Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Spring 2023

Contents

CHAPTER 2

Regression

2.1 Problem Formulation

In the last chapter, we discussed many broad ideas in machine learning.

In this section, we will **review** those concepts, and use one concrete example to help make them clearer: **regression**.

2.1.1 Hypothesis (Review)

In the last chapter, we distinguished between problem and solution. Our **problem** is getting from our input x to our desired output y.

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

Meanwhile, our **solution** is some model we place in between those two: some **function** we can use to compute output based on input. This is our **hypothesis** h.

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

Remember that we store our **parameters** in a vector Θ: this vector defines our **model** within our **model class**.

This Θ is what we hope to be able to improve, and use to find a **better model**.

- Depending on how **precise** we want to be, remember that we can write our hypothesis as $h(x)$ **or** $h(x;\Theta)$: both are **valid**, the latter emphasizes the fact that Θ is a **variable.**
- We'll focus on Θ more if we assume we know which **hypothesis class** we're in - if we know that, Θ fully **defines** our model.

Often, people will treat Θ and h as almost interchangeable: be careful when you're doing this!

2.1.2 The Problem of Regression

In regression, our problem is taking data in a vector, and converting it into a **real number**.

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

This is the mission of our function, the **hypothesis**. Function are important, so we'll introduce some common notation.

Notation 1

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:

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

Often, instead of the "set of inputs", you'll hear people talk about the **input space**. This is essentially the same thing: it is a term worth getting used to.

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

Now, we can write this more efficiently:

Definition 2

Regression is a **machine learning setting** where we use a **vector** of real numbers to return 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-axis), and you want to **predict** the output (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 will not be in the right 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 3

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

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

Sometimes, we use it on already-valid formats to find **new patterns** in data.

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

$$\text{Brand A} = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad \text{Brand B} = \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad \text{Brand C} = \begin{bmatrix}0\\0\\1\end{bmatrix} \tag{2.1}$$

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 $\varphi(x)$.

- 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

Regression is **supervised**, meaning we have training data with a correct output included: we have an "answer key" to a practice quiz.

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

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

Remember that the superscript tells us that this is the 1st data point.

And so 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 Measuring our performance

In the last chapter, we discussed that we use our past **training data** to learn a good **model** for our future **testing data**.

So, even though the training data is what we have in front of us, the **testing data** is what we care the most about. We want our machine to handle new situations.

- We'll assume our data are **IID**, so that we can **learn** and **generalize** effectively. And we'll evaluate ourselves using a **loss function** $\mathcal{L}$, a measure of how poorly our model is running.

We care about **expected loss**, so we'll take an average. We'll start with our training data, so we call it **training error**:

Key Equation 4

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

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

Notice that we treat this error as a function of h, our hypothesis - our **hypothesis** is what we're **adjusting** to try to improve the error.

Clarification 5

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.

Make sure to keep track of the difference as you go through this chapter - the idea will come back later.

2.1.6 Learning to Generalize

Our goal, though, is to perform well on testing data: to be able to **generalize** to data we haven't seen before.

So, let's define **test error**. This time, we have m new data points.

$$\mathcal{E}(h) = \frac{1}{m} \sum_{i} \mathcal{L}\left(h(x^{(i)}), y^{(i)}\right) \tag{2.2}$$

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

Key Equation 6

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 want to minimize **test error**, but by definition of "data we haven't seen before", we can't use it to design our hypothesis.

So, for now, the next best thing to "minimize test error" is "**minimize training error**", while doing our best to help our hypothesis **generalize**. We'll discuss how to do that.

2.2 Regression as an optimization problem

We've mentioned that 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.

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

Lot of research has gone into solving optimization problems!

2.2.1 Objective Function

Our goal is to minimize our loss. But, **training loss** is not our main goal: to compensate for that, we might include other terms to make it more **general** (work better for more situations).

To distinguish between our **loss** versus our overall goal, we will define an **objective** function.

In later sections, we will introduce something called a **regularizer** to do just that!

Definition 7

An **objective function** is the function we are **optimizing** for: usually, this means that our goal is **minimizing** it.

- This term usually contains the **loss**, and sometimes, a **regularizer**.

We minimize our objective function using our **parameters** Θ, so we take that as an input: $J(\Theta)$

We will be seeing a lot of Θ for a while.

2.2.2 Parts of an Objective Function

For now, we will just consider loss, so our **objective function** will be the same as our **training error**, dependent only on the **loss function**.

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

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

But once we introduce the **regularizer** $R(\Theta)$ to improve **generalization**, we'll add that term:

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

We will discuss this term in a later section; don't worry about it for now.

We'll also include a scaling factor λ so we can control this term:

Key Equation 8

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

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

- The left term is the **loss**: how well we perform on training data.
- The right term is the **regularization**: it improves how "general" (good for new situations) our model might be.

In the long run, this is the function we want to **optimize**.

Notice that our objective function **depends** on our training data $\mathcal{D}$ as well: our data determines how well we're doing.

A good solution for a political science exam is probably pretty bad for a geology exam.

- Just like how we can use ";" when writing $h(x;\Theta)$, we'll use the same notation here: $J(\Theta;\mathcal{D})$
- Θ is our "main" input variable, but if we switch out our data $\mathcal{D}$, we would get a different result.

Clarification 9

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.

2.2.3 Minimization Notation

Our goal is to minimize J by adjusting Θ. If we accomplish this, there are two questions we can ask ourselves, and some corresponding notation.

To show our point, we'll use the following example:

Example: Take $f(x) = (x-1)^2$. The minimum output is 0, which happens at $x = 1$. So, we have a minimum at $(1, 0)$.

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

Notation 10

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:

$$\min_{x} (x-1)^2 = 0 \tag{2.3}$$

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

- What **value** of x gives us **minimum** J?

