f : \text{set of inputs} \to \text{set of outputs}$$

- **Example:** Suppose that the input is "**income**" (real number $r \in \mathbb{R}$) and the output is "**number of hats owned**" (natural number $n \in \mathbb{N}$). The function for this would be

$$f : \mathbb{R} \to \mathbb{N}$$

Sometimes, we'll call our set of inputs, the **input space**.

Definition 8

The **input space** is the set of all **possible inputs** to our hypothesis h.

Technically, a **space** is a set "with **added** structure", which is about as broad as it sounds.

With that out of the way, let's talk about **regression**:

- In regression, we receive data as a **real-valued vector**, and converting it into a **real number**.

Remember the notation for real-numbered vectors we introduced in the last chapter!

Writing this a little more formally:

Definition 9

Regression is a **machine learning problem** where we use a **vector of real numbers** to predict a **real-valued number**.

In other words, we want a **hypothesis** h of the form:

$$h : \mathbb{R}^d \to \mathbb{R}$$

Example: If you have **3 values** in your input vector (height, weight, age) and **1 real output** (life expectancy), you would need a hypothesis

$$h : \mathbb{R}^3 \to \mathbb{R}$$

A more visual example:

In this example, you have one input $x \in \mathbb{R}$ (x-axis), and you want to **predict** the output $y \in \mathbb{R}$ (y-axis) based on that. These points are the dataset you want to **learn** to match.

2.1.3 Converting our data

Often, our data **wont fit** this format: maybe we have a car brand, or a color as a variable.

- This requires **converting** this data into real numbers. We do this using something called a **feature** transformation.

Definition 10

A **feature** is one distinct piece of **information** in our input.

A **feature transformation** takes those pieces of information, and **transforms** them - often, a more **useful** data type.

- In other cases, we use it on data that's already in the right format, to find **new patterns** in data (we'll return to this idea in a later chapter).

Example: You have three car brands. Instead of representing them normally, you instead turn them into vectors:

$$\text{Brand A} \rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \text{Brand B} \rightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \text{Brand C} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.4}$$

This particular feature transformation is called **one-hot encoding**! We'll return to it later.

We do our feature transformation with a function: we often notate this function as $\varphi(x)$.

Notation 11

The **function** we use to do a **feature transformation** is typically written as φ.

- If our input data is x, the **transformed** data is written as $\varphi(x)$.

Example: For our above car brand example:

$$\varphi\Big(\text{Brand A}\Big) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{2.5}$$

There are many different feature transformations for different needs. We will come back to this in a later chapter.

- For now, we will simply assume that all of our inputs x are **already** in $\mathbb{R}^d$ (vectors of real numbers).

2.1.4 Our dataset

Now, we want to find a hypothesis that solves our *particular* regression problem well.

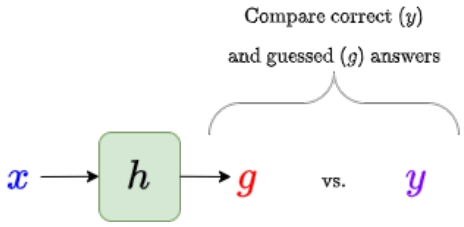
- But in order to predict results, we need **data**.

> **Concept 12**
> **Regression** is a **supervised problem**:
>
> This means that, when we're trying to predict the correct answer y, we **already have** that answer.
>
> - That way, we can check how good/bad our model's **prediction** was.

Example: A student practicing "supervised learning" has a practice exam, with all of the solutions.

- They try to do the exam **without** looking at the solutions.
- After they finish, they **check** the solutions, to see what they did wrong, and how they can do better.

In this analogy, each data point includes a "**question**" (input x) and the "**answer**" (output y) to that question.



Our model doesn't actually produces y: it produces a guess g, which we hope is similar to y.

We want to **pair up** inputs with their correct outputs, so we'll write our first data point as

$$\left(x, y\right) \tag{2.6}$$

But we have many data points we need to sort, like this.

- We'll distinguish each data point using $x^{(i)}$ notation, from last chapter:

Notation 13

$x^{(i)}$ is the i^{th} **data point**, represented as a vector.

- Sometimes, you may instead see the notation x_i.

We can rewrite this as:

$$\left(x^{(1)}, y^{(1)}\right)$$

Repeating this for all of our data, we have a set of n data points, $\mathcal{D}_n$:

$$\mathcal{D}_n = \left\{\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(n)}, y^{(n)}\right)\right\}$$

2.1.5 Training our model

We'll use this **training data** to train our model: hopefully, if our model performs well with this limited data, it'll perform well with new data.

But as we'll see below, that's not always the case.

In order to train our model, we need to know how **good** or bad it is, on the data we can see.

- We'll measure the "badness" of our model as **loss**, from last chapter.

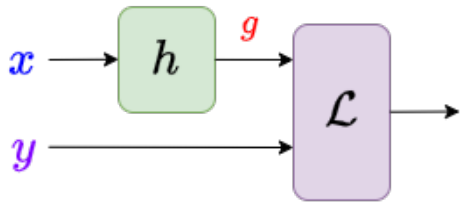
Definition 14

(Review from Introduction Chapter)

A **loss function** measures how **poorly** your machine is **performing** on a **task**.

- The output is a **real number**.

If your machine is performing **well**, then you will have a **low** output. And vice versa: if it is doing **badly**, it will have a **high** output.



Our loss function gives us our tool for "comparing" y to g.

We'll take the **average** loss, across all of our **training data**. This is our **training error**.

Key Equation 15

Training Error $\mathcal{E}_n$ is written as:

$$\mathcal{E}_n(h) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}^{(i)} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(h(x^{(i)}), y^{(i)}\right)$$

This is the **expected** (average) loss over all of our training data.

Note that this is a bit weird: we're using h as an input.

- We want $\mathcal{E}_n$ to **evaluate** and compare different hypotheses h: it has to receive that hypothesis in order to evaluate it.

The hypothesis, in turn, takes the training data as input.

Clarification 16

h is a **function** that takes a **variable**, x, as its input. This is the usual format.

$\mathcal{E}_n$ is also a function, but it takes in h, a **different function**, as an input!

- That means one function is using **another function** as an input. This can sometimes cause confusion.

We do this because $\mathcal{E}_n$ observes our hypothesis, and outputs, "**how bad** in this particular hypothesis at **predicting** this data".

"Training", in the simplest case, boils down to reducing our error as much as possible.

Below, we'll add another component, that makes this more complicated.

- But wait: our goal **isn't** to improve performance on our **training data**.
- Our goal is to perform well on **new data**!

2.1.6 Learning to Generalize

Above, our idea was, "get better at practice data, get better at future data". But this has a problem:

- Even though our training data and future data should be from the **same distribution** (IID), **randomness** means they won't be exactly the same.

So if we perfectly matched our training data, we'd be inaccurate to the real distribution.

In the last chapter, we introduced a solution: a second dataset, that we **test** our data on.

- We want our machine to handle **new situations** it hasn't seen before: testing data allows us to try out a "new situation".

Definition 17

Generalization is the ability to take something **specific**, and apply it to something more **broad**.

- In machine learning, we want our model to look at some **limited data**, and be able to perform well on a much larger body of **future data** it hasn't seen before.

So, let's find out how well our model generalizes. We'll define **test error**: our performance on the **testing data**. This time, we have m new data points.

$$\mathcal{E}(h) = \frac{1}{m} \sum_{i} \mathcal{L}\left(h(x^{(i)}),\, y^{(i)}\right) \tag{2.7}$$

We'll start counting from $n+1$ because we've already used the first n points when training.

Key Equation 18

Testing Error $\mathcal{E}$ is written as:

$$\mathcal{E}(h) = \frac{1}{m} \sum_{i=n+1}^{n+m} \mathcal{L}\left(h(x^{(i)}),\, y^{(i)}\right)$$

We could start over from $i = 1$, but that would be a bit confusing. If you said "the 10th data point", someone might ask, "from the training, or testing data? This avoids this problem.

Because we want to generalize, we want to minimize **test error**.

- But, because we want our model to do well on "data it **hasn't seen** before", we can't use it during our training process.

For now, the next best thing after "minimize test error" is "**minimize training error**", while using techniques to improve how we **generalize**.

How can we "generalize" if we can't see all of that extra data? We'll get into that below.

2.2 Regression as an optimization problem

We want to make our **loss** (error) as low as we can: we want to **minimize** it. This is a form of **optimization** - getting the best results from our system.

Most of computer science boils down to some kind of optimization.

Here, we'll introduce some of the terms and notation of optimization.

2.2.1 Objective Function

Now, we confront a major challenge: we have two different priorities.

- We want to perform well on **training data**: our model will learn some insights about the true distribution of data.
- But we also want our model to **generalize** well: we want it to do well data on it has never seen before.

Because our training data gives us a limited view of the true distribution.

Currently, our approach focuses on one priority: we measure our success using **training error**.

$$\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\left(h(x^{(i)}),\, y^{(i)}\right) \tag{2.8}$$

- This function represents "what we consider **important**": it mathematically describes what we want to improve.
- We call this an **objective function**.

Definition 19

An **objective function** J is the function that tells us what we want to improve, or **optimize**:

- Usually, this means that our goal is **minimizing** it.

We minimize our objective function by adjusting our model, via **parameters** Θ. So, we take that as our input: $J(\Theta)$.

Since we are focusing more on Θ then before, we'll replace $h(x^{(i)})$ with $h(x^{(i)};\Theta)$:

$$J(\Theta) = \text{Training Error} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\left(h(x^{(i)};\Theta), y^{(i)}\right)$$

2.2.2 The Regularizer

If our objective function describes "what we care about", and we care about **generalization**, shouldn't we **include** it in our objective function?

- Let's do that: we'll call it a **regularizer** term.

$$J(\Theta) = \text{Training Error} + \text{Regularizer} \tag{2.9}$$

We continue using our training error from before:

$$J(\Theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(h(x^{(i)};\Theta),\ y^{(i)}\right) + \text{Regularizer} \tag{2.10}$$

Later, we'll construct our regularizer so that minimizing it will create a **more general** model Θ.

- Because we want to make Θ more **general**, we'll use it in our regularizer: we'll call it $R(\Theta)$.

Our strategy is to **minimize** $J(\Theta)$: by reducing training loss and $R(\Theta)$, we hope to make a better model.

$$J(\Theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(h(x^{(i)};\Theta),\ y^{(i)}\right) + R(\Theta) \tag{2.11}$$

We're missing one thing: how much do we want to prioritize $R(\Theta)$, compared to **training error**?

- We'll **scale** our regularizer by a **constant** $\lambda \geqslant 0$.
- The larger λ is, the more we care about **generalizing** to new data.

Key Equation 20

In general, we write the **objective function** as:

$$J(\Theta) = \left(\frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(h(x^{(i)};\Theta),\ y^{(i)}\right)\right) + \lambda R(\Theta)$$

- The left term is the **loss**: how well we perform on training data.
- The right term is the **regularization**: we hope that **minimizing** it will make our model more "general" (good for new situations).

This is the function we want to **minimize**. What does our **regularizer** look like?

- We'll come back to this later. we'll go one step at a time, and first learn how to minimize **training error**.

2.2.3 More on the Objective Function

Notice that our objective function **depends** on our training data $\mathcal{D}$ as well: the same model will work better for some problems, than others.

- **Example:** A model trained in political science learns different things compared to one trained in geology.
- We'd expect it to perform pretty badly on a geology exam.

"$\mathcal{D}$ affects J, but isn't the main input" is **similar** to our previous idea: "Θ affects hypothesis h, but we usually **don't** think of it as the **input**".

- We made this distinction with some **notation**: $h(x;\Theta)$.

Notation 21

Just like how we can use ";" when writing $h(x;\Theta)$, we'll use the **same notation** for J: $J(\Theta;\mathcal{D})$

$$J(\Theta;\mathcal{D}) = \left(\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\left(h(x^{(i)};\Theta),\ y^{(i)}\right)\right) + \lambda R(\Theta)$$

Θ is our "main" input variable, but data $\mathcal{D}$ is important for computing J.

