# Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2022

# Contents

CHAPTER 4

# Classification

### 4.0.1 Regression (Review)

In chapter 2, we handled the problem of **Regression**: taking in lots of data (stored as a **vector of real numbers**), and returning another single **real number**.

Remember that we used a $(d \times 1)$ column vector for our data points $x^{(i)}$.

$$h_{reg} : \mathbb{R}^d \to \mathbb{R} \tag{4.1}$$

This was good for when we wanted to predict some **numeric** output: stock prices, height, life expectancy, and so on.

But, this isn't the **only** type of problem we might have to deal with.

## 4.1 Classification

### 4.1.1 Motivation: Putting things into classes

We don't *always* want a **real number** output: that can be complicated, or not match some problems well.

Often, it's more useful to, rather than give exact values, sort things into **categories**, or what we will call **classes**.

**Definition 1**

A **class** is **set** of things that have something relevant in **common**.

**Example:** A beagle and a golden retriever could both be put in a **class** called "**dog**". This is useful if you just want to know whether you have a dog or not!

### 4.1.2 What is classification?

This is the goal of **classification**: we want to take lots of **information**, and use them to **predict** what **class** a data point belongs in.

**Definition 2**

**Classification** is the **machine learning problem** of sorting items into different, **discrete** classes.

In this setting, we take **real-valued data**, stored in a $(d \times 1)$ **vector**, and return one of our **classes**.

$$h : \mathbb{R}^d \rightarrow \{C_1, C_2, C_3, \dots C_n\}$$

Where $\{C_1, C_2, C_3, \dots C_n\}$ are all **classes**. Sometimes, we call the value we return a **label** instead.

As a refresher, the function notation here just says, "take in a d-dimensional vector, and output one of our $n$ discrete classes."

**Example:** You might classify different **animals** as a bird, a mammal, or a fish. You have 5 pieces of useful data to **classify** with.

$$h : \mathbb{R}^5 \rightarrow \{\text{Bird}, \text{Mammal}, \text{Fish}\} \tag{4.2}$$

**Classification** can be useful for lots of situations:

- **Deciding** which **action** to take in a difficult situation
- **Diagnosing** a patient, and determining the best **treatment**
- **Sort** information to be **processed** later
- And more!

Just like with regression, we can depict our **hypothesis** as the function

$$x \rightarrow \boxed{h} \rightarrow y \tag{4.3}$$

**Concept 3**

**Classification** is also **supervised**: meaning, you have **training** data $\mathcal{D}_n$ with the **correct** answers given:

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

In **unsupervised** problems, you're not told the "correct" answer and have to just guess one!

### 4.1.3 Important Facts about Classes

There's a few important things we should remember about classes moving forward.

- Classes are **discrete**: each class is a distinct "thing", **separate** from other classes.
  - This is unlike real numbers, which are **continuous**: you can **smoothly** transition between them.
- This isn't **always** true, but usually, classes are **finite**: there are only so many of them, which we write as $n$.
  - Meanwhile, there are **infinitely many** real numbers.
- These classes may not have a natural **order**: is there a correct way to order "$\{\text{Bird}, \text{Mammal}, \text{Fish}\}$"? Not really.
  - The real numbers are ordered, too.
- In some problems, you get to **decide** what classes are acceptable: where do you draw the **line** between two categories? Do you care about dogs, or just mammals? And so on.
  - You can change units, but the **structure** of the real numbers is very **consistent**.

**Concept 4**

**Classes** are

- **discrete**
- **finite** (usually)
- **not** necessarily **ordered**
- often **defined** based on your **needs**

### 4.1.4 Binary Classification

So, how do we get **started**? Well, we want to create the **simplest** case, and maybe we can get the **general** idea.

**Two** is the **smallest** number of useful classes: often, this boils down to a **yes-or-no** question. Typically, we **represent** these two choices as $+1$ and $-1$, respectively.

**Definition 5**

**Binary classification** is the **problem** of sorting elements into one of **two categories**. Often, these categories are defined by a "**yes-or-no**" question.

$$h : \mathbb{R}^d \rightarrow \{-1, +1\}$$

**Example:** You could look at a person and say, "are they sick?" or, "is that a dog"? You can **classify** data in a binary way **based** on those questions.

### 4.1.5 Classification Performance

And how do we measure how well this model is doing? The easiest way might be, "count the number of wrong guesses".

This is captured by **0-1 Loss**:

**Definition 6**

**0-1 Loss** is a way of measuring **classification** performance: if you get the **wrong** answer, you get a loss of **1**. If you're **right**, then **0** loss.

$$\mathcal{L}(g, a) = \begin{cases} 0 & \text{if } g = a \\ 1 & \text{otherwise} \end{cases}$$

This type of loss is as **simple** as we can get: similar to counting how many wrong answers you get on a **multiple-choice** test.

If we want to get our training error, we'll just average over the data points:

$$\mathcal{E}_n(h) = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 0 & \text{if } g = a \\ 1 & \text{otherwise} \end{cases}$$

Just like before, we care about **testing loss** more than **training loss**: we want our model to **generalize**.

This relies on our typical IID assumption from chapter 1.

$$\mathcal{E}(h) = \frac{1}{m}\sum_{i=n+1}^{n+m} \begin{cases} 0 & \text{if } g = a \\ 1 & \text{otherwise} \end{cases}$$

Next, we figure out what **model** we use to do our classification.

## 4.2 Linear Classifiers

If you wanted to break up your data into two parts (+1 and −1), how might you do it? Let's explore that question.

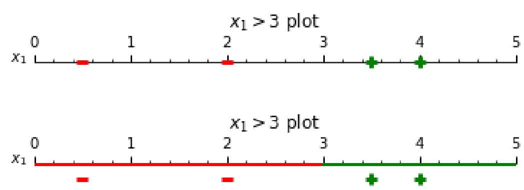
### 4.2.1 1-D Linear Classifiers

As usual, we'll start with the **simplest** case we can think of: 1-D. So, we only have one variable $x_1$ to **classify** with.

The simplest version might be to just **split** our space in **half**: those above or below a certain **value**. This is our parameter, C.

$$x_1 > C \qquad \text{or} \qquad x_1 - C > 0 \tag{4.4}$$

**Example:** For the below data (where green gives positive and red gives positive), could classify positive as $x_1 > 3$.




We plot everything above $x = 3$ as **positive**, and **negative** otherwise.

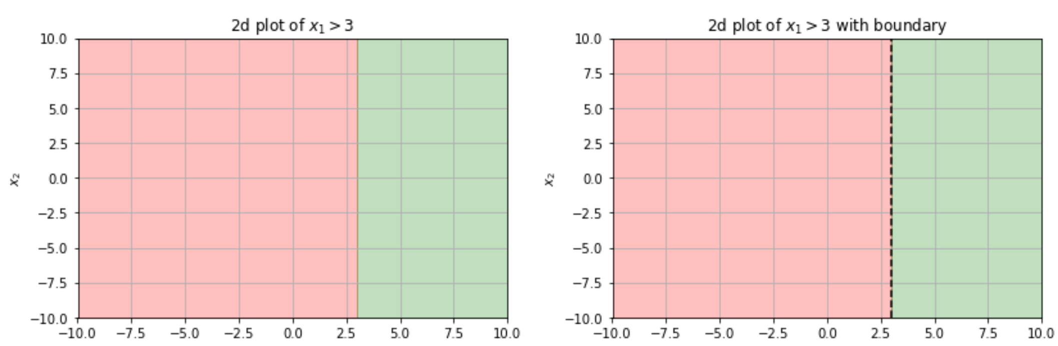
We could also call it $\theta_0$, in the spirit of our $\theta$ notation for parameters.

$$x_1 + \theta_0 > 0 \tag{4.5}$$

### 4.2.2 1-D classifiers in 2-D

Let's add a variable and see how our classifier looks on a 2-D plot.

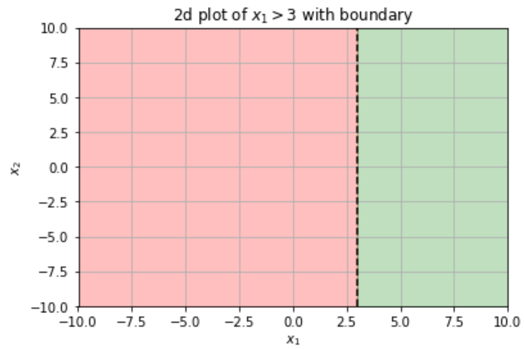
We'll omit the data points for now.

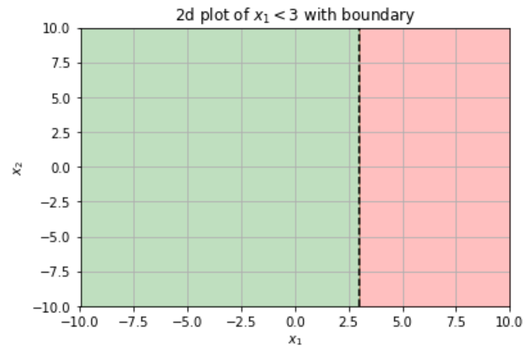



On the right, we've drawn the **dividing** line between our two regions.

Interesting - the **boundary** between positive and negative is defined by a **vertical line**.