Notation 11

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:

$$\arg\min_{x} (x-1)^2 = 1 \tag{2.4}$$

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

Clarification 12

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**.

2.2.4 Optimal Value Notation

So, we want to know what the best model we want get is, where this model is represented by Θ.

Notation 13

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

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

Example:

$$x^* = 1 \text{ for the above example.} \tag{2.5}$$

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

Key Equation 14

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 can pick a concrete example.

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:

$$y = mx + b \tag{2.6}$$

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 $m = \theta_1$.

$$h(x) = \theta_1 x + \theta_0 \tag{2.7}$$

2.3.2 The Linear Model, 2-D

We want to have **multiple** input variables: x will be a **vector**, not a number. So, for our above example, we'll **replace** x with x_1.

$$h(x) = \theta_1 x_1 + \theta_0 \tag{2.8}$$

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.

$$h(x) = \theta_2 x_2 + \theta_1 x_1 + \theta_0 \tag{2.9}$$

2.3.3 The Linear Model, d-D

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

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

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_d x_d \tag{2.10}$$

2.3.4 The Linear Model using Vectors

Here, we are **multiplying** components of x and θ together, then **adding**. This looks like a **dot product**:

$$h(x) = \theta_0 + \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \tag{2.11}$$

If we write this symbolically, we get:

$$h(x) = \theta_0 + \theta \cdot x \tag{2.12}$$

Unfortunately, we had to leave θ_0 out to make it work. θ is used for the parameters of our **dot product**, Θ is **all** parameters.

Notation 15
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**: we'll use matrix multiplication a lot in this chapter, and course.

One benefit is that we can use **matrices** instead of just **vectors**!

Key Equation 16
The **linear regression** hypothesis is written as

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

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

Remember that, when written out, this looks like:

$$h(x) = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \cdots & \theta_d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} + \theta_0 \tag{2.13}$$

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?

We have an **actual** output value a that we want to compare to our **guessed** output g. If they're more **different**, that's worth punishing **more**.

$g - a$ is our difference, but we want to punish it being both too **high** or too **low**: we'll **square** the **difference**.

$$\mathcal{L}(g, a) = (g - a)^2 \tag{2.14}$$

Or, if we use $y = a$:

$$\mathcal{L}(g, y) = (g - y)^2 \tag{2.15}$$

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 17

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

- Always positive: high and low guesses are treated equally.
- When we have vectors, we can **represent** it with a **dot product** $w \cdot w$, or **matrix multiplication** $w^T w$: tools we like using.
- $\|w\|$ is **not smooth**: this isn't good for derivatives! $\|w\|^2$ *is* smooth.
  - We can see that $\|w\|$ isn't smooth at $w = 0$.
- The **slope** is **small** when you're close to the correct answer: if you're close to right, the loss stays low.
  - Inversely, if you're really wrong, then the model really penalizes you for being more wrong.
  - For example: the difference between 2^2 and 1^1 ($2^2 - 1 = 3$) is much smaller than between 24 and 23 ($24^2 - 23^2 = 47$)

2.3.6 Our Goal

Our goal is to minimize the **loss** $\mathcal{L}$ on our data set, using the **linear** model.

We want the smallest distance between our **hypothesis** and the **data points**: we want it as **close** as possible.

Let's write this into a **single equation**: $J(\Theta) = J(\theta, \theta_0)$

$$J(\theta, \theta_0) = \text{Training Loss} \tag{2.16}$$

Get the equation for **expected loss**, using $^{(i)}$ to show which data point we're using in the sum:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(g^{(i)}, y^{(i)}) \tag{2.17}$$

Substitute in **squared loss**:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} (g^{(i)} - y^{(i)})^2 \tag{2.18}$$

Our guess is given by our **hypothesis**, $h(x^{(i)}; \Theta)$

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left(h(x^{(i)}; \Theta) - y^{(i)} \right)^2 \tag{2.19}$$

And finally, we use our linear model, $\theta^T x^{(i)} + \theta_0$.

Key Equation 18

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

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

To clarify:

$$J(\theta, \theta_0) = \underbrace{\frac{1}{n} \sum_{i=1}^{n}}_{\text{Averaging}} \left(\underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}} \right)^2 \tag{2.20}$$

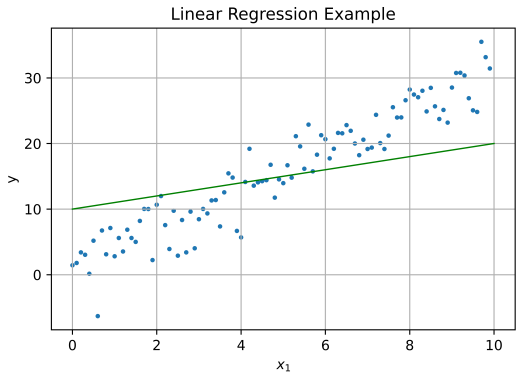
From this, we want to find

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

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

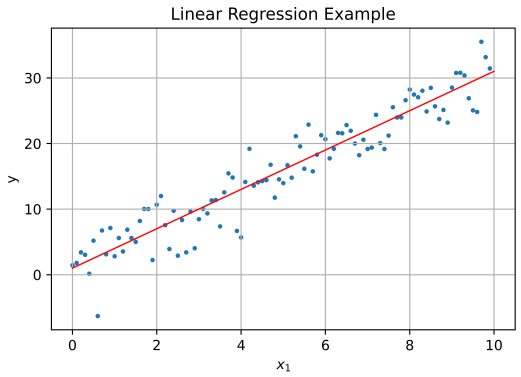
2.3.7 Visualizing our Model

With **one variable**, we've seen that our linear model simply turns into $\theta_1 x_1 + \theta_0$. As you'd expect, on a plot, this looks like a **line** in the **2D plane**.