One more comment:

Clarification 22

Students often get confused by the fact that our **objective function** J is a function of Θ, while **training error** $\mathcal{E}_n$ is a function of h.

$$\overbrace{J(\Theta)}^{\text{Uses } \Theta} = \overbrace{\mathcal{E}_n(h)}^{\text{Uses } h} + \lambda R(\Theta)$$

The difference is that **training error** $\mathcal{E}_n(h)$ is **more general** than the **objective function** $J(\Theta)$, and h is **more general** than Θ.

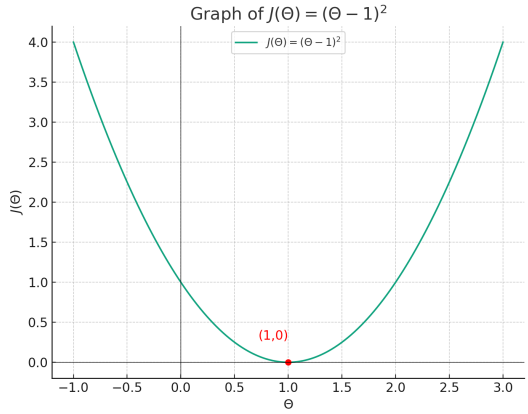
- By "more general", we mean that there are more situations where we can use h than Θ.
  - Θ is a list of parameters: we only use it if we have a **parametric model**.
  - h is used if we have **any kind of model**.

So, let's compare $\mathcal{E}_n(h)$ and $J(\Theta)$:

- **Training error** can be used for any model h, so we don't want to use Θ: our model could be **non-parametric**.
- Our **objective function assumes** we have parameters Θ: we know our model class H, we just want to optimize Θ.

2.2.4 Minimization Notation

Our goal is to **minimize** J by adjusting Θ. To demonstrate, we'll use the following example:



Take $J(\Theta) = (\Theta - 1)^2$. The minimum output is 0, which happens at $\Theta = 1$. So, we have a minimum at $(1, 0)$.

Our goal is to find this **minima**. There are two questions we're interested in:

- What is the **minimum value of** J we can find by **adjusting** Θ?
- Which **model** Θ gives us the minimal J? In other words: which model performs best?

We define a distinct function for answering each of these questions.

First:

- What is the **minimum value of** J we can find by **adjusting** Θ?

Notation 23

The **min function** gives you the **minimum output** of a function we get by adjusting one chosen **variable**.

$$\min_{\Theta} J(\Theta)$$

The **function we want to minimize** is written to the right, while the **variable we adjust** is written below.

Example:

$$\min_{\Theta} (\Theta - 1)^2 = 0 \tag{2.12}$$

0 is the minimum value of J we can find by adjusting Θ.

Next:

- What is the **minimum value of** J we can find by **adjusting** Θ?

Notation 24

The **argmin function** tells you the value of the **input variable** that gives the **minimum output**.

$$\arg\min_{\Theta} J(\Theta)$$

The **function we want to minimize** is written to the right, while the **variable we adjust** is written below.

Example:

$$\arg\min_{\Theta} (\Theta - 1)^2 = 1 \tag{2.13}$$

1 is the value of Θ which gives the minimum J.

Clarification 25

Why is it called "**argmin**"?

"**Argument**" is used as another word for "**input variable**".

And our argmin function returns the **argument** with the **minimum** output. Hence, **arg min**.

2.2.5 Optimal Value Notation

Our goal is to find the best model, represented by some Θ. We'll call this "optimal" model, Θ^*.

Notation 26

We add a **star** * to indicate the **optimal** variable choice.

If that variable is z^*, you would say it as "z-star".

Example:

$$\Theta^* = 1 \text{ for the above example.} \tag{2.14}$$

So, if we want optimal Θ, we're looking for:

Key Equation 27

Our **optimal parameter** vector is written as

$$\Theta^* = \arg\min_{\Theta} J(\Theta)$$

2.3 Linear Regression

Now that we understand the problem of **regression**, and the concept of **optimizing** over it, we'll introduce our **hypothesis class.**

We want a function that can use information to **predict** outputs.

2.3.1 The Linear Model, 1-D

We'll start off small: we have **one variable**, and something we want to predict. And we'll pick the simplest pattern we can:

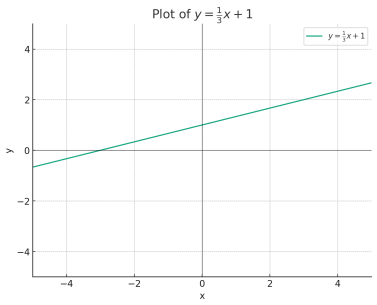
$$y = mx + b \tag{2.15}$$

A **linear** equation.

- m tells us **how much** our input x **affects** our output y.
- b accounts for everything **unrelated** to x: what is y when $x = 0$?

b and m are our **parameters**: that means they're part of Θ. We'll rename them $b = \theta_0$ and $m = \theta_1$.

$$h(x) = \theta_1 x + \theta_0 \tag{2.16}$$



Here's $\frac{1}{3}x + 1$. We call this 1D because there's only one input dimension, x. But we plot it in 2D to see the output, too!

2.3.2 The Linear Model, 2-D

We want to have **multiple** input variables: x will be a **vector**, not a number.

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{2.17}$$

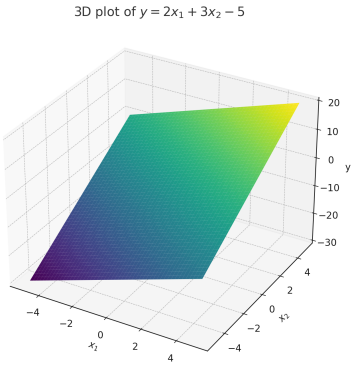
So, for our above example, we'll **replace** x with x_1.

$$h(x) = \theta_1 x_1 + \theta_0 \tag{2.18}$$

The simplest way to include x_2 by just **adding** it. We have a scaling factor θ_1 for x_1, so we'll give x_2 its own **parameter**, θ_2:

If θ_1 is the "slope" for x_1, θ_2 is the "slope" for x_2.

$$h(x) = \theta_2 x_2 + \theta_1 x_1 + \theta_0 \tag{2.19}$$



Here's a 2d regression: it's a plane. The height represents the output.

2.3.3 The Linear Model, d-D

You can **expand** this to d dimensions by **adding more terms**:

This is the "dimension" of our input space: the **number** of input variables we have.

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_d x_d \tag{2.20}$$

We need $n + 1$ dimensions to plot an n-dim regression, so... we can't plot $n > 2$.

2.3.4 The Linear Model using Vectors

We **multiply** components of x and θ together, then **add** them together. This looks like a **dot product**:

$$h(x) = \theta_0 + \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \tag{2.21}$$

If we write this symbolically, we get:

$$h(x) = \theta_0 + \theta \cdot x \tag{2.22}$$

θ includes all of our parameters, **except for** θ_0.

- θ is used for our **dot product**, Θ includes **all** parameters.

Notation 28
We represent the **parameters** of our **linear** equation as $\Theta = (\theta, \theta_0)$.

This formula looks similar to $y = mx + b$ again! Only this time, we have **vectors** instead.

We'll swap out the dot product for **matrix multiplication**.

Key Equation 29
A dot product $a \cdot b$ can be written as **matrix multiplication** instead:

$$a \cdot b = a^T b$$

In this class, we'll usually find it more useful to work with matrix multiplication.

Definition 30
The **linear regression** hypothesis is $h(x) = \theta \cdot x + \theta_0$, or

$$h(x) = \theta^T x + \theta_0$$

Make sure you know what θ^T is: it's the **transpose** of θ.

Remember that, when written out, this looks like:

$$h(x) = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \cdots & \theta_d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} + \theta_0 \tag{2.23}$$

This is the **hypothesis class** of **linear hypotheses** we will reuse throughout the class.

2.3.5 Regression Loss

We need to decide on our **loss function** for regression: how **badly** is our model is performing?

- Our goal is for our **guess** g to be close to the **real** output y.
- The more **different** they are, the worse.

We could use "absolute difference" $|g - y|$, but **squared difference** tends to be much more useful:

Why? We discuss in the Concept box below.

$$\mathcal{L}(g, y) = (g - y)^2 \tag{2.24}$$

We call this **square loss**. It punishes high and low guesses equally, and the punishments become more **severe** as the **difference** increases.

Our slope $\frac{d}{dx}x^2 = 2x$ gets larger as we move away from $x = 0$.

Concept 31

We use **square distance** for a few good reasons:

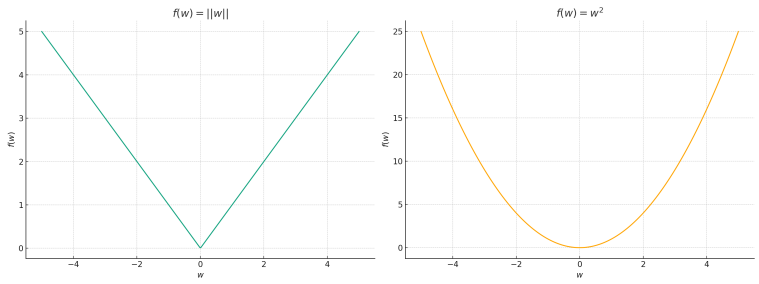
- High and low guesses are treated **equally**.
- It works well with **matrix multiplication**:

$$\|w\|^2 = w^T w$$

- $\|w\|$ is **not smooth**: it doesn't always have a derivative! $\|w\|^2$ *is* smooth.
  - $\|w\|$ doesn't have a derivative at $w = 0$.
- The **slope** becomes **small** when you get closer to the correct answer: you'll know when you're getting close.

A "stronger" version of smoothness requires that **every derivative** (f', f'', f'''...) is **continuous**.

We just care if the derivative exists everywhere, though.



What's the derivative of $\|w\|$ at 0? We don't have one!

2.3.6 Our Goal: Ordinary Least Squares

Now, we have the concepts we need.

- We want to **minimize** loss $\mathcal{L}$ on our data set, using the **linear** model Θ (h).
- We'll use that linear model to **predict** the outputs of our data points.

This goal can be turned into an **objective function**: $J(\Theta) = J(\theta, \theta_0)$

$$J(\theta, \theta_0) = \text{Training Loss} \tag{2.25}$$

Let's go step-by-step:

- Training loss is our **expected loss**, averaged over each data point. $g^{(i)}$ is our prediction for $y^{(i)}$.

Remember that $y^{(i)}$ is the "correct" answer for our i^{th} data point.

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(g^{(i)}, y^{(i)}) \tag{2.26}$$

- We'll use **squared loss** to evaluate each data point:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left(g^{(i)} - y^{(i)} \right)^2 \tag{2.27}$$

- We use our **hypothesis** $h(x^{(i)}; \Theta)$ to make our guess $g^{(i)}$.

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left(h(x^{(i)}; \Theta) - y^{(i)} \right)^2 \tag{2.28}$$

- Our hypothesis is a **linear model** $\theta^T x^{(i)} + \theta_0$.

Key Equation 32

The **ordinary least squares objective function** for **linear regression** is written as

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left((\theta^T x^{(i)} + \theta_0) - y^{(i)} \right)^2$$

If we break this into parts:

$$J(\theta, \theta_0) = \underbrace{\frac{1}{n} \sum_{i=1}^{n}}_{\text{Averaging}} \left(\underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}} \right)^2 \tag{2.29}$$

Now, this is an **optimization** problem. We need to find the model (θ, θ_0), that gives us the best (minimal) J.

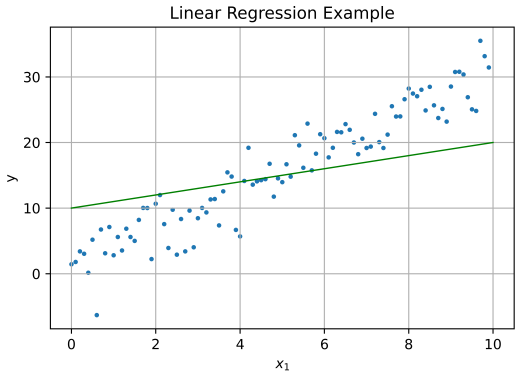
$$\theta^*, \theta_0^* = \arg\min_{\theta, \theta_0} J(\theta, \theta_0)$$

We now have two parameters in our argmin function, but aside from listing both of them, the notation is the same. We just substituted $\Theta = (\theta, \theta_0)$

2.3.7 Visualizing our Model

We'll start with the **one-variable** case. With one input, one output, we use a 2D plot to graph our data.