Or, almost. Compare $x_1 > 3$ and $x_1 < 3$:

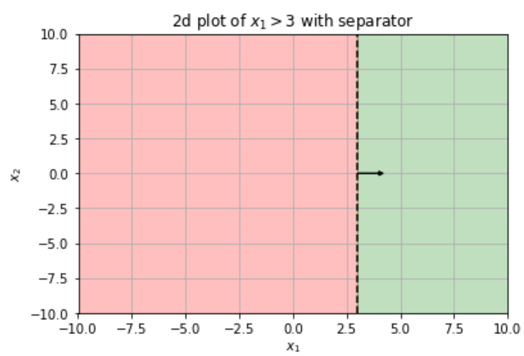


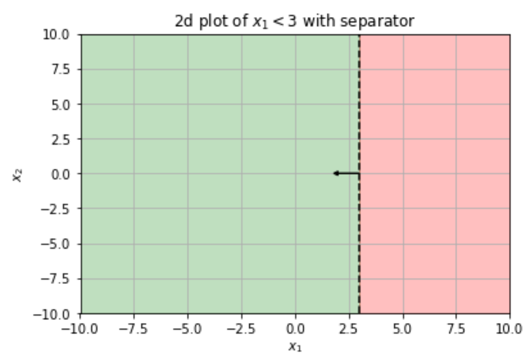




These two plots have the same line, but have their sides flipped.

So, we have a **line** that gives us the boundary, but we **also** need to include information about which way is the **positive** direction.

What tool best represents **direction**? We could use angles, but we haven't used that much so far. Instead, let's use a **vector** to **point** in the right direction.




Now, it's clear which plot is which, just using our **line** and **vector**!

The object that represents our classification is called a **separator**!

Since our variables are $x_1$ and $x_2$, this is a separator in **input space**.

**Definition 7**

A **separator** defines how we **separate** two different classes with our **hypothesis**.

It includes

- The **boundary**: the **surface** where we **switch** from one **class** to another.
- The **orientation**: a **description** of which **side** of the boundary is assigned to **which class**.

For example, let's take our specific separator from above.

We call it "orientation" because you could imagine "flipping over" the space, so the positive and negative regions are swapped.

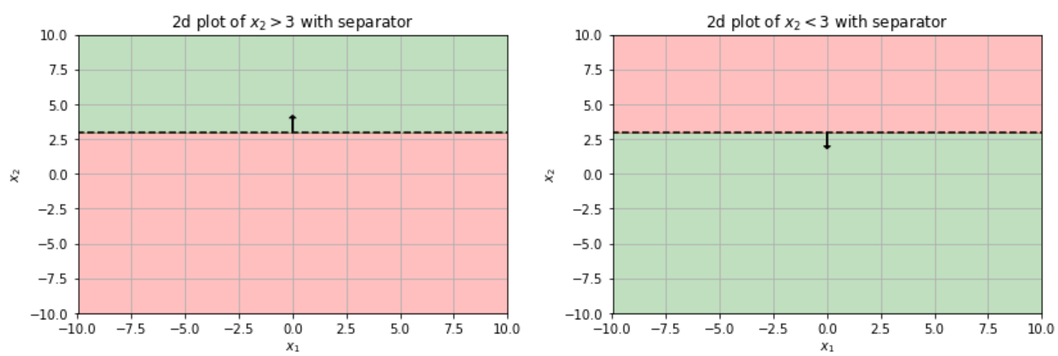
**Concept 8**

We can define our **1-D separator** using

- The **boundary** between the **positive** and **negative** regions: in 2-D input space, this looks like a vertical or horizontal **line**.
- A **vector** pointing towards whichever side is given a $+1$ **value**.

### 4.2.3 A second 1-D separator, and our problem

What if we use $x_2$ to **separate** our data?

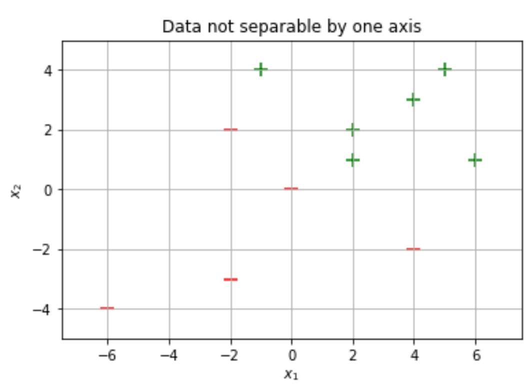



Instead of having a vertical separator, we have a **horizontal** one.

We get the same sort of plot along the **other axis**!

So, this is cool so far, but it's not a very **powerful** model: we can only handle a situation where the data is evenly divided by **one axis**.

And if that's the case, what's the point of our **other** variable?




There's no vertical or horizontal line we can use to split this space!

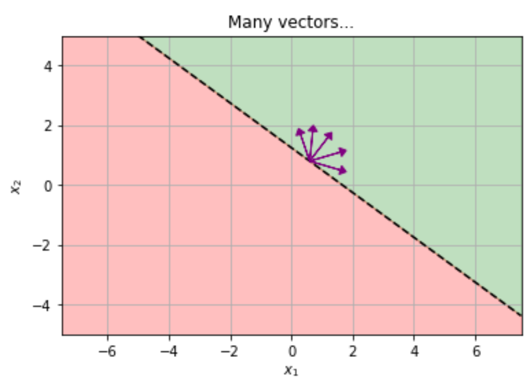
### 4.2.4 The 2-D Separator: What vector do we use?

Just looking at our example, we might wonder, "well, if we can use **vertical** lines or **horizontal** lines, can't we just use a line in **another** orientation?

It turns out, we **can**!


If allow lines at an angle, we can classify all of our data correctly!

So, we've got our **boundary**. But we still need a vector to tell us which side is **positive**. But there are **many** possible vectors we could choose:




All of these vectors point towards the **correct** side of the plane. Is there a **best** one to use?

Above, we used the vector that was **vertical** or **horizontal**. This makes sense: if we're doing $x_1 > 3$, it seems reasonable to have the arrow **point** in the positive-$x_1$ direction.

But this vector also happened to be **perpendicular** to our **line**: this is the line's **normal vector**, $\hat{n}$. This vector has a couple nice properties:

- It is **unique**: in 2-D, there is only 1 **normal** direction. The opposite side is just $-\hat{n}$.
- It points directly **away** from the plane.
- If our plane is at the **origin**, any point with a **positive** $\hat{n}$ component is on the **positive** side. This will be important later!

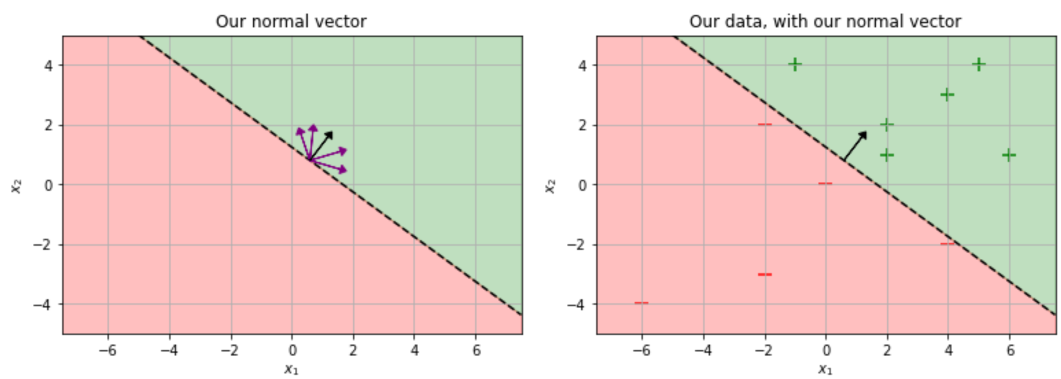
So, we have a **unique** vector that tells us which side is **positive**. Let's go with that!

**Concept 9**

Every **line** in 2-D has a **unique normal vector** that can be used to **define** the **angle/direction** of the line.

The **direction** the vector is "facing" is also called the **orientation**.

Our normal vector for the above separator:




We can define our plane using the **normal** vector!

It's clear that this vector in some way is a **parameter**: if we change this vector, we get a different **orientation**, and a different **classifier**.

We have **represented** parameters in the past using $\theta$. We need **two** different $\theta_k$: one for the $x_1$ component, another for the $x_2$ component.

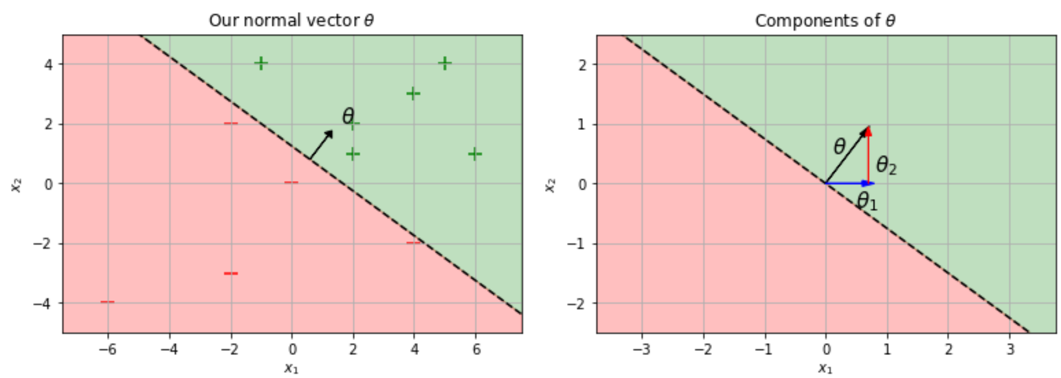
So, we'll use that.