This example of linear regression is not a great fit: $(\theta_0 = 10, \theta_1 = 1)$

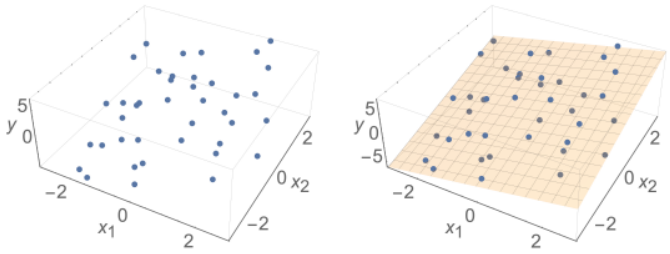
We're trying to get our line as **close as possible** to the points, hoping to find a linear pattern. We're **fitting** our line to the data.



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

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

Extending our line into a second dimension, we create a **plane**.



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

Higher-dimension versions are hard to visualize. So, instead, we don't even try to. Because they're a higher-dimensional version of a **plane**, we call it a **hyperplane**.

Definition 19

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, $(d+1)^{th}$ dimension.

Thus, the "**height**" of our plane ($(d+1)^{th}$ dimension) at a particular point in the (d-dimensional) **input** space represents the **output** of our linear hypothesis, given those inputs.

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.

With this intuition, we can imagine our **hyperplane** as trying to get as **close** to all of the data points as it possibly can.

2.3.8 Another Interpretation

There's another, similar way to interpret our model

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_d x_d \tag{2.21}$$

Before, we took θ_k as just an **extension** of the $mx + b$ formula: θ_k tells us how much x_k affects our output.

- However, we can also think about the **relative** scale of each θ_k: if θ_2 is **larger** than θ_1, then x_2 has a **stronger** effect on the output than x_1.
- We can say that x_2 **weighs** more heavily in our calculation: it has more say in the **result**.

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

Definition 20

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.

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.

Why introduce such a silly algorithm? For two 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.

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

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left(\underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}} \right)^2 \tag{2.22}$$

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, one thing that was strange is that we **separated** θ_0 from our other θ_k terms. Maybe we can **fix** that.

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. We only do this when we don't care which data point we're using.

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_d x_d \tag{2.23}$$

We converted this into a **dot product** because each θ_n term is **multiplied** by an x_k term, except θ_0.

But if we **really** want to include θ_0, then could we? We know what's missing: "θ_0 is **not** multiplied by an x_k term". So... could we get one? Is there a x_0 factor we could **find**?

We need θ_0 to be **multiplied** by something. Is there something we could "**factor** out"? How about: $x_0 = 1$?

You can always factor out 1 without changing the value!

$$h(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_d x_d \tag{2.24}$$

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!

$$x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix}, \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_d \end{bmatrix}, \qquad h(x) = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} \tag{2.25}$$

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

$$h(x) = \theta \cdot x = \theta^T x \tag{2.26}$$

Concept 21

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

In order to do this, we have to **append** a value of 1 to x as well.

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 Summing over data points

Currently, when using our **objective** function, we have to **sum** over **every** single data point. For the 1D case, this means we have to do:

$$J = \frac{1}{n} \sum_{i=1}^{n} (\theta x^{(i)} - y^{(i)})^2 \tag{2.27}$$

- This is a bit of a hassle - it **forces** us to use $x^{(i)}$ notation, and we have to be conscious of that **sum**.

By using **vectors** above, we were able to work with **many** variables θ_k at the same time, making it easier to **represent** and **work** with them in the future.

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

2.5.4 Summing with Vectors: Row Vectors

We want to represent **addition** using **vectors**. We did that when we were adding $x_k \theta_k$ terms with a **dot product**.

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} \tag{2.28}$$

But, dot products also include **multiplication**. Above, our terms are **squared**. So, we can multiply $(\theta x^{(i)} - y^{(i)})$ times itself!

$$J = \frac{1}{n}\sum_{i=1}^{n}(\theta x^{(i)} - y^{(i)})(\theta x^{(i)} - y^{(i)}) \tag{2.29}$$

We'll write $r^{(i)} = \theta x^{(i)} - y^{(i)}$ to simplify our work.

$$J = \frac{1}{n}\sum_{i=1}^{n} r^{(i)} * r^{(i)} \tag{2.30}$$

In a dot product, we **add** the **dimensions** together. So, we'll give each term in our sum its own **dimension**.

$$J = \frac{1}{n}\sum_{i=1}^{n} r^{(i)} * r^{(i)} = \frac{1}{n}\begin{bmatrix} r^{(1)} \\ r^{(2)} \\ r^{(3)} \\ \vdots \\ r^{(n)} \end{bmatrix} \cdot \begin{bmatrix} r^{(1)} \\ r^{(2)} \\ r^{(3)} \\ \vdots \\ r^{(n)} \end{bmatrix} \tag{2.31}$$

We've got a single vector we could call R.

We could make it a **column vector**, but we already use the **rows** to indicate the **dimensions**.

$$\theta = \left.\begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_d \end{bmatrix}\right\} \text{dimensions as rows...} \tag{2.32}$$

So, let's use **columns** instead: each **column** will be a **data point**: we'll use a **row vector** $(1 \times n)$.

$$R = \overbrace{\begin{bmatrix} r^{(1)} & r^{(2)} & r^{(3)} & \cdots & r^{(n)} \end{bmatrix}}^{\text{data points as columns!}} \tag{2.33}$$

2.5.5 Going from x to X

We can do the same for our input data $x^{(i)}$:

Notation 22

We can store all of our 1-D **data points** in a **row vector**:

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \cdots & x^{(n)} \end{bmatrix}$$
$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \cdots & y^{(n)} \end{bmatrix}$$

We can write our **objective function** as

$$J = \frac{1}{n} \begin{bmatrix} r^{(1)} & r^{(2)} & r^{(3)} & \cdots & r^{(n)} \end{bmatrix} \begin{bmatrix} r^{(1)} \\ r^{(2)} \\ r^{(3)} \\ \vdots \\ r^{(n)} \end{bmatrix} \tag{2.34}$$

Or more compactly:

$$J = \frac{1}{n} RR^T \tag{2.35}$$

Since we had $r^{(i)} = (\theta x^{(i)} - y^{(i)})$, we can write

$$R = \theta X - Y \tag{2.36}$$

Still in the 1D case!