- Each piece of data is a simple (x, y) pair: a **point**.
- Meanwhile, our hypothesis is a **line**: for each x, it predicts a different y.



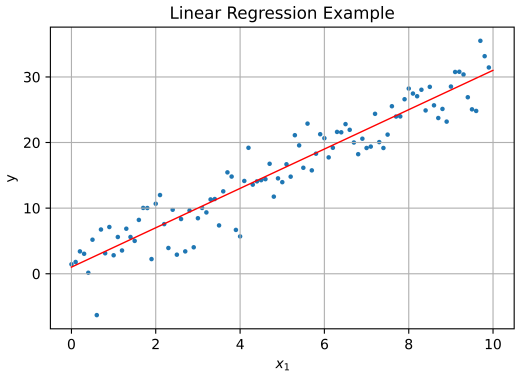
This linear model doesn't fit our data very well: $(\theta_0 = 10, \theta_1 = 1)$

How do we know our line doesn't fit our data? Because it isn't "close" to the shape of the data.

We want our model to really represent the data it comes from: they should look similar.

We're trying to get our line as **close as possible** to the points, hoping to find a linear pattern.

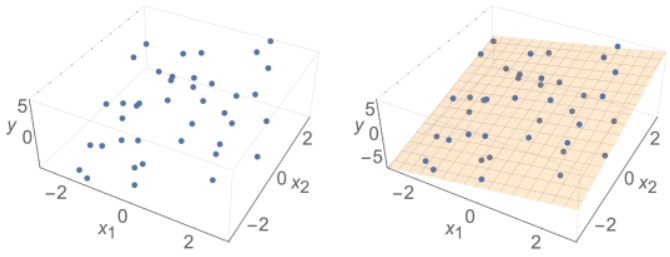
- We call this "**fitting**" our line to the data.



This line is much better fitted to the data: $(\theta_0 = 1, \theta_1 = 3)$

What does this look like if we have **two variables**? You need a 3D space, with 2 dimensions for the input.

- Each piece of data is a **point** (x_1, x_2, y).
- Our hypothesis is a **plane**: for each pair (x_1, x_2), it predicts a different y.



This plane is **fitted** the same way our line was. Notice that y is our **height**: this is the **output** of our regression.

Earlier, we mentioned that we can't really **visualize** higher dimensions.

Looking at a 4D "plane" would be a headache.

- So, instead, we don't even try to. We'll think of them in terms of our math. When we need intuition, we'll rely on the 2D plane.
- Because they're a higher-dimensional version of a **plane**, we call it a **hyperplane**.

Definition 33

A **hyperplane** is a **higher-dimensional version** of a **plane** - a **flat** surface that continues on forever.

We use it to represent our **linear** hypothesis for the purpose of **regression**.

- We have d dimensions (d variables) in our input.
- To represent our output, we need one additional, $(d+1)^{\text{th}}$ dimension.

Visually, the "**height**" of our plane represents the **output** of $h(x)$.

- Our line was a **1-D** object in a **2-D** plane.
- Our plane was a **2-D** object in a **3-D** space.

So, our **hyperplane** is a d dimensional object in a $d+1$ dimensional space.

- Our goal is the same: we want our **hyperplane** to be as **close** to all of our data points as it possibly can.

2.3.8 Another Interpretation

So far, we've generally interpreted our model similarly to $mx + b$.

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_d x_d \tag{2.30}$$

- Using our $mx + b$ analogy, we can see θ_k as the "**slope**" of x_k.

$$\frac{\partial h}{\partial x_k} = \theta_k \tag{2.31}$$

- In other words, θ_k tells us how much x_k affects the **output**.

Concept 34

The **larger** $\|\theta_k\|$ is, the **more important** x_k is to our output.

- If we **increase** $\|\theta_k\|$, then x_k will have a **bigger effect** on $h(x)$.

$$\overbrace{\frac{\partial h}{\partial x_k}}^{\text{Effect of } x_k \text{ on } h} = \theta_k$$

This is a simple **pattern**: "as we change x_k, we change $h(x)$".

We can also **compare** each θ_k term to each other.

- If θ_2 is larger than θ_1, then increasing x_2 would affect the output **more** than increasing x_1.

$$h(x) = 2x_1 + \overbrace{10000x_2}^{x_2 \text{ has greater effect}} \tag{2.32}$$

- We could say that x_2 has a stronger effect on the output than x_1: it **weighs more heavily** in the calculation.

Because of this, we sometimes call θ_k the **weight** for x_k.

Definition 35

A **weight** is a **parameter** that tells us how **strongly** a variable influences our **output**.

It is usually a **scalar** that we **multiply** by our variable.

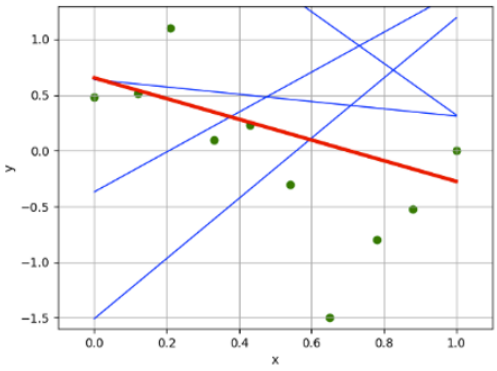
- We say that θ_0 is an "offset", and the other θ_i terms are "weights".

2.4 The stupidest possible linear regression algorithm

So, now we want to try to **optimize** J based on θ and θ_0. How do we do that? Let's start as **simple** as we possibly can.

We can't try **every** possible Θ, because there are an **infinite** number of them. Rather than thinking too hard about a possible pattern, or an **algorithm**, let's just **randomly** try options.

We'll try **random** values for θ and θ_0, and **pick** whichever option gives us the **best result**. Seems simple, if inefficient.



Each line (hypothesis) was randomly generated. We focus on the red one: this is the best model, out of all the ones that we tried.

Why introduce such a silly algorithm? For a few reasons:

- It gives us an **example** of an optimization algorithm that's very **simple**.
- **Randomly** generated results create a good **baseline** - more intelligent algorithms can be compared to this one, to see how well we're doing.

Sometimes, you might come up with a clever technique, only to find out it isn't better than a random model! It happens more than you'd think.

2.5 Analytical solution: ordinary least squares

We can do **better** than randomly **generate** parameters, though. In fact, in this rare case, we can actually **solve** for optimal parameters!

2.5.1 Trying to Simplify

Our solution will involve a lot of algebra. Because of that, it's worth it to **simplify** our formula as much as possible beforehand.

By algebra, we just mean "shuffling around math symbols in an equation".

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left((\theta^T x^{(i)} + \theta_0) - y^{(i)} \right)^2 \tag{2.33}$$

Most parts of this equation can't really be **simplified**: y and x are just variables, and we can't do anything with the **sum** without knowing our data points.

- But, there's one notable detail: we **separated** θ_0 from our other θ_k terms.
- If we can **include** θ_0 in the dot product, our math will be easier.

2.5.2 Combining θ and θ_0

Let's go back to our **original** equation for $(\theta^T x + \theta_0)$, before we switched to **vectors**.

We drop the $^{(i)}$ notation whenever it isn't necessary, to de-clutter the equations.

Let's just say we've picked one random data point.

$$h(x) \quad = \quad \theta_0 + \theta \cdot x \quad = \quad \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_d x_d \tag{2.34}$$

We simplified our notation with a **dot product**: each θ_k term is **multiplied** by an x_k term.

- This is, of course, **excluding** θ_0.
- We would end up with a simpler result if we could include θ_0 in the θ vector. But, θ_0 would need to be **multiplied by an x_0 term**.

$$h(x) = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \overbrace{\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}}^{\text{We need } x_0} \tag{2.35}$$

We have a trick: let's factor out $x_0 = 1$.

You can always factor out 1 without changing the value!

$$h(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_d x_d \tag{2.36}$$

So, this means we just have to **append** a 1 to our vector x. At the **same time**, we'll append θ_0 to θ!

$$x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}, \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix}, \qquad h(x) = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \tag{2.37}$$

We'll write that symbolically, and then apply a transpose.

$$h(x) = \theta \cdot x = \theta^T x \tag{2.38}$$

Concept 36

Sometimes, to simplify our algebra, we can **append** θ_0 to θ.

To make this possible, we **choose** $x_0 = 1$.

- This requires **appending** a value of 1 to x.

Once we do this, we can **write**

$$h(x) = \theta^T x$$

We **have** to append this 1 to every single $x^{(i)}$ in order for this to **work**. But, now we can treat our **parameters** as **one vector**.

2.5.3 Combining data points

There's another place we can clean things up:

- Currently, when using our **objective** function, we have to **sum** over **every** single data point.

Note that, for convenience, we've included θ_0 in θ.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2 \tag{2.39}$$

- This is a bit of a hassle - we have to consider every $(x^{(i)}, y^{(i)})$ term separately.
- Is there a better way?

We've solved this kind of problem before: using vectors above, we were able to work with **many parameters** θ_k and **many variables** x_k at the same time.

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_d x_d \xrightarrow{\text{vector form}} h(x) = \theta^T x$$

- This made it easier to do lots of math quickly.

Concept 37

One of the biggest benefits of **matrices** is being able to do **lots of math at the same time**.

- In particular, **matrix multiplication** allows you to do **addition** and **multiplication** on as many elements as you want.

Can we do the **same** here - combining **many data points** into one object?

2.5.4 Many data points in a matrix

We want to combine all of our data points into a single matrix.

- We're already using **rows** to represent **multiple dimensions** of a data point.

A reminder: a "dimension"(feature) is one aspect of our input data. For an animal, it might be height, weight, age, etc.

Each dimension stores one piece of information.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \Bigg\} \; k^{th} \text{ dimension in row } k \tag{2.40}$$

- We'll need different notation to separate **data points**: we'll use **columns**.

$$X_{1D} = \overbrace{\begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \cdots & x^{(n)} \end{bmatrix}}^{i^{th} \text{ data point in column } i} \tag{2.41}$$

Definition 38

We use **rows** to indicate the different **dimensions** x_k of a single data point, and **columns** to indicate each **data point** $x^{(i)}$.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \qquad X_{1D} = \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(n)} \end{bmatrix}$$

- Note that the capitalized X matrix is used for **all of our data points** $x^{(i)}$.

These formats are both useful, but limited:

- x can handle **many dimensions**, but represents **one data point**.
- X_{1D} represents **many data points**, but with only **one dimension** for each.

Our solution? Combine them into a single object:

Key Equation 39

X is our **input matrix** in the shape $(d \times n)$ contains information n **data points** with d **dimensions each**.

$$X = \overbrace{\begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \cdots & x_d^{(n)} \end{bmatrix}}^{n \text{ data points}} \Bigg\} d \text{ dimensions}$$

Example: Consider 3 data points in 2 dimensions: $[1,2]^T$, $[9,5]^T$, $[10,11]^T$.

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 9 \\ 5 \end{bmatrix}, \begin{bmatrix} 10 \\ 1 \end{bmatrix} \longrightarrow X = \begin{bmatrix} 1 & 9 & 10 \\ 2 & 5 & 1 \end{bmatrix} \tag{2.42}$$

If we want to replace $\theta^T x + \theta_0$ with $\theta^T x$, we can include 1's at the top: _____ Or the bottom, if we want.

$$X = \overbrace{\begin{bmatrix} 1 & \cdots & 1 \\ x_1^{(1)} & \cdots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \cdots & x_d^{(n)} \end{bmatrix}}^{n \text{ data points}} \Bigg\} d+1 \text{ dimensions} \tag{2.43}$$

We can do the same for Y: combine all of the data points into one matrix.