**Notation 10**

The vector $\theta$ represents the **normal vector** to our line in 2D.

$$\hat{n} = \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$

We add this to our diagram:

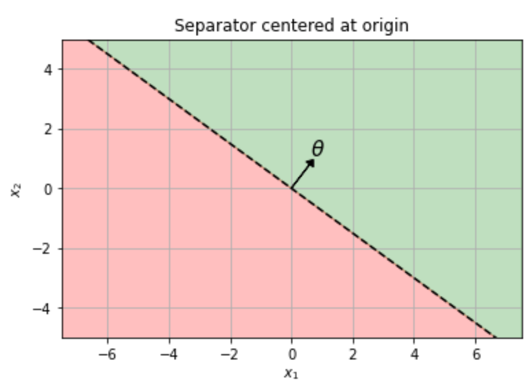



θ is our normal vector!

Nice work so far. The next question is: how do we describe this separator **mathematically**?

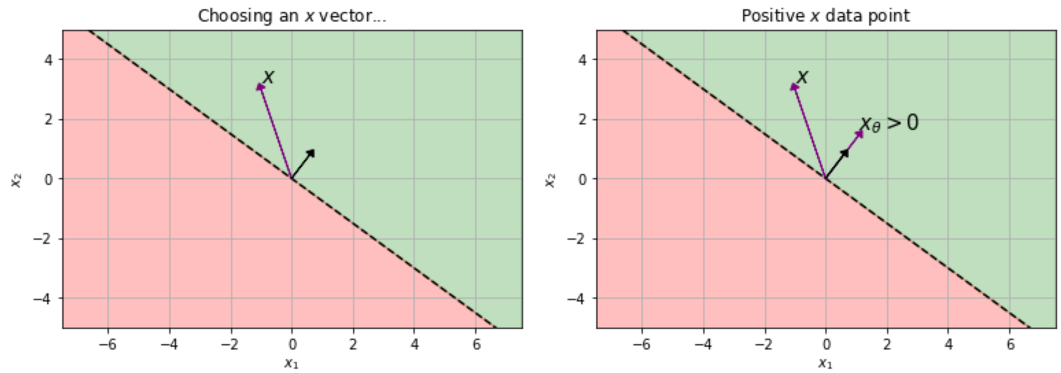
### 4.2.5 2D Separator - Matching components

As always, we'll **simplify** the problem to make it more manageable: for now, we'll assume our **separator** is centered at the **origin**.

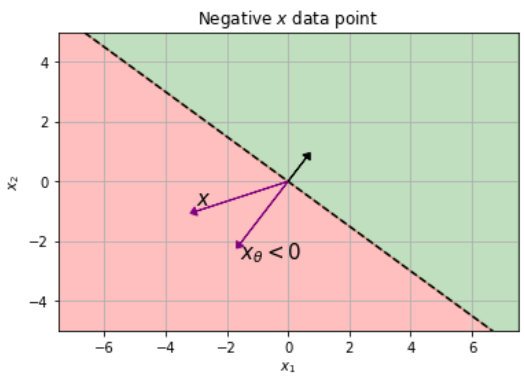



So, we have our vector, $\hat{n}$. As we mentioned above, anything on the **same** side as $\hat{n}$ is **positive**, and anything on the **opposite** side is **negative**.

For a line on the origin, "On the same side of the line" can be interpreted as "has a positive $\hat{n}$ component". We'll find that component next.




This vector has a **positive** component in the θ direction.




This vector has a **negative** component in the θ direction.

How do we represent "on the same side" mathematically? How do we **find** whether the component is **positive** or **negative**? We use the **dot product**.

### 4.2.6 The Dot Product (Review)

How to calculate the dot product should be familiar to you, but we'll talk about some **intuition** that you may not be exposed to.

**Concept 11**

You can use the **dot product** between unit vectors to measure their "**similarity**": if two vectors are more **similar**, they have a **larger** dot product.

In the most clear cases, take unit vectors $\hat{a}$ and $\hat{b}$:

- If they are in the **exact same** direction, $\hat{a} \cdot \hat{b} = 1$
- If they are in the **exact opposite** direction, $\hat{a} \cdot \hat{b} = -1$
- If they are **perpendicular** to each other, $\hat{a} \cdot \hat{b} = 0$

Remember, **unit vectors** have a length of 1.

What about non-unit vectors?

These unit vectors are then scaled up by the **magnitude** of each of our vectors. Because magnitudes are **always positive**, the dot product sign doesn't change.

**Concept 12**

You can use the **dot product** between non-unit vectors to measure their "similarity" **scaled by their magnitude**.

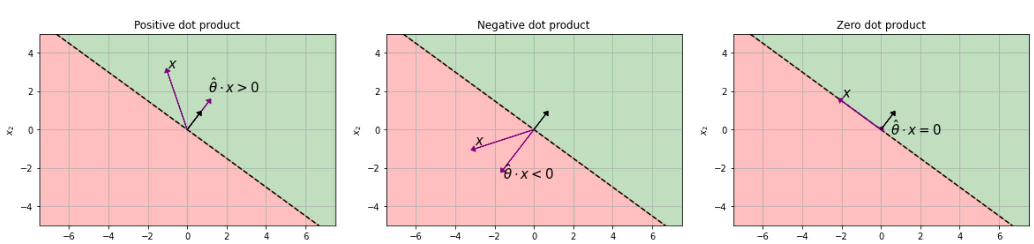
If two vectors are more **similar**, they have a **larger** dot product.

- If the vectors are **less** than 90° apart, they are more similar: they will share a **positive** component: $\vec{a} \cdot \vec{b} > 0$
- If the vectors are **more** than 90° apart, they will share a **negative** component: $\vec{a} \cdot \vec{b} < 0$
- If they are **perpendicular** (90°) to each other, $\vec{a} \cdot \vec{b} = 0$

### 4.2.7 Using the dot product

So, the **sign** of the dot product is a useful tool. If a point is on the line, it is **perpendicular** to $\theta$, our **normal vector**.

So, if a point has a **positive** dot product, it is on the **same side** as $\theta$, and if it's **negative**, it's on the opposite side.




Our various dot products can show us where in the space we are.

So, we can classify things based on the **sign** of it. Written as an equation, we can define the sign function:

**Key Equation 13**

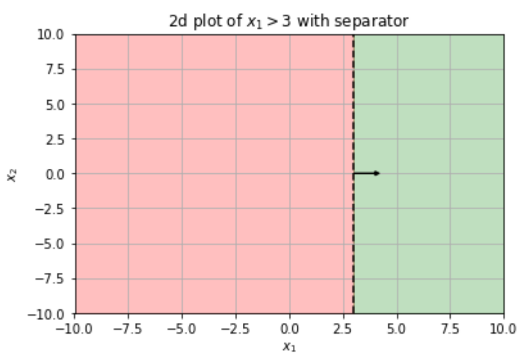
For a **linear separator** centered on the **origin**, we can do **binary classification** using the hypothesis

$$h(x;\theta) = \text{sign}(\theta \cdot x) = \begin{cases} +1 & \text{if } \theta \cdot x > 0 \\ -1 & \text{otherwise} \end{cases}$$

### 4.2.8 Introducing our offset

Now that we have handled the case where our linear separator is on the **origin**, we want to **shift** our separator **away** from it.

In our **1-D** case, we easily **shifted** away from the origin: any separator $x_1 > C$ where C **isn't zero**, we shift by C units.




By making our inequality $x_1 > 3$ **nonzero**, we moved away from the origin by 3 units!

We could make our inequality **nonzero**, then! That could move us **away** from the origin, just in a different **direction**.

Or, we could equivalently do this...

Note: $A \iff B$ means A and B are equivalent!

$$x_1 > 3 \iff x_1 - 3 > 0 \tag{4.6}$$

So, instead, we could just add a constant to our expression, which we will call $\theta_0$.

We'll also switch out $\theta \cdot x = \theta^T x$.

**Key Equation 14**

A general **linear separator** can do **binary classification** using the hypothesis

$$h(x;\theta) = \text{sign}(\theta^T x + \theta_0) = \begin{cases} +1 & \text{if } \theta^T x + \theta_0 > 0 \\ -1 & \text{otherwise} \end{cases}$$

Notice that this looks very similar to what we did in regression! We'll get into that in a bit.

First, a quick look at the components of our equation:

**Concept 15**

For **binary classification**, $\theta$ and $\theta_0$ entirely **define** our **linear separator**.

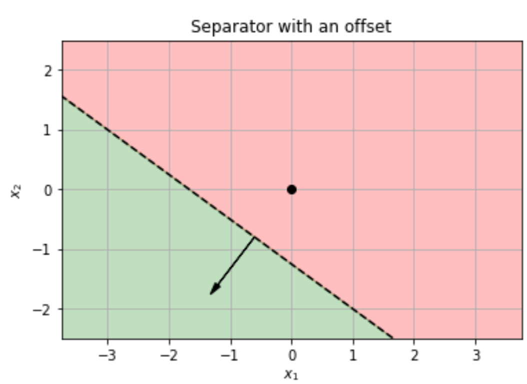
- $\theta$ gives us the **orientation** of our line.
- $\theta_0$ **shifts** that line around in **space**.