Let's use this to expand our objective function:

Concept 23

In 1-D, we can use row vectors to sum our data points as

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

We've successfully **removed** the **sum**!

This format **stores** all of our **data points** in **one object**, just like how we wanted.

2.5.6 Putting it together: Matrices

Now, we have shown both a way to express x_1, x_2, x_3 as a single $(d \times 1)$ matrix:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} \tag{2.37}$$

We'll leave off the appended 1 for now.

And a way to express $x^{(1)}, x^{(2)}, x^{(3)}$ as a single $(1 \times n)$ matrix:

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \cdots & x^{(n)} \end{bmatrix} \tag{2.38}$$

Why not combine them into a single object?

Key Equation 24

X is our **input matrix** in the shape $(d \times n)$ that contains information about both **dimension** and **data points**.

$$X = \overbrace{\begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \cdots & x_d^{(n)} \end{bmatrix}}^{n \text{ data points}} \Bigg\} d \text{ dimensions} \tag{2.39}$$

If we include the appended 1, we write this as the $((d+1) \times n)$ matrix

$$X = \overbrace{\begin{bmatrix} 1 & \cdots & 1 \\ x_1^{(1)} & \cdots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \cdots & x_d^{(n)} \end{bmatrix}}^{n \text{ data points}} \Bigg\} d+1 \text{ dimensions} \tag{2.40}$$

Because each data point $y^{(i)}$ has only one dimension, it's the same as in the last section:

Key Equation 25

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

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

All we have to do is combine our **equations**: We can use the one in the last section, but because θ is a matrix, we have to **transpose** it.

There was no point in transposing it when it was just a constant!

Key Equation 26

Using our **appended** matrix, we can write our **objective function** for **multiple** variables and **multiple** data points as

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

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

Concept 27

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 $d + 1$.

These shapes are worth **memorizing**.

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

2.5.7 Alterate Notation

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

Notation 28

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

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

2.5.8 Optimization in 1-D - Calculus Returns!

Now that we have our **problem** presented the way we **want**, we can figure out how to **optimize** our θ.

For now, we'll revert to **sum** notation, but we'll come back to our **matrices** later.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2 \tag{2.41}$$

How do we **optimize** this? Let's just take **one data point**:

$$J(\theta) = (\theta^T x - y)^2 \tag{2.42}$$

And we'll start in 1D.

$$J(\theta) = (\theta x - y)^2 \tag{2.43}$$

If we treat θ like any ordinary **variable**, this is just a simple 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...

Note that we aren't taking $\frac{d}{dx}$: we want to change our **model**, not our **data**! So, since θ represents our **model**, we'll take $\frac{d}{d\theta}$.

$$J'(\theta) = 2x(\theta x - y) = 0 \tag{2.44}$$

We just find where the slope is 0, and solve for θ!

We technically need to prove whether this is minimum, maximum, or neither. For now, we'll assume we have a minimum.

$$\theta^* = \frac{y}{x} \tag{2.45}$$

Concept 29

If our function $J(\theta)$ has **one variable**, we can **explicitly** find **local minima** by solving for θ when the **derivative** is zero.

$$\frac{dJ}{d\theta} = 0$$

Then, you check each candidate using the second derivative to see if it is a minimum ($J''(\theta) > 0$). In this class we will often be able to ignore this step.

This concept is review from 18.01 (Single-variable calculus), but is worth repeating!

2.5.9 Using our sum

Now, we'll go back to having multiple data points we want to **average**:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta x^{(i)} - y^{(i)}\right)^2 \tag{2.46}$$

We want to do the same optimization. Thankfully, derivatives are **linear**: addition and scalar multiplication are not affected!

Check the prerequisites chapter, chapter 0, for a full definition of linearity.

$$J'(\theta) = \frac{1}{n}\sum_{i=1}^{n} 2x^{(i)} \cdot (\theta x^{(i)} - y^{(i)}) = 0 \tag{2.47}$$

And we can **solve** this just the same way.

2.5.10 Optimizing for multiple variables

Now, the tricky part is working with **vectors**.

We'll ignore the averaging and $^{(i)}$ notation since that's easy to add on afterwards.

$$J(\theta) = (\theta^T x - y)^2 \tag{2.48}$$

We want to **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 to **change**.

Derivatives are all about **change** in variables, and our **change** $\Delta\theta$ is a **combination** of changing the different **components**, $\Delta\theta_k$.

$$\Delta\theta = \begin{bmatrix} \Delta\theta_0 \\ \Delta\theta_1 \\ \Delta\theta_2 \\ \vdots \\ \Delta\theta_d \end{bmatrix} \tag{2.49}$$

So, maybe it would be reasonable to just set **every** derivative to **zero**? It turns out, the answer is **yes**!

We can show this by using the **chain rule** definition:

$$\overbrace{\Delta\theta \frac{dJ}{d\theta}}^{\text{The change in J from } \theta \text{ overall}} \approx \overbrace{\Delta\theta_0 \frac{\partial J}{\partial \theta_0} + \Delta\theta_1 \frac{\partial J}{\partial \theta_1} + \cdots + \Delta\theta_d \frac{\partial J}{\partial \theta_d}}^{\text{The change in J from each } \theta_k \text{ term}} \tag{2.50}$$

So, if all the derivatives are zero, the **overall** derivative is zero.