Key Equation 40

Y is our **output matrix** in the shape $(1 \times n)$ that contains all data points.

$$Y = \begin{bmatrix} y^{(1)} & \cdots & y^{(n)} \end{bmatrix}$$

Why is this a row vector, not a matrix?

This is a **regression** problem: each output is a scalar, not a vector!

2.5.5 Objective Function in matrix form

Now that we can use all of our **data points** at the same time, we can condense our objective function.

$$J(\Theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2 \tag{2.44}$$

We can simultaneously compute $(\theta^T x^{(i)} - y^{(i)})$ for all of our data points at the same time:

$$\theta^T X - Y \quad = \quad \begin{bmatrix}\theta^T x^{(1)} - y^{(1)} & \theta^T x^{(2)} - y^{(2)} & \cdots & \theta^T x^{(n)} - y^{(n)}\end{bmatrix}$$

We could show that this matrix multiplication gives the results we want, with calculation.

But this is a little tedious, and doesn't teach us much, so we recommend trying it yourself if you're unconvinced.

This isn't exactly what we want, though: we want $(\theta^T x^{(i)} - y^{(i)})^2$: multiplied by itself. We can do this with a **dot product**:

$$\left(\theta^T x^{(i)} - y^{(i)}\right)^2 \quad \longrightarrow \quad \left(\theta^T X - Y\right)\cdot\left(\theta^T X - Y\right) \tag{2.45}$$

How do we write this dot product with matrix multiplication?

Clarification 41

When a and b were **column vectors**, we could take their dot product as $a^T b$.

$$a = \begin{bmatrix}a_1\\a_2\\\vdots\\a_m\end{bmatrix} \quad b = \begin{bmatrix}b_1\\b_2\\\vdots\\b_m\end{bmatrix} \quad \Longrightarrow \quad a \cdot b = a^T b$$

But we have to do the opposite for **row vectors** p and q: we get pq^T.

$$p = \begin{bmatrix}p_1 & p_2 & \cdots & p_m\end{bmatrix}$$
$$q = \begin{bmatrix}q_1 & q_2 & \cdots & q_m\end{bmatrix} \quad \Longrightarrow \quad p \cdot q = pq^T$$

Here, we're comparing $a^T b$ to ab^T.

Because $(\theta^T X - Y)$ is a **row vector**, we'll have to write it in the pq^T format:

$$\left(\theta^T x^{(i)} - y^{(i)}\right)^2 \quad \longrightarrow \quad \left(\theta^T X - Y\right)\left(\theta^T X - Y\right)^T \tag{2.46}$$

This formula represents our original objective function:

$$\frac{1}{n}\left(\theta^T X - Y\right)\left(\theta^T X - Y\right)^T = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2 \tag{2.47}$$

Key Equation 42

Using X, Y, and θ can write our **objective function** for **multiple** variables and **multiple** data points as

$$J(\theta) = \frac{1}{n}\left(\theta^T X - Y\right)\left(\theta^T X - Y\right)^T$$

Because n is a constant, we sometimes ignore it when we're minimizing $J(\Theta)$.

It is important to **remember** the **shape** of our objects, as well.

Concept 43

Our matrices have the shapes:

- X: $(d \times n)$ - matrix
- Y: $(1 \times n)$ - row vector
- θ: $(d \times 1)$ - column vector
- θ_0: (1×1) - scalar
- J: (1×1) - scalar

If we combine θ_0 into θ, replace every use of d with $d+1$.

Notice that these shapes make sense for our above equation! Try working through the matrix multiplication to verify this.

These shapes are worth **memorizing**.

2.5.6 Alterate Notation

One side problem: some ML texts use the **transpose** of X and Y.

Notation 44

Some subjects use **different notation** for **matrices**. The main difference is that X and Y use their **transpose**, which we'll notate as

$$\tilde{X} = X^T \qquad \tilde{Y} = Y^T$$

Thus, our equation above becomes

$$J = \frac{1}{n}\left(\tilde{X}\theta - \tilde{Y}\right)^T\left(\tilde{X}\theta - \tilde{Y}\right)$$

2.5.7 Optimization in 1-D - Using Calculus

Now that we've sorted our data, we can start **optimizing** θ. Our goal is to modify θ, to find the minimal J.

We're gonna focus on one data point/dimension at a time, so we don't need our matrix notation.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2 \tag{2.48}$$

We'll start with a simplified case, and build our way up:

- First, we limit our attention to one data point ($n = 1$):

$$J(\theta) = (\theta^T x - y)^2 \tag{2.49}$$

- And we'll assume θ and x are one-dimensional ($d = 1$).

$$J(\theta) = (\theta x - y)^2 \tag{2.50}$$

If we use θ as a **variable**, this is an ordinary single-variable function! How would we find the **minimum**?

- Using **calculus**! Anywhere there's a local **minimum**, we typically know the **derivative is 0**.

Assuming a "smooth" surface...

Concept 45

If our function $J(\theta)$ has **one variable**, we find possible **local minima** θ wherever the **derivative** $\partial J/\partial\theta$ is zero.

$$\frac{\partial J}{\partial \theta} = 0$$

- To make sure it's a minimum (not a maximum), we also need to make sure that the **second derivative** is positive ($J''(\theta) > 0$).
- We already know this is true for **squared loss**, so we won't bother with this step.

Why do we need $\partial J/\partial\theta = 0$?

If $\partial J/\partial\theta > 0$, then decreasing θ slightly would reduce J: there's a lower point nearby, so this isn't a minimum!

If $\partial J/\partial\theta < 0$, increasing θ has the same effect.

- Note that we're taking $\frac{\partial}{\partial\theta}$, **not** $\frac{\partial}{\partial x}$.
- This is because our goal is to modify θ (our model), not x (our data).

We want to train our model to match our data, not the other way around!

Let's do this for our simple example:

$$J'(\theta) = 2x(\theta x - y) = 0 \tag{2.51}$$

We just find where the slope $J'(\theta)$ is 0, and solve for θ!

Because this is the optimal θ, we call it θ^*.

$$\theta^* = \frac{y}{x} \tag{2.52}$$

2.5.8 Optimizing for multiple variables

This time, we'll do **one data point**, having d **dimensions**.

- This gets a bit tricky, because we have to do our math with **vectors**.

Because we only have one data point, we'll omit the $^{(i)}$ notation.

$$J(\theta) = (\theta^T x - y)^2 \tag{2.53}$$

We'll **optimize** this. In the **one-dimensional** case, we wanted to set the **derivative** of J to **zero**, using a single θ variable.

- Now, we have **multiple variables** θ_k that we could optimize.
- How do we know when we've optimized all of them?

Well, if we consider each dimension separately, θ_k would be optimized if

$$\frac{\partial J}{\partial \theta_k} = 0 \tag{2.54}$$

So, maybe it would be reasonable to just set **every** derivative to **zero**?

- It turns out, the answer is **yes**!

We can use the reasoning that we used for 1D:

If one of our derivatives $\partial J/\partial \theta_k > 0$, then we could decrease θ_k to find a nearby point which is "lower" than our current point: we don't have a minimum.

Thus, every derivative must be 0.

If every individual derivative $\partial J/\partial \theta_k$ is zero, we have a potential minimum.

Concept 46

If our function $J(\theta)$ has $d+1$ **parameters**, we find possible **local minimum** θ anywhere that obeys the system of equations

$$\frac{\partial J}{\partial \theta_0} = 0 \qquad \frac{\partial J}{\partial \theta_1} = 0 \qquad \frac{\partial J}{\partial \theta_2} = 0 \qquad \dots \qquad \frac{\partial J}{\partial \theta_d} = 0$$

Or in general,

$$\frac{\partial J}{\partial \theta_k} = 0 \qquad \text{for all } k \text{ in } \{0, 1, 2, \dots, d\}$$

- Why $d+1$ parameters instead of d? Because we're including θ_0 as one additional parameter.

Again, we ignore the second requirement of making sure this isn't a **maximum** or **saddle** point.

The **solution** to this system of equations will be our **desired list of parameters**, θ^*.

$$\theta^* = \begin{bmatrix} \theta_1^* \\ \theta_2^* \\ \vdots \\ \theta_d^* \end{bmatrix} \tag{2.55}$$

2.5.9 Gradient Notation

Above, we wrote our derivative **element-wise**: each derivative got its own equation.

- We can make things easier by storing our derivatives in a **column vector**, just like how θ stores θ_k terms.

How do we know that this is a column vector, and not a row vector?

Check out the matrix derivative notes for a complete explanation.

$$\frac{\partial J}{\partial \theta} = \begin{bmatrix} \partial J/\partial \theta_0 \\ \partial J/\partial \theta_1 \\ \vdots \\ \partial J/\partial \theta_d \end{bmatrix} \tag{2.56}$$

We'll think of this as a bigger, **multivariable** derivative, called the **gradient**.

We'll give more conceptual intuition for the gradient in the next chapter. Look forward to it!

Key Equation 47

The **gradient** of J with respect to θ is

$$\nabla_\theta J = \frac{\partial J}{\partial \theta} = \begin{bmatrix} \partial J/\partial \theta_0 \\ \partial J/\partial \theta_1 \\ \vdots \\ \partial J/\partial \theta_d \end{bmatrix}$$

It has the same dimensions $(d \times 1)$ as θ.

Why is $\partial J/\partial \theta$ a $(d \times 1)$ matrix instead of $(1 \times d)$?

We don't have a special reason: it's a convention we've picked, that works well with the rest of our math.

In fact, some other texts might do differently. To learn more about our rules, check the Matrix Derivatives notes.

Now, we can rewrite our previous rule, "every derivative is 0":

$$\nabla_\theta J = \begin{bmatrix} \partial J/\partial \theta_0 \\ \vdots \\ \partial J/\partial \theta_d \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} = \vec{0} \tag{2.57}$$

Concept 48

If θ is a **vector**, we can find possible **local minima** θ of $J(\theta)$ anywhere that the **gradient** $\nabla_\theta J$ is zero.

$$\nabla_\theta J = \vec{0}$$

Once again: we should check if it's a minimum. But we continue to ignore this caveat.

This is the general equation we **solve** to find θ^*.

2.5.10 Matrix Calculus

Now, we return to the general case: n **data points**, each having d **dimensions**.

$$J(\theta) = \frac{1}{n}\left(\theta^T X - Y\right)\left(\theta^T X - Y\right)^T$$

Using the "alternate" notation":

$$J = \frac{1}{n}\left(\tilde{X}\theta - \tilde{Y}\right)^T\left(\tilde{X}\theta - \tilde{Y}\right) \tag{2.58}$$

We will **not** show how to take this matrix derivative. But our result is:

Want to know how? Check out A.4 in the official appendix.

$$\nabla_\theta J = \frac{2}{n}\tilde{X}^T\left(\tilde{X}\theta - \tilde{Y}\right) = 0 \tag{2.59}$$

Clarification 49

Note that matrix derivatives often look **similar** to traditional, single-variable derivatives. However, they are **not the same**.

- Often, this can result in **shape errors**: we end up with the wrong matrix shape.

How do we take a matrix derivative in general? Check out an explanation in the Matrix Derivatives chapter.

From here, we just solve for θ, using matrix multiplication rules.

Key Equation 50

The **solution** for **OLS optimization** is

$$\theta = \underbrace{\left(\tilde{X}^T \tilde{X}\right)^{-1}}_{d \times d} \underbrace{\tilde{X}^T}_{d \times n} \underbrace{\tilde{Y}}_{n \times 1}$$

Or, in our **original** notation,

$$\theta = \underbrace{\left(X X^T\right)^{-1}}_{d \times d} \underbrace{X}_{d \times n} \underbrace{Y^T}_{n \times 1}$$

- If θ_0 is included in θ, then dimension d is replaced with $d + 1$.

Note that this requires that XX^T is invertible: we need to compute that inverse $(XX^T)^{-1}$ in the equation, after all.

This technique doesn't work if that's not the case. But, we'll introduce a different technique to solve this problem: regularization!

We've finished with "ordinary least squares".

- We call it "ordinary" because this is the **simple** version of the problem.
- We'll make it more complex by introducing **regularization**.

2.6 Regularization

The above solution gives us the **best** model for matching our **training data**.