### 4.2.9 How does the offset affect our classifier?

So, how exactly does our offset $\theta_0$ affect our **classifier**? Well, we mark our classifier with our **normal vector** and the **boundary**.

Our **normal vector** is entirely captured by $\theta$: it's unchanged by $\theta_0$.

What about our **boundary**? We have its **orientation**, but we don't know where it has **shifted** to.




Note that the origin has been marked.

Well, let's use our equation: the **boundary** line is given by

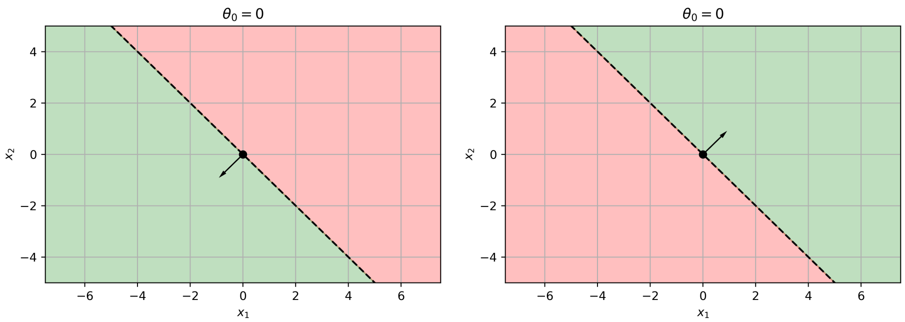
$$\theta^T x + \theta_0 = 0 \iff \theta^T x = -\theta_0 \tag{4.7}$$

We'll break the effects of $\theta_0$ into three cases:

Note: the below statements are true no matter what $\theta$ we choose!

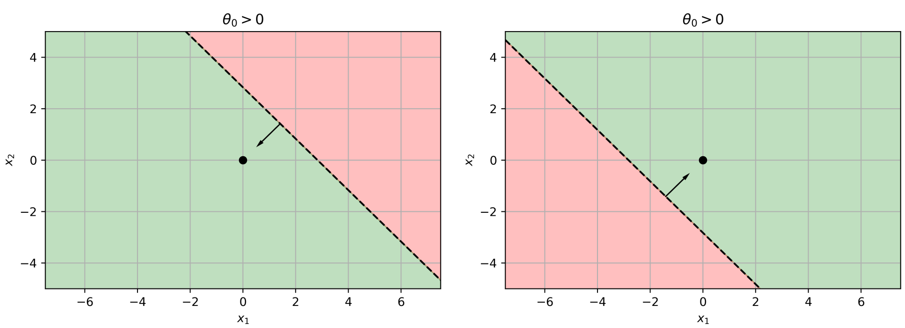
For each, we'll show two different $\theta$ values.

- If $\theta_0 = 0$, then $x = (0, 0)$ is **on the line**.
  - Without an **offset**, our line goes through the **origin**.



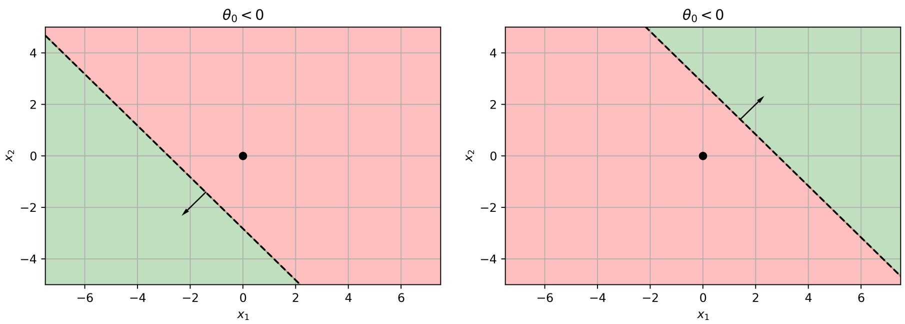

The boundary is on the origin.

- If $\theta_0 > 0$, then $x = (0, 0)$ is in the **positive** region.
  - That means the positive region is **larger**: the line must have moved in the $-\theta$ direction.




If we have a **positive** constant, it's "easier" to get a positive **result**: more positive space.

- If $\theta_0 < 0$, then $x = (0, 0)$ is in the **negative** region.
  - That means the positive region is **smaller**: the line must have moved in the $+\theta$ direction.




If we have a **negative** constant, it's "harder" to get a positive **result**: more negative space.

This can be a bit confusing, so we'll summarize:

**Concept 16**

The **sign** of our $\theta_0$ and the **direction** we move away from the origin are **opposite**.

If $\theta_0 > 0$ (positive), our boundary moves in the $-\theta$ **direction**.

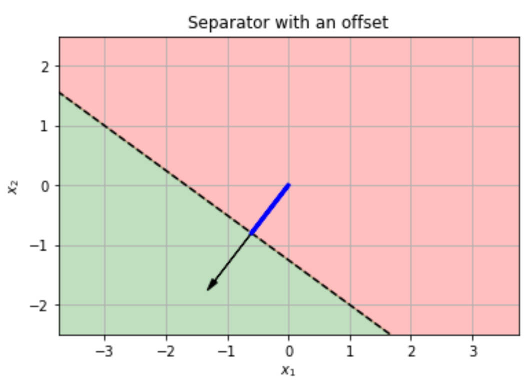
If $\theta_0 < 0$ (negative), our boundary moves in the $+\theta$ **direction**.

This gives us a general idea of how the offset affects it, but what is the **exact** effect of $\theta_0$ on the line?

We'll focus on one point on the line: the **closest point to the origin** We want to look at this **point** because it's **unique**.

Points that aren't unique are hard to keep track of!

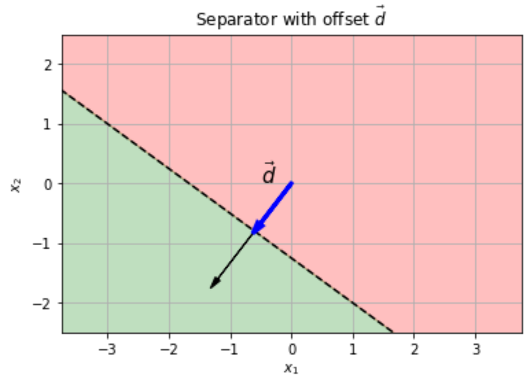
### 4.2.10 Distance from the Origin to the Plane




Notice that the **shortest** path from the origin to the line is **parallel** to $\theta$!

So, we can think of our **line** as having been **pushed** in the $\theta$ direction. This **matches** what we did for 1-D separators: $x_1 > 3$ was moved in the $x_1$ direction.

So, we'll take the closest point on the line, $\vec{d}$. The **magnitude** $d$ will give us the **distance** that the separator has been **shifted**.




Since $\vec{d}$ is in the direction of $\theta$, the direction can be captured by the unit vector $\hat{\theta}$. Let's take a look at that:

Remember, a vector is direction (unit vector) times magnitude (scalar).

$$\theta = \|\theta\|\hat{\theta} \tag{4.8}$$

$$\vec{d} = d\hat{\theta} \tag{4.9}$$

They're in the same **direction**, so they have the same **unit vector** $\hat{\theta}$.

$\vec{d}$ is on the **line**, so it satisfies:

We'll use $\theta \cdot \vec{d}$ instead of $\theta^T\vec{d}$ here.

$$\theta \cdot \vec{d} + \theta_0 = 0 \tag{4.10}$$

We can plug our equations 4.8 and 4.9, where we've separated magnitude from unit vector:

$$\overbrace{\left(\|\theta\|\hat{\theta}\right)}^{\theta} \cdot \overbrace{\left(d\hat{\theta}\right)}^{\vec{d}} + \theta_0 \tag{4.11}$$

We can move the scalars $\|\theta\|$ and $d$ out of the way of the dot product:

$$\|\theta\| d \left(\hat{\theta} \cdot \hat{\theta}\right) + \theta_0 \tag{4.12}$$

We know that $\hat{u} \cdot \hat{u} = 1$:

$$\|\theta\| d + \theta_0 = 0 \tag{4.13}$$

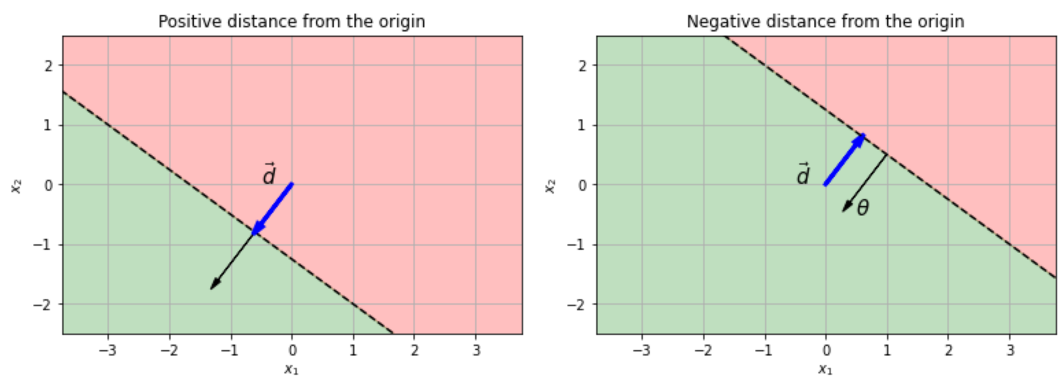
And now, we just solve for $d$:

**Concept 17**

The **distance** d from the **origin** to our **linear separator** is

$$d = \frac{-\theta_0}{\|\theta\|} \tag{4.14}$$

A "negative" distance means $\vec{d}$ (the vector from the origin to the line) is pointed in the opposite direction of $\theta$.