This approximation formula becomes exact as the step size shrinks: we go from $\Delta\theta$ to $d\theta$.

Concept 30

If our function $J(\theta)$ has $d+1$ **different parameters**, we can **explicitly** find **local minima** by getting all of the **equations**

$$\frac{\partial J}{\partial \theta_0} = 0, \quad \frac{\partial J}{\partial \theta_1} = 0, \quad \frac{\partial J}{\partial \theta_2} = 0 \ldots \quad \frac{\partial J}{\partial \theta_d} = 0$$

Or in general,

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

...And solving this **system of equations** for the **components** of θ.

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

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

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

2.5.11 Gradient Notation

Writing it this way can be a **hassle**. So, we'll continue our tradition of using **matrix-based** notation to make our lives easier.

You may recognize these **component-wise** derivatives as part of the "**multivariable** version" of the derivative: the **gradient**.

Key Equation 31

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

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

Note the subscript on the gradient! This emphasizes that our **space** is the components of θ, not the components of our data x.

For example, our previous approach boiled down to saying

$$\nabla_\theta J = \begin{bmatrix} \partial J/\partial \theta_0 \\ \vdots \\ \partial J/\partial \theta_d \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} = 0 \tag{2.51}$$

For our purposes, we will simply **represent** that **zero vector** with a single 0.

Concept 32

If our function $J(\theta)$ has a **vector variable**, we can **explicitly** find **local minima** by solving for θ when the **gradient** is **0**.

$$\nabla_\theta J = 0$$

Ignoring the requirement from earlier! We're assuming it's a minimum.

This is the form we will be solving.

2.5.12 Matrix Calculus

Taking derivatives of vectors falls under **vector calculus**. That would solve our **above** problem.

But, before, we showed that it's more **convenient** if we can store these instead as **matrices**: that way, we don't need the **sum**. Luckily, we can generalize our work with **matrix calculus**.

Below, we will use the alternative notation, to be consistent with the official notes.

$$J = \frac{1}{n}\left(\tilde{X}\theta - \tilde{Y}\right)^T\left(\tilde{X}\theta - \tilde{Y}\right) \tag{2.52}$$

We will not show here how to **find** these derivatives, but the important rules you need to compute our derivatives are in the **appendix**.

Note that **matrix** derivatives often look **similar** to **traditional** derivatives, but they are **not the same**. Most often, making this mistake will result in **shape errors**.

Sometimes, you can guess a derivative by using the familiar rules and fixing shape errors with transposing/changing multiplication order. But be careful!

When we take our derivative, we get

$$\nabla_\theta J = \frac{2}{n}\tilde{X}^T\left(\tilde{X}\theta - \tilde{Y}\right) = 0 \tag{2.53}$$

From here, we just solve for θ just like in the official notes.

Key Equation 33

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,

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

- Remember that if we're including θ_0, we replace d with $d + 1$.

And we're done with OLS!

2.6 Regularization

So far, we've shown how to make the **best** model for our **training data**. But now, we want to move to our **real** goal: performing well on **test data**.

This means we want to make a model that is **general**: it can apply well to **new data**.

2.6.1 Regularizers

Only focusing on training data is a **weakness** for our model - if by chance, we have a training data that doesn't **match** our overall distribution, we are likely to make a **bad model**.

Example: You flip **4 coins**, and get **3 heads**. You determine that this coin has a **75% chance** of landing heads. It turns out this **isn't true**: it's a fair coin, and you got **unlucky**.

You can also increase sample size (flip more times), but for complex problems this isn't always an option!

We may need a **second** way to measure our performance: one that focuses **less** on **current** performance, and **more** on predicting how **generalizable** it is.

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

Definition 34

A **regularizer** is an added term to our **loss function** that helps measure how **general** our hypothesis is.

By **optimizing** with 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: $R(\Theta)$

Example: You figure that the coin is **equally likely** to bias towards heads or tails: even if it's **weighted**, you don't know **which way**. So, you start with **50-50** odds, and **adjust** that based on evidence.

Instead of just focusing on the **specific** data for our coin, we consider how coins act in **general**.

2.6.2 Regularizer for Regression: Prior Knowledge

Now, the question is, **how** do we choose our regularizer? What will make our model more **general**?

- We want to **resist** the effects of random **chance**, like in the **coin** example above. In that example, we saw the problem with our guess by starting with a **prior assumption**.
- If you have some **previous** guess, or past experience, you might have some **model** you **expect** to work well: the data has to **convince** you otherwise.

So, we might consider a model **more different** from that past one, Θ_{prior}, to be **suspicious**, and less likely to be good.

Concept 35

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.

2.6.3 Regularizer for Regression: No Prior Hypothesis

But, what if we **don't have** a prior hypothesis? What if we have **no clue** what a **good** solution looks like?

- Well, just like in the **coin** example, we don't expect it to be **more likely** to be **weighted** towards heads or tails.
- So, even if we **didn't know** most coins are fair coins, we still would've chosen **50-50** as our guess.
- In this case, as far as we know, every θ_k term is **equally likely** to be **positive or negative** - we have no clue.

So, **on average**, we could push for it to be **closer to zero**, so it doesn't drift in any direction too strongly.

For each variable x_i, we don't know if it's likely to increase or decrease y. So we assume that the most likely effect is "none".

Key Equation 36

In general, our **regularizer for regression** will be given by **square magnitude** of θ:

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

In this model, we are biasing our $\|\theta\|$ towards 0.

This approach is called **Ridge Regression**.

We'll discuss why it's called "ridge" regression once we find our solution.

In practice, this means that we're assuming that no particular variable is likely to affect the output very much.

- A small amount of change in the output would still be expected, because of random noise.

So, we start off assuming that we can't make a more detailed prediction, and allow the data to try to convince us otherwise.

- If the effect is small, then we're more sure that it's just noise.
- But, if the effect is large, it'll override the regularizer, and we expect that variable x_i really is affecting y.