- But earlier, we mentioned that we want our model to be able to **generalize** to new data.

Because our training data doesn't perfectly reflect all of our future data.

We need some math to represent this goal of "generality".

- This equation won't measure our performance on training data, but instead encourages us to do well on **future data**.

We call this type of function a **regularizer**.

Definition 51

A **regularizer** is a term to our **objective function** that helps measure how **general** our hypothesis is.

- By **optimizing** this term, we hope to create a model that works better with **new data** we didn't train with.

This function takes in our **vector of parameters** Θ as an input: $R(\Theta)$.

But how do we make a model "more general?"

- First, we need to **understand** the problem: **what's wrong** with only using our training data?

2.6.1 Coincidences, and fake patterns

The goal of our model θ is to look for **patterns**. This is a concept we discussed earlier (section 2.3.8):

Concept 52

(Review)

The **larger** $\|\theta_k\|$ is, the **more important** x_k is to our output.

- If we **increase** $\|\theta_k\|$, then x_k will have a **bigger effect** on $h(x)$.

$$\overbrace{\frac{\partial h}{\partial x_k}}^{\text{Effect of } x_k \text{ on } h} = \theta_k$$

This is a simple **pattern**: "as we change x_k, we change $h(x)$ by this much".

θ is trying to identify **which variables** have an effect on our output, and by how much.

- If $\|\theta_k\|$ is large, our models thinks that x_k is important to our output.

$\theta_k < 0$ doesn't mean that x_k doesn't matter: it just means that it affects $h(x)$ by decreasing it, instead of increasing it.

Our goal is to modify each θ_k term until it matches the real distribution.

But seeing a pattern (x_k affecting $h(x)$) doesn't mean that it's real: **random chance** can cause us to see patterns that don't really exist.

- **Example:** You take 20 quizzes during a semester. Every time you did **well**, you happened to be wearing a **red shirt**.
- You might come to believe that red shirts help you study **better**, even though it was just luck.

Isn't this kind of coincidence a bit unlikely? Maybe.

- But what happens if we have 10 possible coincidences? It's less likely that we avoid all of them.
- In other words: the more **opportunities** there are for something rare to happen, the **more likely** it is to happen.

Suppose the chance of one coincidence is p.

The chance of **one** coincidence not occurring is $(1-p)$. The chance of **ten** coincidences not occurring is $(1-p)^{10}$.

As we get more chances, we're more likely to see a coincidence.

Concept 53

The more **patterns** we're looking for, the more likely that **at least one** of the them shows up by **coincidence**.

- Our data can, by chance, match a pattern that **isn't even real**.

For some entertaining examples, search the phrase "spurious correlation".

This really becomes a problem for our model: often, we **don't know** what data matters, so we include **everything** we possibly can.

- But the more data we include, the more likely that something looks **important** on **accident**!

This is an interesting situation, where **learning more** about our training data can cause our model to perform **worse**: we're **overfitting**.

Learning more about our training data isn't necessarily the same as learning more about the true distribution that data came from!

Definition 54

(Review from Introduction chapter)

Overfitting occurs whenever we **learn** ("fit") our training data too exactly, and it causes problems when we see **new data**.

- Often, we **memorize** very specific patterns, that don't actually hold up in general: they only appeared in our randomly sampled training data.

Example: You flip a coin 10 times, and it comes up heads 8 times.

- You've decided that the coin has an 80% chance of coming up heads.
- But it's a fair coin: the 'training data' (10 coin flips) doesn't match the 'true distribution' (50% chance of heads).

Now, we understand our problem. Let's come up with a solution.

2.6.2 Ridge Regression

We're worried about our model **finding false patterns** based on weak evidence.

- If our model finds a **pattern**, then it'll increase $\|\theta_k\|$: it considers x_k to be important for predicting $h(x)$, because the data coincidentally makes it **look** important.

If our model is "too eager" to find patterns, we can make it **more skeptical** of patterns it might see.

It still need to look for *some* (hopefully real) patterns, so that it can make good predictions $h(x)$.

- In other words, we'll **punish** our model for increasing $\|\theta_k\|$ too easily.

We'll actually use θ_k^2, for the same reasons we use squared loss:

Smoother, works well for positive and negative θ_k, etc.

Concept 55

We want to **discourage** our model from prematurely deciding that x_k has a **effect** on $h(x)$.

- Thus, we'll **discourage** our model from increasing $\|\theta_k\|$: θ_k represents this "**effect**".

$$R(\theta_k) = \theta_k^2$$

- Our algorithm will **minimize** $R(\Theta)$, so we'll make this θ_k^2 **small**.

We can repeat this process for all of our θ_k terms, and combine them into a vector θ:

$$R(\Theta) = \sum_k \theta_k^2 \quad = \quad \|\theta\|^2 = \theta \cdot \theta$$

This is our **ridge regression** model.

Why "ridge regression"? We'll get into that later.

Key Equation 56

Our **regularizer for regression** will be given by **square magnitude** of θ:

$$R(\Theta) = \|\theta\|^2 = \theta^T\theta$$

- In this model, we are biasing $\|\theta\|$ towards 0: the farther from 0 we are, the more we punish the model.

This approach is called **Ridge Regression**.

2.6.3 λ, our regularization constant

We can now create an "**objective function**" that includes training loss, **and** regression.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}}\right)^2 + \underbrace{\|\theta\|^2}_{\text{Regularizer}} \tag{2.60}$$

Notice that these terms "compete" with each other:

Concept 57

Our **training loss** and **regularizer** compete with one another, affecting our model in opposing ways:

- Our regularizer wants us keep θ small, being more **suspicious** of predicting patterns.
- But in order to make good predictions for our **training data**, we need to find the **real** patterns: some θ values need to be bigger, to predict our data.

This is why we don't end up with model $\theta = \vec{0}$: while this model would have low **regularization** $R(\Theta)$, it'll have high **training loss**.

We need a **balance** between training loss and regularization. It would be useful to have a way to **control** that balance.

- That way, we can decide, "how much do we care about matching training data, versus avoiding coincidental patterns?"

We have a tool for this: we'll represent "how much we care about **regularization**" with a **constant** λ.

Definition 58

Lambda, or λ, is the constant ($\lambda \geqslant 0$) we **scale** our **regularizer** by.

$$\text{Total Regularization} \quad = \quad \lambda R(\Theta) \quad = \quad \lambda ||\theta||^2$$

It represents **how strongly** we want to regularize: the larger it is, the more strongly we try to **generalize** our model.

Why is $\lambda \geqslant 0$?

Clarification 59

We keep $\lambda \geqslant 0$, because $\lambda < 0$ would encourage our model to make $||\theta||$ **bigger**, no matter what.

- There's a limit to how small $|\theta|$ can be, but there's **no limit** on how big it can be: it would just keep getting bigger forever.

Finally, we have our **completed** objective function:

Key Equation 60

The **objective function** for **ridge regression** is given as

$$J(\Theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}}\right)^2 + \underbrace{\lambda\|\theta\|^2}_{\text{Regularizer}}$$

Readers might catch that our regularizer doesn't include θ_0. There's a good reason for that! We'll discuss it below.

Once again, we note the contrast between "training loss" and "regularization".

- We've only made one change: λ can increase or decrease our focus on regularization.

This λ is crucial to our model: the larger it is, the more we punish our model for increasing $\|\theta\|$.

Concept 61

The more **regularization** (large λ) we have, the more we're **focused** on keeping $|\theta|$ small.

- If we make λ **too big**, then $\|\theta\|$ becomes **very small**: our model doesn't learn enough information.
- If we make λ **too small**, then $\|\theta\|$ becomes **very big**: our model learns **all** the patterns of our data, even the ones that come from random noise.

2.6.4 Why not regularize θ_0?

Note that when we regularize with $\lambda\|\theta\|^2$, we're **not including** θ_0 in our vector:

$$R(\Theta) = \lambda\theta^T\theta \qquad \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \tag{2.61}$$

This suggest that we're **not regularizing** θ_0. Why is that?

Concept 62

We **do not regularize** θ_0.

- We **allow** θ_0 to take whatever value fits best.

The first reason: θ_0 works **differently** from our other θ_k terms.

- A term θ_k tell us how **important** one variable x_k is to our output $h(x)$.
- θ_0 works almost the opposite way: it **ignores** our input x, and gives a baseline output.
  - In other words: if we **remove** the effect of all of our variables ($x = \vec{0}$), what do we expect to see?

If our input data is centered on $x = \vec{0}$ (equally above and below on each axis), θ_0 is the **average** output we expect to see.

Naturally, regularization has very different effects:

- If you regularize $\|\theta_k\|$, your model i will emphasize the **effect** of x_k by a smaller amount.
- If you regularize $\|\theta_0\|$, you've just **shifted** every output, by the same amount.

Concept 63

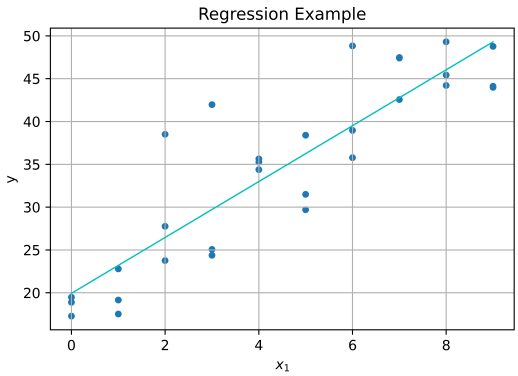
Unlike θ_k, θ_0 doesn't find **patterns** in the data: it just tells us roughly how **large** our output is.

- In fact, $\|\theta_0\|$ affects all of our data in the **exact same way**.
- Decreasing $\|\theta_0\|$ would shift all outputs equally: this doesn't do anything to make it more **accurate** for future data.

Because of this difference, it's not necessary to **regularize** θ_0.

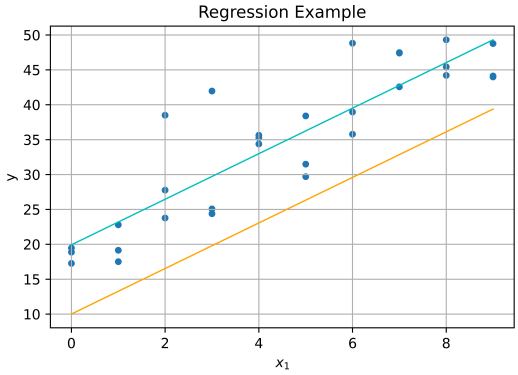
Another reason we don't regularize θ_0 is that it's often **necessary** for good predictions. This is best shown with a **visual** example.

- Let's take an example with one input, x_1. So, we have a **linear** function: $h(x) = \theta_1 x_1 + \theta_0$.



Our regression example.

Let's suppose we **regularize**, or shrink, our offset θ_0, while keeping everything else the same:



Reducing our offset pulls our line further away from all of our data! This is a serious problem.

This shows that we **need** our offset!

- We use it to **shift** our hyperplane around the space: otherwise, otherwise, it's difficult to fit data **far** from the origin.

Imagine that we have three data points at $(1, 1001)$, $(2, 1002)$, and $(3, 1003)$.

It's clear that our data is offset by roughly 1000: we need θ_0 to address this.

Concept 64

θ_0 gives us a baseline for the **size** of our output values.

- If we reduce θ_0, all of our outputs will be reduced equally: they'll be **less accurate.**

2.6.5 Ridge Regression Solution

Now, we have our regression loss function,

$$J(\Theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}}\right)^2 + \underbrace{\lambda\|\theta\|^2}_{\text{Regularizer}} \tag{2.62}$$

Which we can express in matrix form:

$$J(\theta) = \frac{1}{n}\left(\theta^T X - Y\right)\left(\theta^T X - Y\right)^T + \lambda\theta^T\theta \tag{2.63}$$

We can now **optimize** this for θ. We'll do some matrix calculus, omitting the steps here:

$$\nabla_\theta J = \frac{2}{n}\tilde{X}^T\left(\tilde{X}\theta - \tilde{Y}\right) + 2\lambda\theta = 0 \tag{2.64}$$

Finally, we **solve** (using some linear algebra – multiplying by inverses, distributive property, add/subtracting, etc.):

We're gonna cheat a bit, and ignore θ_0.