Notice, again, that this agrees with our **earlier** thought: the sign of $\theta_0$ is the opposite $(-1)$ of the $\theta$ direction we move in.

### 4.2.11 Extending to higher dimensions

We've now fully conquered the 2D problem! Now, we can move up in **dimensions**.

In terms of equations, the answer is simple, just like it is for regression: just add more terms to $\theta$.

**Key Equation 18**

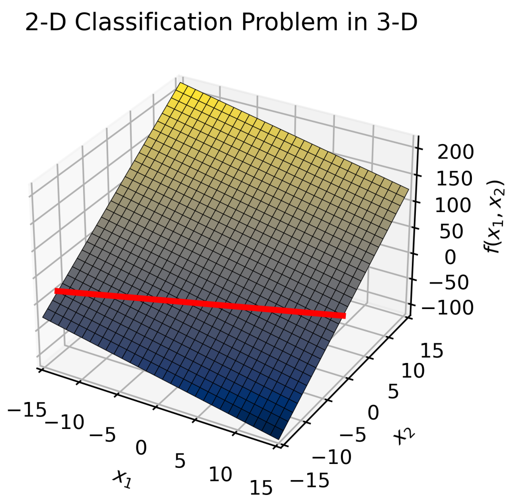
A general d-dimensional **linear separator** can do **binary classification** using the hypothesis

$$h(x;\theta) = \text{sign}(\theta^T x + \theta_0) = \begin{cases} +1 & \text{if } \theta^T x + \theta_0 > 0 \\ -1 & \text{otherwise} \end{cases}$$

Where

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix}$$

What about how it looks? Well, if we have 3 input variables, our line turns into a **plane**:




As we move into 3D, we can **separate** points on **three different axes**.

Just like with regression, this is when we introduce the **hyperplane**:

**Concept 19**

Our $n$-dimensional **linear separator** solution to the **binary classification** problem **splits** our space into two **halves**: a positive and a negative half.

The **surface** that **splits** space like this is a $(n-1)$-dimensional **hyperplane**.

The hyperplane is **oriented**: there is a **normal** vector $\theta$ which defines the **orientation** of the hyperplane, and which side is **positive**.

It also has an **offset** term $\theta_0$, that slides it in the $\theta$ direction **away** from the origin.

For any dimensional input, we can use hyperplanes as separators.

### 4.2.12 IMPORTANT: A difference between regression and classification

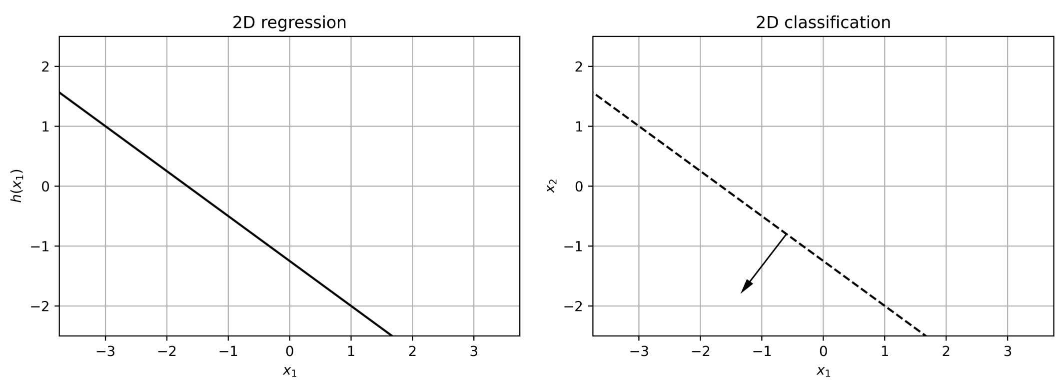
Here is an important misconception that comes up between regression and classification.

Both functions use the equation

$$\theta^T x + \theta_0 \tag{4.15}$$

So, one might think of them as interchangeable.

However, they are **not**. Why is that?




These two plots look almost the same, but represent completely different things!

Notice that these two plots are **both** plotted in 2-D, and both have a **line** plotted. But, they **aren't** as **similar** as they look.

Notice, for example, that the regression plot has **only** $x_1$, while the classification plot has $x_1$ **and** $x_2$.

The reason why? The **output**.

- In **regression**, the output is a **real number**: every point on that line represents an input $x_1$, and an output $h(x_1)$.

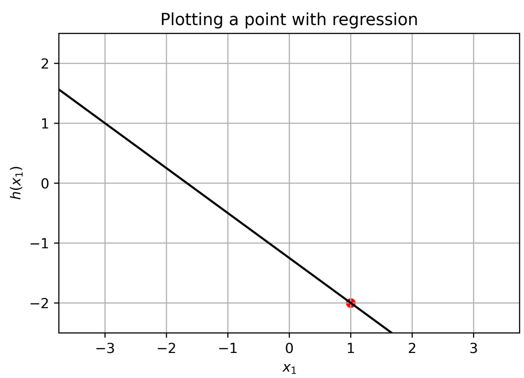
   – This plot can only contain **one** input variable: the **second** axis is reserved for the **output**!

- In **classification**, the output is **binary**. So, that line represents only the **values** where the output is $h(x) = 0$.

   – This plot can contain **two** input variables: $x_1$ and $x_2$. Rather than **displaying** the output, we only show one **slice** of the output: the $h(x) = 0$ slice.

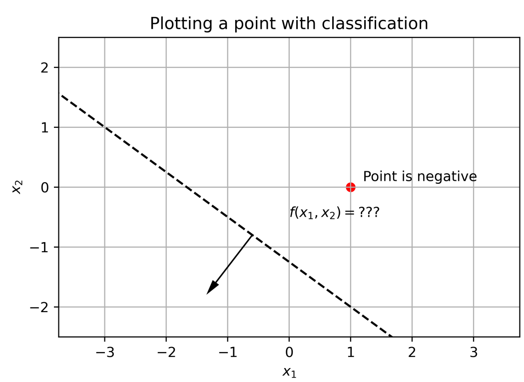
If we think in terms of $f(x) = \theta^T x + \theta_0$, we can compare them directly.

The regression plot shows the exact value on the y-axis. If we want to know what $f(x_1 = 1)$ looks like, we can check the plot: we just get $f(1) = -2$.




We have one input, and we get the exact value of our output.

But the classification plot **doesn't**! We aren't given the value of $\theta^T x + \theta_0$ at $x = (1, 0)$: we just know that it's **negative**.




We have two input, and we **don't** get the exact output.

If we wanted to know the exact value of our 2-D classification, we would need to view it as a plane in 3-D space.

This is the trade-off between these two plots: one gives more information about the output, and the other allows for more inputs in a lower dimension.

**Clarification 20**

**Regression** and **classification** plots that look the same, have **different functions**:

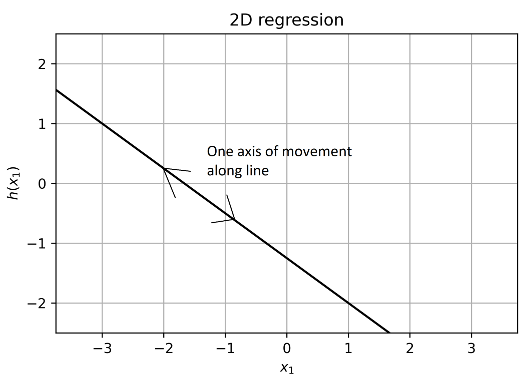
When looking at the output of $f(x) = \theta^T x + \theta_0$,

- A **regression** plot gives the **exact numeric** $f(x)$.
- A **classification** plot only gives the **sign** of the $f(x)$.

When plotting $n$ inputs,

- A **regression** plot uses a $d + 1$ dimensions ($d$-dim hyperplane) to plot: +1 for the **output**.
- A **classification** plot only needs $d$ dimensions ($(d - 1)$-dim hyperplane): we only see the $f(x) = 0$ **hyperplane**.

Why do we need $d + 1$ dimensions to plot a $d$-dimensional **hyperplane**? You can think of it this way: a **line** in 2-D space is a 1-D **hyperplane**: we have only **one axis** we can move on the line.




Our plot is 2-D, but we can only move along one axis on our line!

Because of these differences, $\theta$ also acts differently:

**Clarification 21**

$\theta$ appears differently in 2-D regression and classification:

- In **2-D regression**, $\theta$ is the **slope** of the line

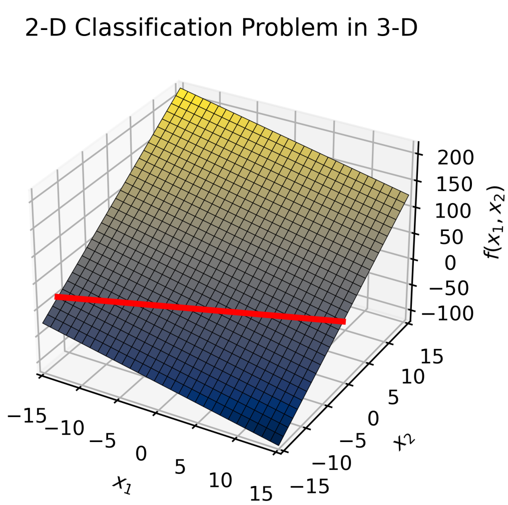
$$h(x) = \theta x + \theta_0 \tag{4.16}$$

- In **2-D classification**, $\theta$ is the **normal vector** of the line

$$0 = \theta^T x + \theta_0 \tag{4.17}$$

### 4.2.13 3d plot of 2d separator

For additional understanding, you might view the full output of $\theta^T x + \theta_0$, before we simplify the output to $\{-1, +1\}$.