2.6.4 A second benefit: avoid error amplification

Let's demonstrate another possible problem: a large θ_k can "amplify" the error due to randomness.

Let's see how this can happen.

- 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 $y^{(1)} = 25$, $y^{(2)} = 26$.

Same variable x_1, but two different datapoints $i = 1, i = 2$

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.

$$\frac{dy}{dx} = \frac{\Delta y}{\Delta x} = \frac{1}{.01} = 100 \tag{2.54}$$

This suggests that x_1 has a **100x** effect on our output!

$$\theta_1 = 100 \tag{2.55}$$

$$100x_1 + 25 = h(x) \tag{2.56}$$

This model perfectly fits our data.

- But in reality, it's possible (and pretty common in real data) for y to change a bit (for example, ± 3) under the exact same circumstances, just because of **randomness**.
- If this is true, y only changed by 1. We probably shouldn't expect this change to give us much information!

Now, we have a problem. Even if x_1 never mattered, we think it does. We see the effect for our new data point, $x_1^{(3)} = 2$.

$$100(2) + 25 = 225 \tag{2.57}$$

If we compare to $x^{(1)}$, our result has changed by 800%. But if x_1 didn't really matter, and everything else is the same, then we've made a big mistake: $y^{(3)} \approx 25$.

This is what we mean by **error amplification**: a **small** difference in the input of $+2$ becomes a **very large** error in the output of $+200$.

Concept 37

Regularization guards agains the problem of **error amplification**.

Suppose the input x_1 changes by a tiny amount, and the output y changes by a larger (but still small) amount.

- In this situation, x_1 looks like it's a very important variable!
- θ_1 becomes large.
- This can happen even if it's just random chance.

If x_1 changes by a larger amount (again randomly), then you get a **huge change** in the **output**: this can cause a **huge error**!

Thus, the **error** resulting from random noise has been **amplified**.

If we bias θ_1 towards remaining smaller, then we avoid this problem.

Regularization biases against this kind of issue. Of course, including more data also helps fight overfitting.

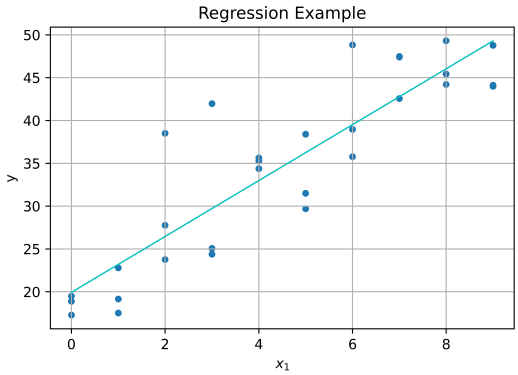
2.6.5 Why not include θ_0?

One thing you might immediately notice is that we used the magnitude of θ instead of Θ: this omits θ_0. Why would we do that?

We'll show that we need to **allow** the **offset** to have whatever value works best, and we shouldn't **punish** it.

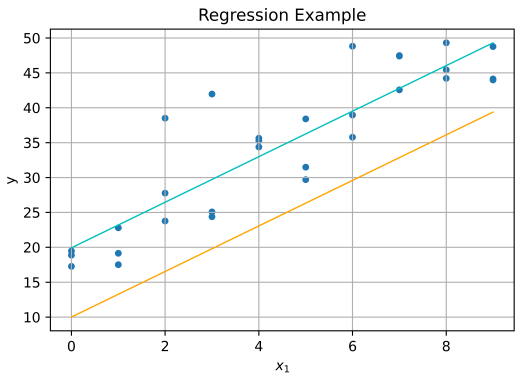
For simplicity, we won't do any regularization here: we can make our point without it.

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 **push** for a **much lower** (offset) θ_0 term, while keeping everything else the **same**:



Reducing our offset pulls our line further away from all of our data! That's not helpful.

And regularizing θ_1 wouldn't make this any better: it would just be flatter.

This shows that we **need** our offset! We use it to **slide** our hyperplane around the space: if all of our data is **far** from $(0, 0)$, we need to be able to **move** our **entire line**.

So, we'll keep θ_0 **separate** and **allow** it to take whatever value is **best**.

Concept 38

We **do not regularize** our **offset** term, θ_0.

Instead, we allow θ_0 to **shift** our hyperplane wherever it **needs** to be.

The other terms θ control the **orientation** of the hyperplane: the **direction** it is **facing**. We **regularize** this to push it towards less "complicated" orientations.

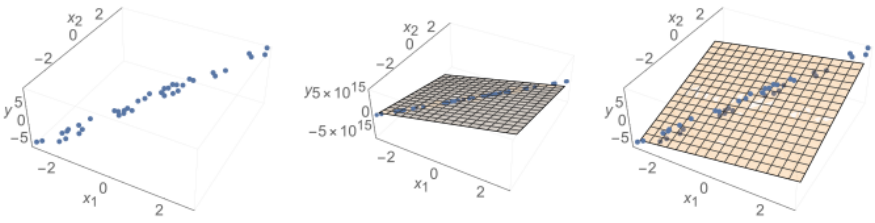
This will be discussed more in-depth in the Classification chapter!

2.6.6 A third benefit of regularization

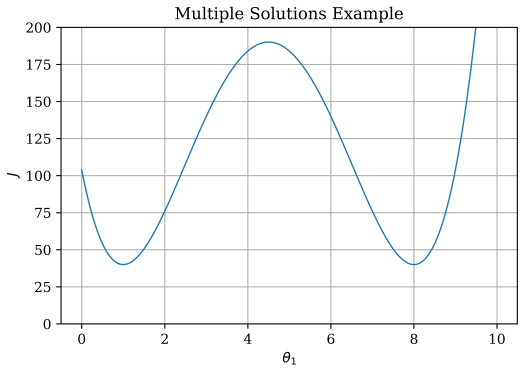
Another benefit of regularization is that it solves a more abstract problem: having **multiple optimal solutions**.