One way to do this is to subtract a constant from every value in Y, so that the data is centered on $y = 0$: you don't *need* an offset θ_0.

But in some problems, we might just include θ_0 in θ, to make our problem simpler, even though we're accidentally regularizing θ_0.

Key Equation 65

The **solution** to the **ridge regression** problem allows us to find the **optimal** model θ^*.

$$\theta^* = \left(\tilde{X}^T\tilde{X} + n\lambda I\right)^{-1}\tilde{X}^T\tilde{Y}$$

or, in our original notation:

$$\theta^* = \left(XX^T + n\lambda I\right)^{-1}XY^T$$

Where I is the $(d \times d)$ identity matrix.

Review: an identity matrix is a square matrix with 1's on its diagonal, and 0's everywhere else.

In general, $AI = IA = A$.

2.6.6 Invertibility

This solution is great! We just have one problem:

- It requires that our **inverse** of $(XX^T + n\lambda I)^{-1}$ exists.

Thankfully, this is true so long as $\lambda > 0$!

Concept 66

If $\lambda > 0$, we can be sure that $(XX^T + n\lambda I)$ has an **inverse**.

- And thus, we have a **unique solution** to θ for our **ridge regression problem.**

You **do not need to know** the linear algebra that justifies this statement. You just need to know that it's true.

The short version of the justification is:

XX^T is positive semi-definite: $v^T XX^T v \geqslant 0$.

If you add positive elements on the diagonal ($A = XX^T + n\lambda I$), then you can only increase $v^T Av$: now it's $v^T Av > 0$.

Thus, $(XX^T + n\lambda I)$ is positive definite: this means it has an inverse.

If that doesn't make any sense, don't worry about it.

This, by the way, presents one more justification for regularization:

Concept 67

Another benefit of **regularization** is that we can **always** find an **analytical solution** for θ.

- $\left(\tilde{X}^T\tilde{X} + n\lambda I\right)$ is invertible, thus, we can compute

$$\theta^* = \left(XX^T + n\lambda I\right)^{-1} XY^T$$

Sometimes, we call non-invertible matrices **singular**.

Definition 68

All of the following statements about square matrix A are equivalent:

- $\det(A) = 0$.
- A has **no inverse**.
- A is **singular**.
- A is **not full rank**.

A few more definitions of singular:

- A has at least one eigenvalue of 0.
- A has linearly dependent rows and columns.
- $Ax = 0$ has a solution other than $x = 0$.

Sometimes, you say this as, "$Ax = 0$ has a non-trivial solution".

2.6.7 Uniqueness of θ^*

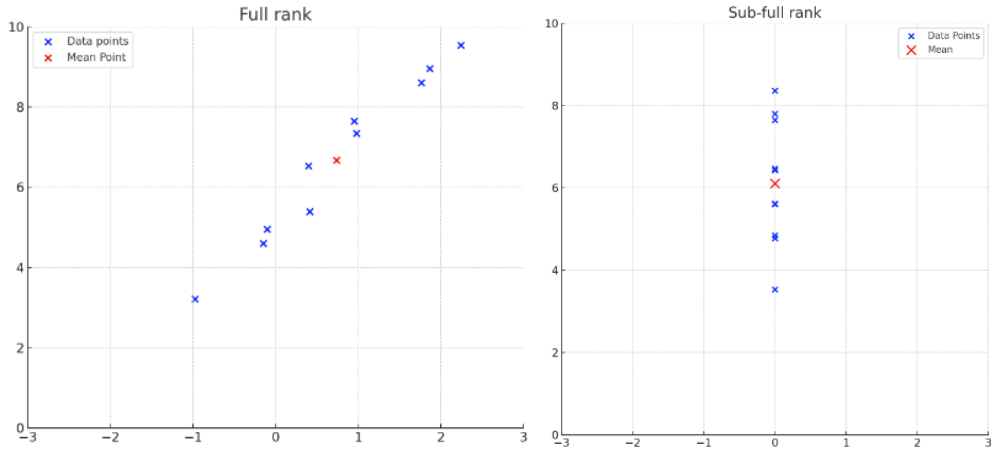
This seems suspicious: why on earth would **regularizing** θ allow us to find a solution?

- In order to understand this, we need to understand **what happens** when XX^T is **singular**.

If XX^T isn't full rank, that means that X **isn't full rank**, either.

Showing that this is true is more a linear algebra problem than an ML problem, so we'll defer it here.

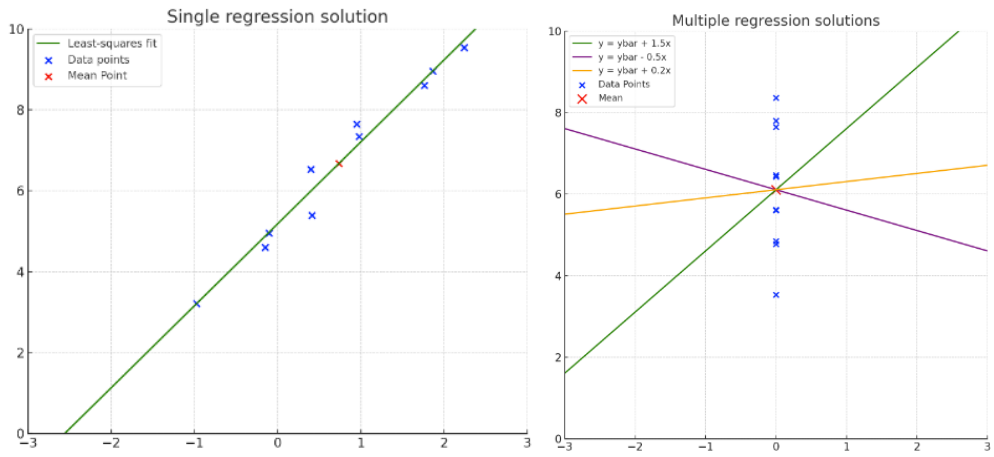
What does this mean? Let's consider an example in 1d: we'll compare a "full rank" version, to one that isn't full-rank.



Typically (left plot), we expect our data to be **full rank**: our input space is 1-D, our input data occupies 1-D.

But sometimes (right plot), we might have data which is **less than full rank**: in this case, the input data is 0-D: only occupying $x = 0$.

Sure, this data looks weird, but why is this problematic? Because it creates **multiple optimal solutions**:



Our "full rank" (left plot) data has one optimal solution.
Our "sub-full rank" (right plot) data has many!

This is the real reason why, if XX^T isn't invertible, we can't find an analytical solution: there are actually **many** possible solutions!

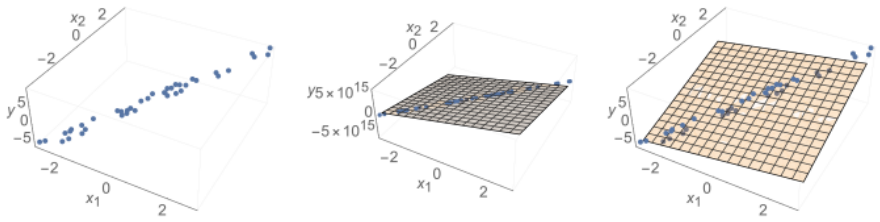
Concept 69

If XX^T isn't invertible, we **cannot** use our formula to find an analytical (formula-based) **solution** for θ.

- That's because there are **many optimal solutions**.
- In fact, there are **infinitely many of them!**

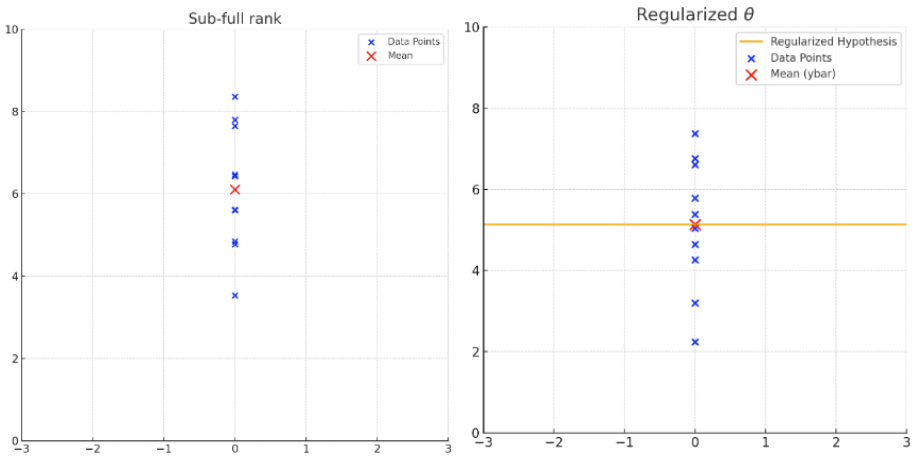
We call this kind of problem **collinearity**: our input data sit on a lower-dimensional "linear" surface.

We can see this in higher dimensions too: below, we have a 2-d input space (3-d plot), but our inputs only occupy a **line**.



There are many possible planes that go through that line: each of these is an **equally good** solution for regression.

We don't have this problem if we use regularization: among all of our **equivalent** options, we just pick the one with the **minimal** $||\theta||$.



Now, we have one solution: we can get this analytically!

2.6.8 Error Amplification

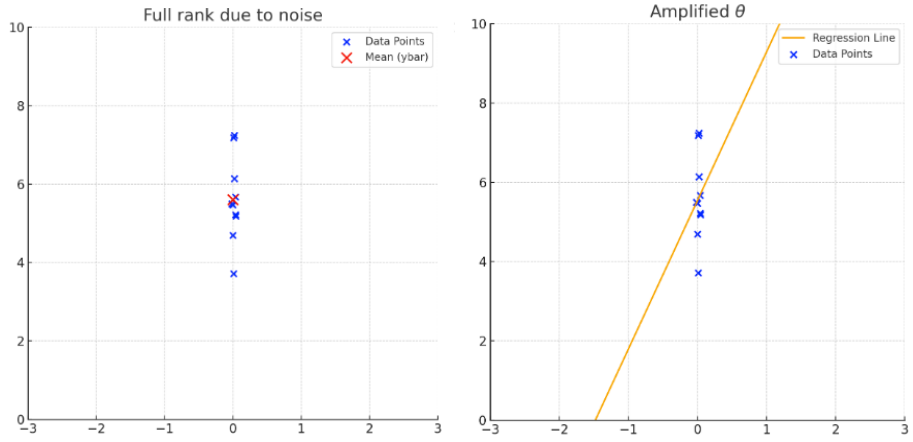
Surely, this is a really rare situation: our data typically won't line up **perfectly**.

- Maybe, but that's part of the problem: let's see what happens if our data is **not** perfectly aligned. We say that XX^T is **ill-conditioned** or "nearly singular".

Definition 70

We call a matrix **ill-conditioned** or **nearly singular** if it is very close to a singular/non-invertible matrix.

You can compute whether a matrix is ill-conditioned using something called a "condition number", but we'll omit that point.



Our data has a **large slope** now! This is a problem: x has **no effect** on y, we just added a little noise to the x-axis.

Our model notices, "**very small** change in x, **moderate** change in y", and assumes the slope θ should be **large**.

Another way to see this: technically, if you wanted to draw a line through the data, you'd draw a vertical line: "infinite" slope.

- This is wrong: our change in y isn't actually explained by x, it's explained by some "**randomness**" in the output (which is common in real data).

Concept 71

One problem with data that *almost* falls on a line, is **error amplification**.