The red line represents where $f(x) = 0$. The black line is our **normal vector**: notice that it's normal to the **line**, not the **plane**.

We mentioned before that, if we wanted to show the exact value of $f(x)$ for our 2-D classifier, we'd need a 3-D plot (just like for regression).

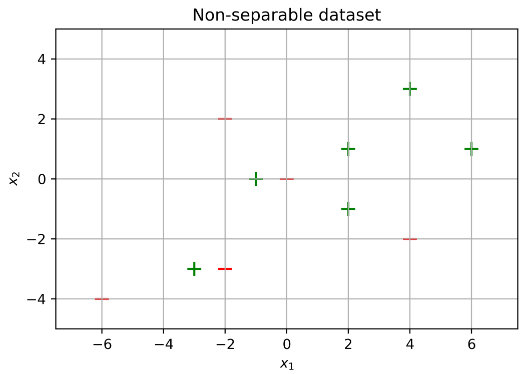
So here, we've done **exactly** that: the **height** is the output of $h(x)$.

But, because we don't **care** about the **exact** output in classification, we usually only graph the **red line**: where $f(x) = 0$.

This shows how we're taking a 2D slice ($f(x) = 0$) out of a 3-D plot (full hyperplane), to **save** on one dimension of **plotting**.

### 4.2.14 Separable vs Non-separable data

One more consideration: **not all** data can be correctly **divided** by a linear separator!




There's no line we could draw through this data to **separate** the points from each other.

If we can, we call it **linearly separable**.

**Definition 22**

A **dataset** is **linearly separable** if you can **perfectly** classify it with a **linear classifier**.

A couple common reasons for data to not be linearly separable:

- A positive and negative data point have the exact **same position** in input space.
- Two points on either **side** of a point with opposite classification: $+-+$ or $-+-$, for example.

Very often, real-world datasets **can't** linearly separated, because of **complexities** in the real world, or random **noise**.

But, sometimes, we can **almost** linearly separate it: we get very high **accuracy**. In those cases, it may be **fine** to use a linear separator: we might risk **overfitting** if we use a more complex model.

What is "high enough accuracy"? Depends on what you need it for!

Still, if a dataset is not **linearly separable**, or at least **high-accuracy** with a linear separator, that could mean we need a **richer** hypothesis class.

Remember: a "richer" or more "expressive" hypothesis class is one that can create more hypotheses that our current one can't!

We'll get into ways to make a **richer** class in the **next** chapter: **feature transformations**.

## 4.3 Linear Logistic Classifiers

### 4.3.1 The problem

Now, our goal is to create a **good model** for our problem, **binary classification**.

To do this, we can **try** using our 0-1 loss $\mathcal{L}$:

$$J(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\text{sign}(\theta^T x^{(i)} + \theta_0), y^{(i)}) \tag{4.18}$$

The **first** thing to note is that there isn't an easy **analytical** solution, no simple **equation**: $\text{sign}(u)$ isn't a function that we can explicitly **solve**, like we could for **linear regression**.

To be fair, this is true for most possible problems: most of them can't be solved analytically.

So, we refer to our other approach, **gradient descent**.

But in order to do that, we'll just need to get the **gradient**.

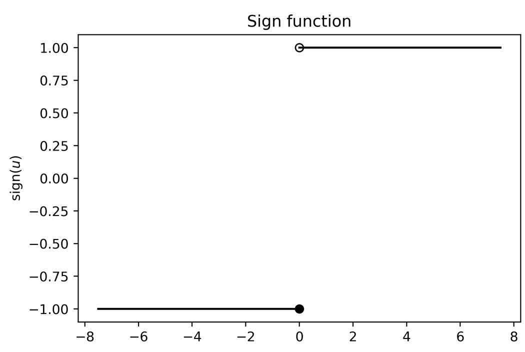
$$\nabla_\theta J = 0 \tag{4.19}$$

...Well that's not good.

Why not? Because we use our **gradient** to decide **how** to change $\theta$, if the gradient is 0, we'll never **improve** $\theta$ at all!

### 4.3.2 The real problem: sign$(u)$ is flat

What's going on here? Let's look at the sign function:




Sign is a flat function! The slope is 0 everywhere, except $u = 0$, where it's **undefined**.

Well, that explains why we can't use the gradient: the function is **flat**.

Another way to say this is that our function doesn't **tell** us when we're **closer** to being right.

There's **no difference** between being **wrong** by 1 unit or being wrong by 10 units: you can't tell if you're getting **closer** to a correct answer.

And the **gradient** doesn't tell you which way to move in **parameter space** to further improve.

Remember, parameter space is what we move through as we change our parameter vector $\theta$.

In fact, the best way we know how to approach this kind of problem takes **exponential** time: it takes exponentially **longer** to solve based on our **number** of data points.

That's way too **slow**. So, we'll have to come up with a **better** function: something to **replace** $\text{sign}(u)$, that still serves the same role.

**Concept 23**

The **sign function** is difficult to optimize, because it isn't **smooth**: not only is the slope undefined at 0, it is 0 everywhere else.

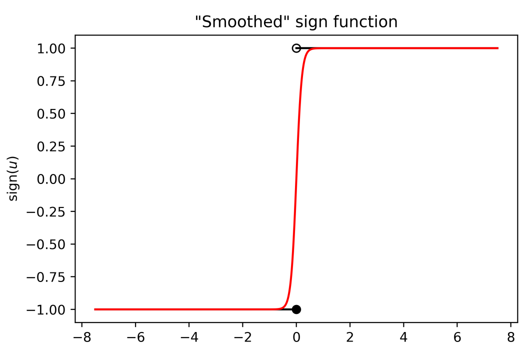
This causes two problems:

- We can't tell whether one **hypothesis** is **closer** to being **correct**, if it has gotten **better**, unless its accuracy has increased.
  - This makes it harder to **improve**.
- We can't indicate how **certain** we are in our answer: $\text{sign}(u)$ is **all-or-nothing**: we choose one class, with no information about how **confident** we are in our choice.
  - Knowing how **uncertain** we are can be **helpful**, both for **improving** our machine and also **judging** the choices or machine makes.

So, we need to explore a **new** approach: we'll **replace** $\text{sign}(u)$ with something else.

### 4.3.3 The sigmoid function

So, what do we **replace** sign with? We like the way sign **works** (choosing between two different classes based on a **threshold**), so maybe we want a **smoother** version of it.

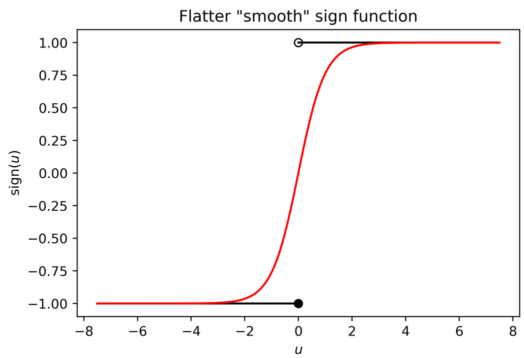



The red line shows a "**smoother**" sign function, that mostly behaves the same, while solving our problem.

This solves **one** of our two problems: the **gradient** is **nonzero**.

It's hard to see visually, but the function is **smooth**, and the slope is nonzero **everywhere**!

We could also make it less steep:




So, we need a **function** that accomplishes this. It turns out there are **several** that work: $\tanh u$, for example.

For our purposes, we'll use the following function:

**Definition 24**

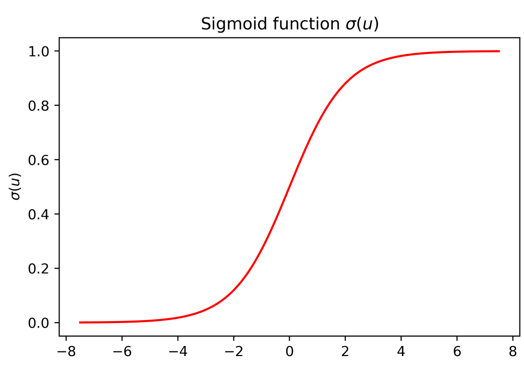
The **sigmoid** function

$$\sigma(u) = \frac{1}{1 + e^{-u}} \tag{4.20}$$

...is a nonlinear function that we use to **compute** the output of our **classification** problem.

It is also called the **logistic** function.

The function looks like this:




### 4.3.4 Sigmoid as a probability

Something you may **notice** is that $\sigma(x)$ is always between 0 and 1. But before, $\text{sign}(x)$ was **always** between -1 and +1. Why would we use *this* function?

Because going between 0 and 1 has a different advantage: we can interpret it as a **probability**.

Your **value** of $\sigma(u)$ can be stated as, "what does the machine think is the **probability** we **classify** this data point as +1".

And, on the **flip** side, $1 - \sigma(u)$ is the **probability** we **classify** as $-1$.