If we have **multiple** best outcomes, we have to pick one of them. We can make this choice by **picking** the one with the **smallest** magnitude.

We can **visualize** the problem of "multiple best solutions" a couple different ways:



There are many **planes** that can go through this line: multiple equally good solutions!



This compares different hypotheses (θ_1) and sees how well they perform (J): two are equally good!

Either way, we can pick a solution based on lowest θ **magnitude**!

2.6.7 A Math Perspective: Unique Solutions

We can also view this problem more **mathematically**.

Let's look at our **analytical** solution:

$$\theta = \left(XX^T\right)^{-1} XY^T \tag{2.58}$$

This solution only works if $\left(XX^T\right)^{-1}$ is **valid**. But we have a problem: **not all matrices** have **inverses**.

If XX^T has a **determinant** of **zero**, then we cannot find an inverse.

Without an inverse, we have **no unique solution**! This is a problem.

This is an important idea in linear algebra! If you don't know what this means, here's a great video.

This is one thing our **regularizer** $R(\Theta)$ helps us solve: we'll see that our **new solution** will not have this problem!

The reason will be clear in the **algebra**, but it's **equivalent** to the reason we discussed the above: we take the best **models** that are all **equally good**, and pick the one with **lowest**

magnitude.

Concept 39

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**.

2.6.8 Lambda, a.k.a. λ

We now have a term that can help us choose a more **general** hypothesis. One important question is, **how general** do we want it to be?

The more general we make our model, the **less specific** to our current data it is. This may seem like a good thing, but too much can make our model **worse**!

If λ is **too large**, then your model will stay **very close** to $\|\theta\| = 0$. This probably isn't a good solution for most cases.

But if it's **too small**, then it **won't** have enough of an **effect**. So, we need to be able to adjust how **much** we're regularizing.

To do this, we will **scale** our regularizer by a **constant** factor, λ.

Definition 40

Lambda, or λ, is the constant we **scale** our **regularizer** by.

It represents **how strongly** we want to regularize: how much we prioritize **general** understanding over **specific** understanding.

2.6.9 Our new objective function

Now that we have our regularizer,

$$R(\Theta) = \lambda \|\theta\|^2 \tag{2.59}$$

We can add it to our objective function:

Key Equation 41

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

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

This is the form we will **solve**.

2.6.10 Matrix Form Ridge Regression

Just like before, we'll switch from a **sum** to a **matrix** in order to solve this problem.

Creating an **equation** for both θ and θ_0 is, frankly, annoying to **derive**. **Instead**, we'll cheat a little, and keep θ_0 in and create our **matrix-form** objective function:

$$J = \frac{1}{n}(\tilde{X}\theta - \tilde{Y})^T(\tilde{X}\theta - \tilde{Y}) + \lambda(\theta^T\theta) \tag{2.60}$$

Our work begins. Let's take the **gradient**: what we want to set to zero.

$$\nabla_\theta J = \frac{2}{n}\tilde{X}^T(\tilde{X}\theta - \tilde{Y}) + 2\lambda\theta = 0 \tag{2.61}$$

We do some algebra and **solve** as we do in the **official notes**:

Key Equation 42

The **solution** for **ridge regression optimization** is

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

Or, in our original notation,

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

2.6.11 Our new term, $n\lambda I$

So, we already established that **regularization** helps us create more **general** hypotheses that are lower in magnitude.

But, how does this **mathematically** solve our invertibility problem?

$$\theta = (XX^T + n\lambda I)^{-1}XY^T \tag{2.62}$$

This term, $n\lambda I$, is added to the matrix we want to invert. Let's see what this matrix looks like. We'll use a (3×3) example:

I is the **identity matrix** in our notation.

$$n\lambda I = n\lambda \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = n \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix} \tag{2.63}$$

This visual, having a "**ridge**" of λs along the diagonal, is why we call it **ridge regression**.

2.6.12 Invertibility

This term $n\lambda I$ **shifts** the values of XX^T so that we **avoid** having a **determinant of zero**.

Since the **determinant is nonzero**, we don't have to worry about an **uninvertible matrix**: we now have a **unique** inverse, and thus a **unique** solution.

Concept 43

Ridge Regression solves the problem of **matrix invertibility** (non-unique solutions) by adding a term $n\lambda I$, our **ridge** of diagonals.

This turns the inverse $(XX^T)^{-1}$ into

$$(XX^T + n\lambda I)^{-1}$$

Which can prevent a **determinant** of zero in our solution, given $\lambda > 0$.

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**.

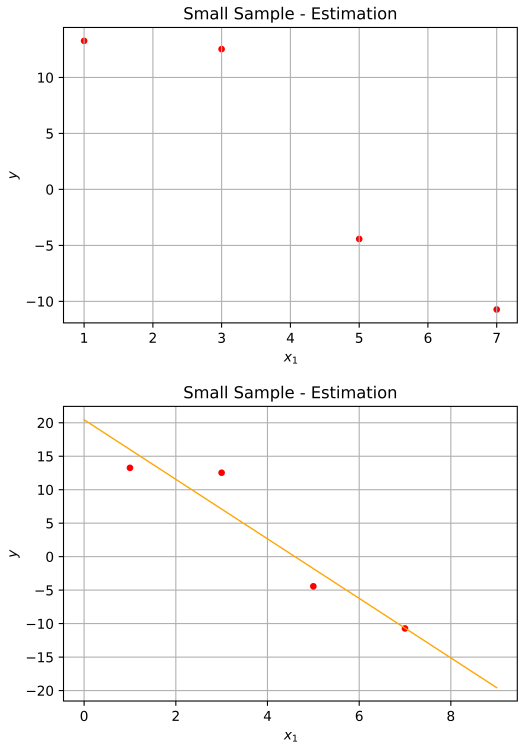
When we generalize, we're trying to avoid **estimation error**: we incorrectly guess the overall distribution we're trying to fit. We **estimate** poorly if we **generalize** poorly.