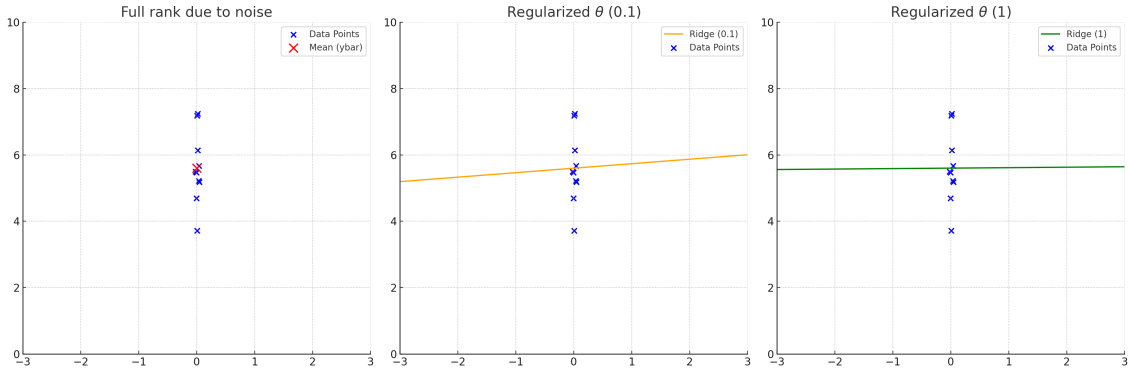
- If x_i varies by a small amount, while y varies by a larger amount, our model may assume θ_i is **very large**.
- That means that, if you had a much larger x_i value, the model will predict y is **way larger**.

Thus, our error gets **amplified** as we move further away.

If our problem is that θ is too large, then we can solve this problem by regularizing θ.

Sometimes, you might hear people mention "numerical instability" of the inverse when a matrix has a small determinant: these are, mathematically, the same problem.

If $\det(A) = 0.01$, then $\det(A^{-1}) = 100$. The



With $\lambda = 0.1$, our slope is already much less sharp. with $\lambda = 1$, we end up with (mostly) the same solution as we did in the singular case.

Concept 72

Ridge Regression helps **improve** our model by

- Making our model more **general** and resistant to **overfitting**
- Making sure **solutions** are **unique**
- Keeping our matrix XX^T **invertible**, so we can find a **solution**.

2.6.9 Error Amplification Example (Optional)

Here's a very simple computational example:

- **Example:** Suppose that two inputs are **almost exactly the same**: $x_1^{(1)} = 0$, and $x_1^{(2)} = 0.01$.
- But, the outputs are **somewhat** different. The outputs are $y^{(1)} = 25$, $y^{(2)} = 26$.

If we assume **all** of the change in the output is a result of the the **input**, then it looks like a **tiny** change in x_1 has a **huge** effect on the output y.

$$\frac{dy}{dx} = \frac{\Delta y}{\Delta x} = \frac{1}{.01} = 100 \tag{2.65}$$

This suggests that x_1 has a **100x** effect on our output! Even though, it could just be that x has small variation, and y has larger variation.

$$100x_1 + 25 = h(x) \tag{2.66}$$

Imagine that $x_1 = 10$: suddenly, the prediction is 1025. That's why we call it **error amplification**: a small error near $x = 0$, becomes huge as we get further away.

2.6.10 Regularizer justification: Prior Knowledge (Optional)

One more way to justify our regularizer applies to a lot of broader statistics: the concept of a **prior belief**.

- Suppose we have prior expectations about what our model should look like: based on theory, or prior experience.
- We might consider a model **more different** from that past one, Θ_{prior}, to be **suspicious**, and less likely to be good.

So, we can **punish our model** for being too different from that expectation.

- This makes our model more **conservative**: it avoids creating a very "extreme" model, without strong justification from the training data.

Our data has to "convince" us that it's worth trying a different model.

Concept 73

If we have a **prior** hypothesis Θ_{prior} to work with, we might improve our **new** model by encouraging it to be **closer** to the old one.

$$R(\Theta) = \left\|\Theta - \Theta_{\text{prior}}\right\|^2$$

We measure how **similar** they are using **square distance**.

Example: You have a **pretty good** model for **predicting** company profits, but it isn't perfect. You decide to train a **better** one, but you expect it to be **similar** to your old one.

In our case, we **don't have** a prior hypothesis $\theta_{\texttt{prior}}$. We have no clue of what a **good solution** looks like.

We'll take a neutral stance:

- When we know nothing, we're **equally likely** to expect θ_k to be positive, or negative.
- In other words, we don't know if x_i is likely to increase or decrease y.

So, our guess should average out to 0: the most likely effect is **none**.

- Another way to justify this: if we pick a bunch of variables randomly, we might expect a lot of them to be **irrelevant**.
- **Example:** Without knowing anything, we probably don't expect your birth date to affect your academic performance, positive or negatively.

Thus, we treat $\Theta_{\text{prior}} = \vec{0}$.

Concept 74

We can interpret **ridge regression** as expecting each θ_k terms to be close to 0.

- In other words, $\Theta_{\text{prior}} = \vec{0}$.

$$R(\Theta) = ||\theta - \vec{0}||^2 = ||\theta||^2$$

This is the same formula we arrived at earlier.

2.7 Evaluating Learning Algorithms

Now, we have successfully developed an **algorithm** for **learning** from our data. But, did our algorithm make a **good** hypothesis? How do we do **better**?

2.7.1 What λ should we choose?

There's something we ignored earlier: how do we pick the **best** value of λ? We didn't go into detail, but that value of λ will affect our algorithm's **performance**.

- We mentioned that different λ values have different **tradeoffs**, so we need to figure out which λ value is best for our problem.
- This λ adjusts exactly how we learn: how do we balance learning from **data** against the need to **generalize**?

So, we need to **optimize** our λ value. Let's figure out how to go about that.

2.7.2 Tradeoffs: Estimation Error

High and low λ values have benefits and drawbacks. These tradeoffs can be loosely divided into **two categories**.

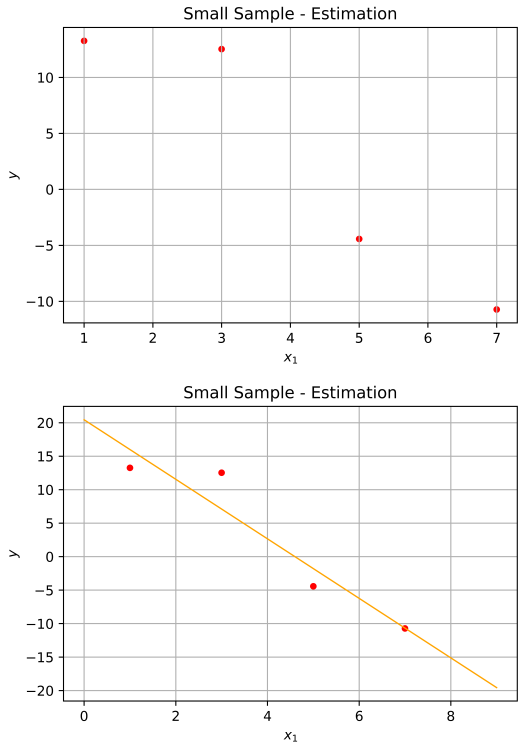
When we generalize, we're trying to avoid **estimation error**: we incorrectly guess the overall distribution we're trying to fit. We **estimate** poorly if we **generalize** poorly.

Definition 75

Estimation error is the error that results from poorly **estimating** the **solution** we're trying to find.

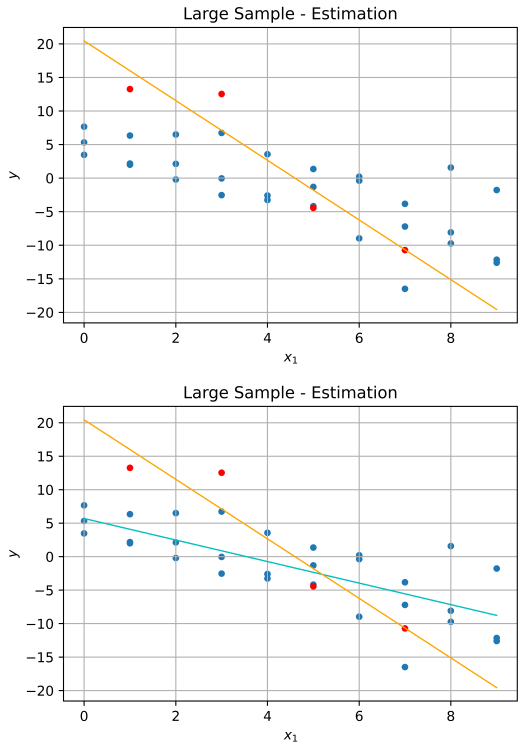
This can be caused by **overfitting**, getting a bad (**unrepresentative**) sample, or not having enough **data** to come to conclusion.

Example: Let's try a regression problem, but we'll use only 4 points to make our plot.



This is the regression solution we get based on our small dataset.

We might be suspicious. One way to reduce **estimation error** is to increase our number of data points (though this isn't always an option, or sufficient!)



Our regression from before doesn't look so good on this model... We make an updated regression, and get a more accurate result.

Clarification 76

λ doesn't lower **estimation error** in the **same way** that increasing **sample size** does, but the problem is **similar**.

2.7.3 Tradeoffs: Structural Error

However, not all problems are caused by estimation error: sometimes, it **isn't even possible** to get a good result - you chose the wrong **model class**.

This means the **structure** of your model is the problem, not your method of **estimation**. Thus, we call this **structural error**.

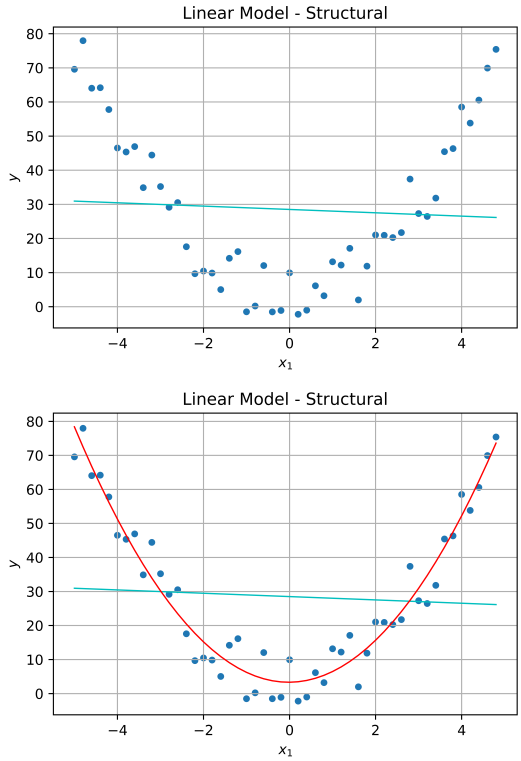
Definition 77

Structural error is the error that results from having the wrong **structure** for the **task** you are trying to accomplish.

This can result from the **wrong class** of model, but sometimes, your model class doesn't have the **expressiveness** it needs for a complex problem.

It can also happen if your algorithm **limits** the available models in some way, like how λ does.

Example: If the **true shape** of a distribution is a parabola x^2, there is **no** linear function $mx + b$ that can match that: this creates **structural error**.



Our **linear** model isn't able to represent a quadratic function... so, we switch to a more **expressive** model: a **quadratic** equation.

Remember that **expressiveness** is about how many possible models you have: if you have more models, you can solve more problems.

Clarification 78

Note that λ does not restrict our model class **as severely** as **switching polynomial order**, like above.

But, λ **limits** the use of larger θ, which does make it **unable** to solve some problems. So, the **structural error** problem is similar.

2.7.4 Tradeoffs of λ

Based on these two categories, we can discuss the tradeoffs of λ more easily.

As we mentioned, regularization **reduces** estimation error:

If we overfit to our current data, we are poorly **estimating** the distribution, because the training data may not perfectly **represent** it.

Concept 79

A **large** λ means **more regularization**: we more strongly push for a more **general** model, over a more **specific** one.

This results in...

- **Reduced** estimation error
- **Increased** structural error

However, **regularization** also **limits** the possible models we can use - those it views as less "general", it **penalizes**.

- That means the scope of possible models is **smaller** - some models are no longer **acceptable**. What if the only valid solution was in that space we **restricted**? Well, then we can't **find** it.
- That means there are certain **structural** limits on our model: that means that regularization **increases** structural error!

Concept 80

A **small** λ means **less regularization**: we care less about a more **general** model, allowing more **specific** data to come into play.

This results in...

- **Increased** estimation error
- **Reduced** structural error

2.7.5 Evaluating Hypotheses

So, we know that we have these **two** types of **error**. But it's **difficult** to **measure** them separately.