This solves the second problem we mentioned **earlier**: we can indicate how **confident** the machine is in its answer!

**Concept 25**

The output of the **sigmoid function** $\sigma(u(x))$ gives the **probability** that the data point $x$ is classified **positively**.

$$\sigma(u) = \mathbf{P}\{x \text{ is classified } +1\}$$

$$1 - \sigma(u) = \mathbf{P}\{x \text{ is classified } -1\}$$

Note that this works because $\sigma(u) \in (0, 1)$.

### 4.3.5 Logistic Regression

So, we've seen the benefits of switching from $\text{sign}(u)$ to $\sigma(u)$. So we'll do that:

We're using $u(x) = \theta^T x + \theta_0$

**Key Equation 26**

**Logistic Regression** is a **modification** of **linear regression**.

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

where

$$\sigma(u) = \frac{1}{1 + e^{-u}}$$

It outputs the **probability** of a **positive** classification.

If we **plug** this in, we get this slightly ugly expression:

$$h(x; \theta) = \frac{1}{1 + e^{-(\theta^T x + \theta_0)}}$$

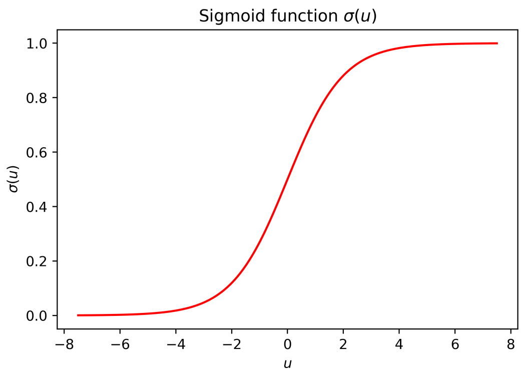
We have a problem, though: **logistic regression** is a... **regression** function. It takes in a real **vector**, and outputs a real **number**: $\mathbb{R}^d \to \mathbb{R}$.

We can't use this to do **classification**, where want $\mathbb{R}^d \to \{-1, +1\}$!

### 4.3.6 Prediction Threshold

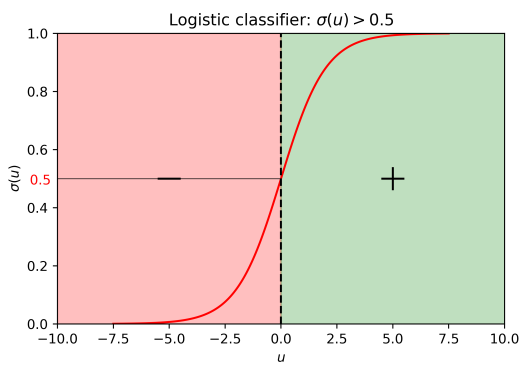
When we were just using $u(x) = \theta^T x + \theta_0$, we classified data points by saying whether $u(x) > 0$. Our boundary was $u(x) = 0$.

We can't quite do that here, because $\sigma(u) = 0$ is **impossible**: $\sigma(u)$ is **always** greater than 0.




$\sigma(u)$ approaches 0 as $u$ approaches $-\infty$, but it never reaches it.

Well, what happens when $u(x) = 0$? We get $\sigma(0) = .5$. So, we could use that as our classification: $\sigma(u) > .5$




But, we don't necessarily always want to use .5:

**Example:** Imagine if you wanted to **classify** whether someone needs **life-saving** treatment. Classify $-1$ if sick (they need it), $+1$ if healthy (they don't).

Let's say you got $\sigma(u) = .6$, so you're only 60% sure they **don't** need it. You'd classify that as $\sigma(u) > .5$: they're '**healthy**'.

Even so, you probably shouldn't **refuse** someone treatment that's 40% likely to **save** their life. We might not want to use $\sigma(u) > .5$ after all.

We call the **boundary** between positive and negative the **prediction threshold**.

**Definition 27**

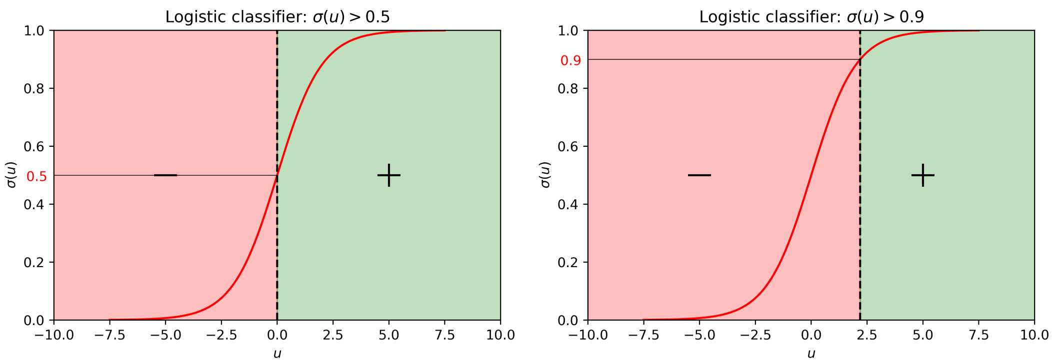
The **prediction threshold** $\sigma_{\text{thresh}}$ is the value where you go from **negative** classification to **positive**.

In general, we say

Our **default** value is a threshold of .5, but our threshold can be **anywhere** in the range

$$0 < \sigma_{\text{thresh}} < 1$$

**Example:** If $\sigma_{\text{thresh}} = .9$, we would see:




We switch from a .5 threshold to a .9 threshold.

### 4.3.7 Linear Logistic Classifier

This finally gives us our **linear logistic classifier** (LLC)

**Key Equation 28**

The **linear logistic classifier** is a **binary** classifier of the form

$$h(x;\theta) = \begin{cases} +1 & \text{if } \sigma(u(x)) > \sigma_{\text{thresh}} \\ -1 & \text{otherwise} \end{cases}$$

where

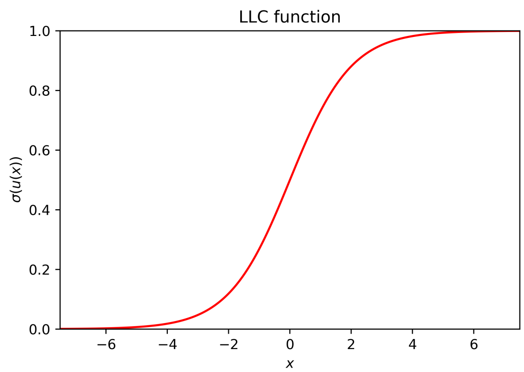
$$u = \theta^T x + \theta_0 \qquad\qquad \sigma(u) = \frac{1}{1+e^{-x}} \tag{4.21}$$

We call it linear because of the linear inner function $u(x)$, and logistic because of the outer function $\sigma(u)$.

### 4.3.8 Modifying our sigmoid

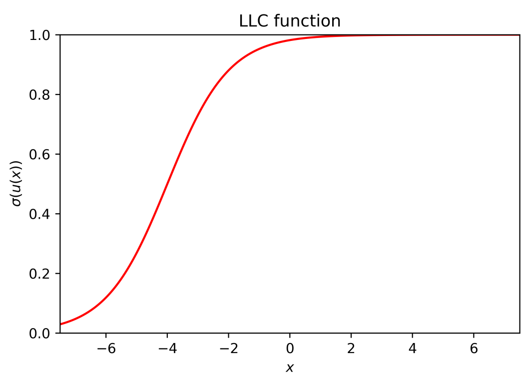
What happens when you modify the **parameters** of an LLC? Let's find out.

We'll use a 1-D input: our variables will be $\theta$ (scalar) and $\theta_0$: $\theta x + \theta_0$




Our baseline LLC: $u(x) = 1x + 0$

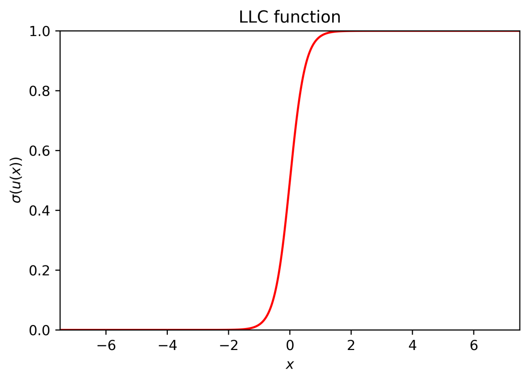
What if we shift by increasing $\theta_0$?




Our shifted LLC: $u(x) = 1x + 4$. $\theta_0$ shifts us along the $x$-axis!

Just like before, it **shifts** us in the **opposite** direction: if $\theta_0$ is **positive**, we shift in the **negative** direction, and vice versa.

What if we increase the magnitude of $\theta$!




Our new LLC: $u(x) = 4x$. Increasing $\theta$ makes our function steeper!

Making the magnitude of $\theta$ larger makes our function **change** faster.

This makes some sense: if $\theta$ (linear slope of $u(x)$) makes $u(x)$ **change** faster, it will make $\sigma(u)$ change faster **too**.

You can combine these changes as well: you can shift your LLC with $\theta_0$, and also make it steeper/less steep by changing magnitude of $\theta$.