Definition 44

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

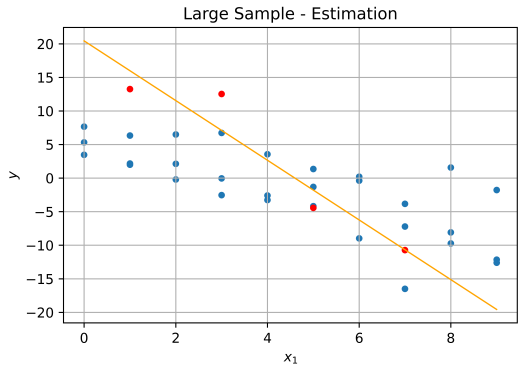
This can be caused by **overfitting**, getting a bad (**unrepresentative**) sample, or not having enough **data** to come to conclusion.

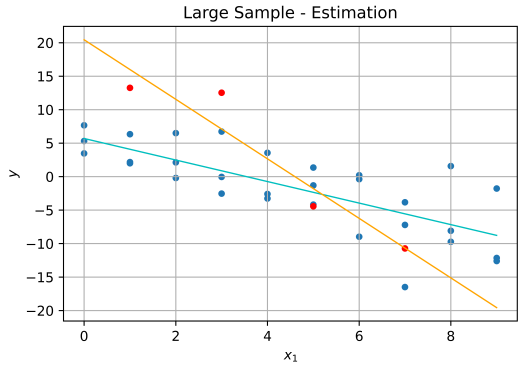
Example: Let's try a regression problem, but we'll use only 4 points to make our plot.



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 45

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**.

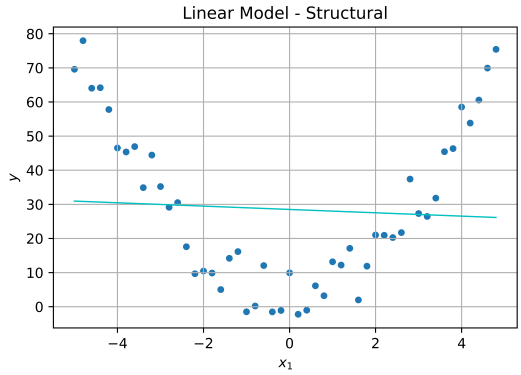
Definition 46

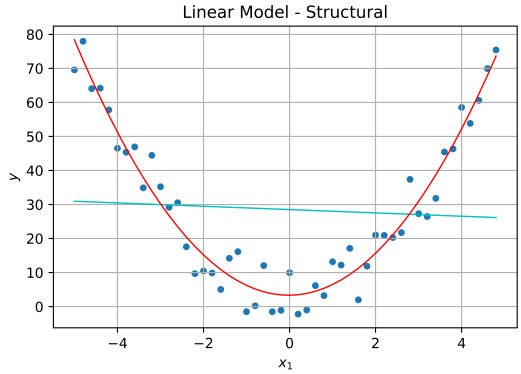
Structural error is the error that results from having the wrong **structure** for the **task** you are trying to accomplish.

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 47

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 48

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 49

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.

$$\mathcal{E}(h) = \frac{1}{m} \sum_{i=n+1}^{n+m} \left(h(x^{(i)}) - y^{(i)} \right)^2 \tag{2.64}$$

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 50

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 51

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 52

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 53

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 54

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 55

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?

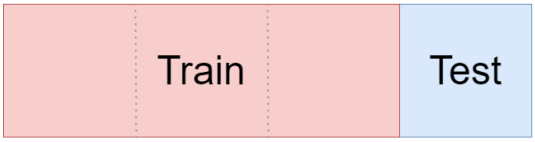
Data is often **expensive**. It might even be impossible to get more!

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?

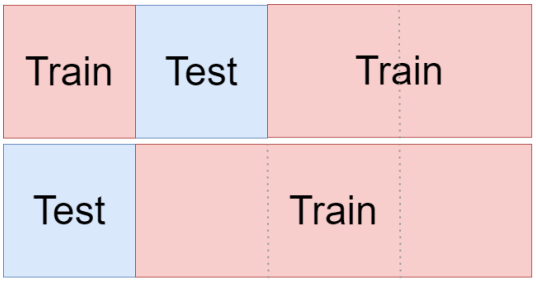
2.7.11 Cross-Validation

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 56

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 57

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 58

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

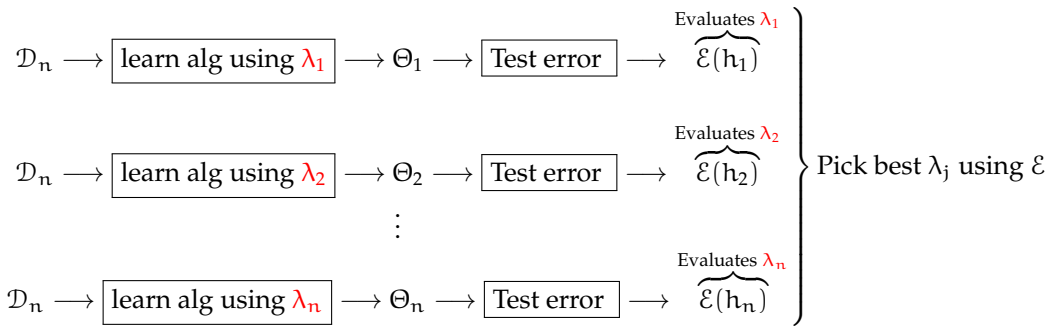
There's something often **confusing** about hyperparameter tuning to students:

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 59

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