So instead, we just want to measure the **overall performance** of our hypothesis.

We do this using our **testing error**: this tells us how good our hypothesis is **after** training.

$$\mathcal{E}(h) = \frac{1}{m} \sum_{i=n+1}^{n+m} \left(h(x^{(i)}) - y^{(i)} \right)^2 \tag{2.67}$$

Note that, before, we were using **regularization**. This is so we can **make** a more **general** model.

But here, we've **removed** it, because training is **done**: we're **not** going to make our hypothesis **better**. We just care about how **good** it came out.

We're already measuring the **generalizability** by using **new data**!

Clarification 81

When we **evaluate a hypothesis** using **testing error**, we are **done training**: our hypothesis will not change.

Because of this, we **do not** include the **regularizer** when **evaluating** our hypothesis.

2.7.6 λ's purpose: learning algorithms

Notice that we **removed** regularization when we were **evaluating** our hypothesis: regularization was used to **create** our hypothesis, but it is not **part** of that hypothesis.

Our hypothesis only includes the parameters Θ: not λ!

That's because λ is part of our **algorithm**: it determines how we find our hypothesis. So, let's talk about that.

Definition 82

A **learning algorithm** is our procedure for **learning** from data. It uses that data to create a **hypothesis**. We can diagram this as:

$$\mathcal{D}_n \longrightarrow \boxed{\text{learning alg } (\mathcal{H})} \longrightarrow h$$

In a way, it's a function that takes in **data** $\mathcal{D}_n$, and outputs a **hypothesis** h.

We're choosing **one hypothesis** h from the hypothesis class $\mathcal{H}$: this is why $\mathcal{H}$ appears in the notation above.

We can write this as $h \in \mathcal{H}$

2.7.7 Comparing Hypotheses and Learning Algorithms

We can take our learning algorithm

$$\mathcal{D}_n \longrightarrow \boxed{\text{learning alg } (\mathcal{H})} \longrightarrow h$$

And compare it to our hypothesis h:

$$x \rightarrow \boxed{h} \rightarrow y$$

In a way, our learning algorithm is a function, that outputs another function!

This is similar to $\mathcal{E}_n$, which instead takes a function as **output**!

- Our **hypothesis** can be adjusted with our **parameter** Θ: if we change Θ, we change our **performance**.
- Our **learning algorithm** depends on λ: so, λ is like a **parameter**. But, it's different from Θ: Θ **is** our model, λ controls how we **choose** our model.

– So, it's a parameter (λ) that affects other parameters (Θ). Because of that, we call it a **hyperparameter**.

It affects our hypothesis by pressuring it to have lower magnitude!

Definition 83

Parameters are **variables** that adjust the behavior of **our model**: our hypothesis.

A **hyperparameter** is a **variable** that can adjust **how we make models**: our learning algorithm.

The **only** hyperparameter we have for now is λ, but the **development** of hyperparameters is an ongoing area of **research**.

Concept 84

Lambda, or λ, is a **hyperparameter**: it controls our **learning algorithm.**

2.7.8 Evaluating our Learning Algorithm

So, while we can evaluate each **hypothesis**, it's also important to measure how our **learning algorithm** is performing.

How do we measure it? Well, the job of our **learning algorithm** is to **pick good hypotheses**.

Concept 85

We can **evaluate** the performance of a **learning algorithm** using **testing loss**: a good learning algorithm will create **hypotheses** with low testing loss.

You could think of this as measuring the **skill** of a **teacher** (the learning algorithm) by the **success** of their **student** (the hypothesis) on a **test** (testing loss).

2.7.9 Validation: Evaluating with lots of data

When we were creating hypotheses, **randomness** caused some problems: you might not get **training data** that matched the **testing data** very well.

The **same** can happen here, when **evaluating** your **algorithm**: maybe your model happened to create a bad (or unusually good!) hypothesis because of **luck**.

The easy solution to **randomness** is to add **more data**: we get more **consistency** that way.

So, we **repeatedly** get new training data and test data. For each, we train a **different hypothesis**. We can **average** their performance out, and use that to **estimate** the quality of our algorithm.

Definition 86

Validation is a way to **evaluate a learning algorithm** using **large amounts of data**.

We do this by **running** our algorithm **many times** with new data, and **averaging** the testing error of all the hypotheses.

- This process is often requires having **lots of data** to train with, but is a **provably** good approach.

2.7.10 Our Problem: When data is less available

As mentioned, this takes up **lots of data**. What if our data is limited?

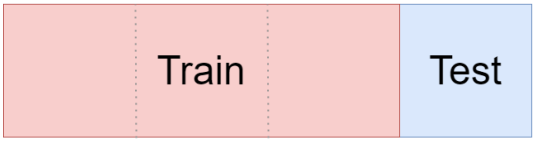
Data is often **expensive**. It might even be impossible to get more!

In this case, we'll assume that have some **finite** data, $\mathcal{D}_n$. We **can't get more**.

Previously, we solved validation by using **more data**, and generating **multiple hypotheses**.

- One set of data gives us one **hypothesis**.
- But, what if, rather than using **completely** new data for each hypothesis, we used **slightly different** data each time?

First, need to break $\mathcal{D}_n$ into a chunk for training, and a chunk for testing.



How do we get more hypotheses from this dataset?

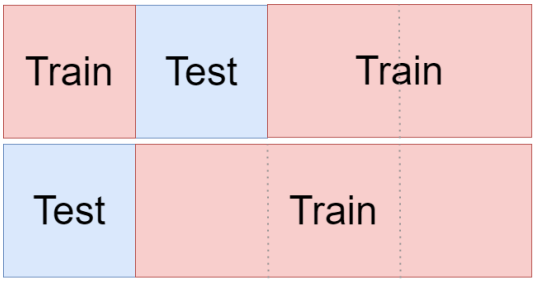
2.7.11 Cross-Validation

We mentioned that we want **different** hypotheses. Our hypotheses depend on our **training data**. So we want to **change** our training data.

We can't **add** data to it, because then we **lose** testing data. We shouldn't **remove** training data, because then we're just making a hypothesis that's **less well-informed**.

Instead, we'll **swap** some of the training data for testing data.

This will create a new hypothesis, and the data is partially different! In fact, we can do this for each of our chunks:



We now have **four different hypotheses** for the price of one!

Definition 87

Cross-validation is a way to **evaluate** a learning algorithm using **limited data**.

- We do this by **breaking** our data it into **chunks** to create **multiple hypotheses** from one dataset.
- For each **chunk**, we train one dataset on all the data **not in that chunk**. We get our **test error** using the chunk **we left out**.

For k chunks, we end up with k hypotheses. By **averaging** out their performance, we can **approximate** the quality of our algorithm.

This approach is much **less expensive**, and very common in machine learning!

But, some of the theoretical **benefits** of validation are not **proven** to be true for cross-validation.

Clarification 88

Note that the goal of validation and cross-validation is **not** to evaluate **one hypothesis**.

Instead, it is instead meant to evaluate a **learning algorithm**. This is why we have to create **many** hypotheses: we want to see that our algorithm is **generally** good!

2.7.12 Hyperparameter Tuning

Now, we know how to **evaluate** a learning algorithm, just like how we **evaluate** a hypothesis.

Once we knew how to evaluate a hypothesis, we started optimizing our **parameters** for the **best** hypothesis. So, we could do the same for our **learning algorithm**.

How do we **optimize** a learning algorithm?

Each λ value creates a slightly **different** learning algorithm: we can **optimize** this **hyperparameter** to create the **best** learning algorithm.

2.7.13 How to tune our algorithm

When we were **optimizing** our hypothesis, we started by **randomly** trying hypotheses. Then, we used an **analytical** approach.

By "analytical", we mean directly creating an equation, and solving it.

We don't always have **simple** equations to work with: with all of our data, it's hard to come up with **manageable** equations. So, we **won't** try doing it **analytically**.

So, we could **randomly** try λ values and pick the **best** one. This is pretty **close** to what we usually end up doing. For each value we pick, we'll use **cross-validation** to evaluate.

For now, we'll systematically go through λ values: $\lambda = .1, .2, .3 \ldots$

Concept 89

Hyperparameter tuning is how we **optimize** our **learning algorithm** to create the **best** hypotheses.

The simplest way to do this is to try **multiple** different values of λ. For each value, we use **cross-validation** to evaluate that learning algorithm.

Finally, we pick whichever λ gives you the **best** algorithm, and thus the **best** hypotheses.

2.7.14 Hyperparameter Tuning: Two kinds of optimization

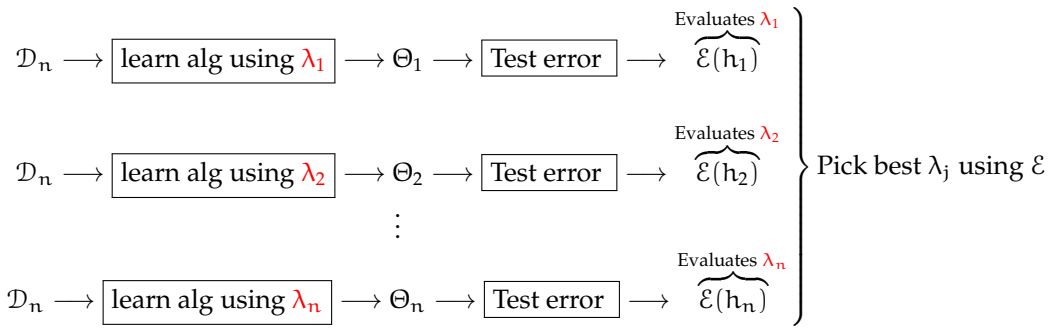
There's something often **confusing** about hyperparameter tuning to students:

In case the word "optimization" starts to look like gibberish in this section, remember: it just means, "find the best option".

- When we're **optimizing** λ, we try many values λ_j. Each λ_j create a **learning algorithm** we have to evaluate.
- But a single learning algorithm is already an optimization problem: the learning algorithm is supposed to find Θ.
  - So, we have to **optimize** Θ, while we're in the middle optimizing λ.

That means, **every time** we try a different λ value, we have to do one optimization problem. Optimizing Θ many times, lets you optimize λ once.

Remember that, in this situation, Θ and h are almost(but not quite) the same thing.



That means we have **two layers** of optimization!

Clarification 90

We **optimize** λ by trying many values.

- But, for each λ value, we have to **optimize** Θ.

So, we have to optimize Θ **repeatedly** in order to optimize λ **once**! This gives us λ^*.

Once we've found our best hyperparameter λ^*, we can use it to get our best parameters: θ^*.

2.7.15 Pseudocode Example

This technique is **not** limited to regression. Thus, we'll be a bit more **general**: we won't assume an **analytical** solution. Instead, we **optimize** by just trying different Θ values.

We can represent this in pseudocode:

If this pseudocode isn't helpful to you, don't worry! Some students like it, some don't.

```
LAMBDA-OPTIMIZATION(D, lambda_values, theta_values)
for λ in lambda_values          #Try lambda values
    for Θ in theta_values       #Try theta values
        Calculate J(Θ)          #Compare values
    Choose best theta value Θ*  #Best for each lambda
Choose best lambda value λ*

return λ*
```

To reiterate: this λ^* will then we used to get our final result, θ^*.

2.8 Terms

- Hypothesis
- Theta (Θ)
- Input Space
- Regression
- Feature
- Feature Transformation
- Training Error
- Test Error
- Objective Function
- Min function
- Argmin function
- Star Notation (θ^*)
- Linear Regression
- Hypothesis Class
- Square Loss
- Ordinary Least Squares (OLS) Problem
- OLS Objective Function
- Hyperplane
- Weight
- Input Matrix
- Output Matrix
- Gradient
- OLS Solution
- Regularization
- Regularizer
- Regularizer for Regression
- Lambda (λ)

- Ridge Regression
- RR Objective Function
- RR Solution
- Invertibility
- Estimation Error
- Structural Error
- Expressiveness
- Learning Algorithm
- Hyperparameter
- Validation
- Cross-Validation
- Chunk (Cross-Validation)
- Hyperparameter Tuning