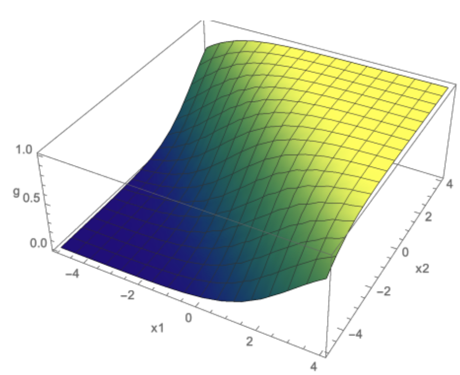
**Concept 29**

When working with **sigmoids**, you can **transform** them using your **parameters**:

- A higher **magnitude** $\|\theta\|$ makes the slope **steeper**, and answers more **confident**.
- **Increasing** $\theta_0$ **shifts** the sigmoid in the $-\theta$ **direction**, and vice versa.

### 4.3.9 Viewing our sigmoid in 3D

Let's quickly take a look at a sigmoid in 3D, with two inputs:



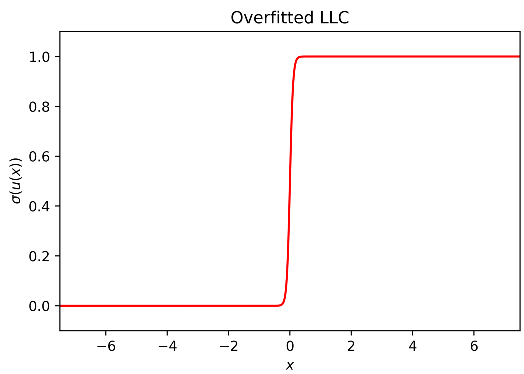

As you can see, you get mostly the same shape: if you look at it from the side, it's exactly the same, in fact! Just stretched out into 3D.

### 4.3.10 LLCs and overfitting

In chapter 2, we reduced **overfitting** by limiting the **magnitude** of $\theta$ using

$$R(\theta) = \lambda\|\theta\|^2 \tag{4.22}$$

In this chapter, it's more clear why reducing **magnitude** reduces **overfitting**. Let's see what happens when $\theta$ is very **large**:




Our shifted LLC: $u(x) = 20x$.

This function starts looking more and more like the **sign** function. This means we very, **very quickly** go from **confident** in one answer, to confident in another.

But if you have a limited dataset, and you're very carefully tuned to it, it's doesn't make sense to be **very confident** in your answer. Especially when you're **close** to your **boundary**.

You have closely **fitted** your sigmoid function to your data: in a way, you may have **overfitted** it.

The problem is, you're **rewarded** for increasing $\theta$! If you're just a little bit **more sure** of the answers you get correct, the loss function continues going **down**.

So, we need to prevent $\theta$ from **growing** to an unreasonably high value. We'll **penalize** a large $\|\theta\|$.

This means we're **penalizing** the machine's **overconfidence** in its answer, so that it **generalizes** better.

**Concept 30**

In **classification**, the **regularizer** follows the form

$$R(\theta) = \lambda \|\theta\|^2$$

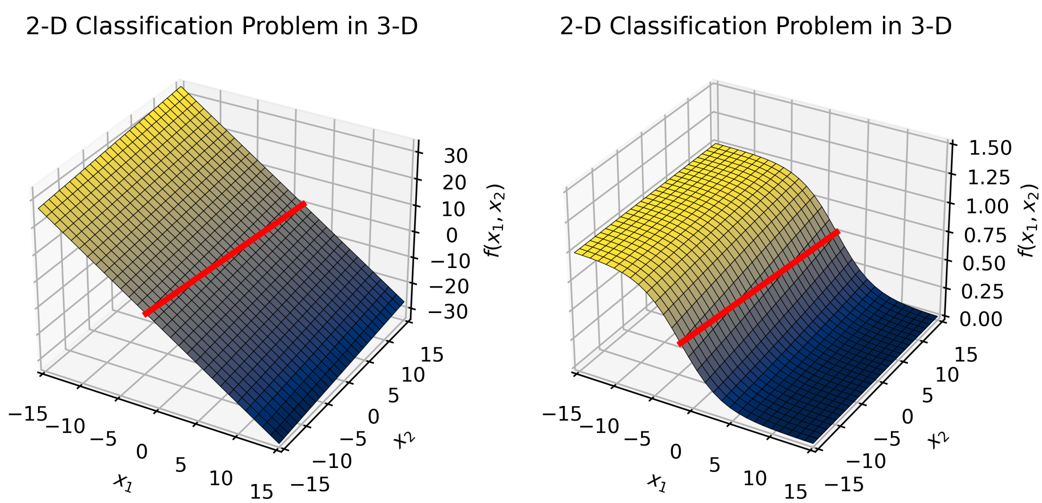
Regularization in this form reduces **overfitting** to our data by

- Making the **transition** between classifications less **sudden**, when it shouldn't be so **certain** of the boundary.
- It also prevents our model from becoming **overly confident** in its answer.

### 4.3.11 LLCs and LCs have the same boundary

One more important thing to note: noticed that we set $\sigma_{\text{thresh}} = .5$, because that was when $u(x) = 0$.

This means that, if our threshold is 0.5, then the boundary of our LLC should look exactly the same as if it were LC: the only difference is the values that we *can't* see:




Despite having different shapes in 3D, they both create 2-D **linear** classifiers: on the left, $u(x) = 0$, and on the right, $\sigma(u) = .5$.

One way to think about this difference is that while one may be logistic, they are both **linear**: they both create the same **linear separator**.

The main benefit of switching to LLC is that $\sigma(u)$ has a useful **gradient**, while $\text{sign}(u)$ does **not**, so we can do **gradient descent**.

The probabilistic interpretation is also more appropriate: we shouldn't be fully confident in our answers.

Even if we adjust our threshold $\sigma_{\text{thresh}}$, that will simply shift the linear classifier.

**Concept 31**

**LLCs** (Linear Logistic Classifiers) and **LCs** (Linear Classifiers) both create a **linear hyperplane separator** in $d-1$ dimensional **space**.

If the **threshold value** is 0.5, then they have the **exact same** separator.

### 4.3.12 Learning LLCs: Loss Functions

Now that we have fully **built up** LLCs, we can start trying to **train** our own.

In order to do that, we need a way to **evaluate** our hypotheses: a **loss function**.

Earlier in the chapter, we tried **0-1 Loss**:

$$\mathcal{L}_{01}\Big(h(x;\Theta), \quad y\Big) = \begin{cases} 0 & \text{if } y = h(x;\Theta) \\ 1 & \text{otherwise} \end{cases}$$

But, this **loss** function the same problem our **sign** function did: it isn't **smooth**!

It's a **discrete** function based on our **discrete classes**: so, it won't have a smooth **gradient** we can do **descent** on.

For our **sign** function, we switched to the **sigmoid** function, which measures in terms of **probabilities**: this gave us some **smoothness** to our classification.

Could we do the same here?

### 4.3.13 Building our new loss function

So, the **output** of our sigmoid $\sigma(u)$ is a **probability**: how **likely** do we think a point is to be in class $+1$?

We want a loss function

$$\mathcal{L}(g, y) \tag{4.23}$$

That considers two facts: the **correct** answer $y$, and how likely we **expected** $+1$ to be, $g = \sigma(u)$.

**Notation 32**

For our **loss function**, rather than using $y \in \{-1, +1\}$, we'll use $y \in \{0, 1\}$.

That way, $\sigma(u)$ and $y$ **match**:

$$y \in \{0, 1\} \qquad g \in (0, 1)$$

So, if the correct is 1, then we want $\sigma(u)$ to be **high**. If the correct answer is 0, we want $\sigma(u)$ to be **low**.

For **one** data point, then, we can consider, "how likely did we think the right answer was?"

$$G(g, y) = \begin{cases} g & \text{if } y = 1 \\ 1 - g & \text{else } (y = 0) \end{cases} \tag{4.24}$$

If we to choose 1 with probability $g$, this could also mean, "how likely were we to be **right**?"

This $G$ is how "**good**" our function is, so the **loss** would need for us to take the **negative**: we'll do that later.

### 4.3.14 Loss Function for Multiple Data Points

Now, how do we consider **multiple** data points? Well, let's think in terms of **probability**: guessing each point is a separate **event**.

We *could* add or **average** our guesses. But, since we're working with **probabilities**, there's a natural way to **combine** them: multiple events **occurring** at the same time.

Before, we asked, "how likely were we to be **right**?" for **one** data point. We could **extend** this question to, "how likely are we to get **every** question right?"

Well, each question we get right is an **independent** event $C_i$. If we want two independent events to **both** happen, we have to **multiply** their probabilities.

**Key Equation 33**

The probability of two independent events $A$ and $B$ happening at the same time is

$$\mathbf{P}\{A \text{ and } B\} = \mathbf{P}\{A\} * \mathbf{P}\{B\}$$

So if we want **all** of them, we just multiply:

$$\mathbf{P}\{E_{all}\} = \mathbf{P}\{E_1\} * \mathbf{P}\{E_2\} * \ldots * \mathbf{P}\{E_n\} \tag{4.25}$$

Written using pi notation, and also $g^{(i)}$ for multiple data points:

This notation is described in the prerequisites chapter! The short version: instead of adding terms with $\sum$, you multiply with $\prod$.

$$\mathbf{P}\{E_{all}\} = \prod_{i=1}^{n} \mathbf{P}\{E_i\} = \prod_{i=1}^{n} \begin{cases} g^{(i)} & \text{if } y^{(i)} = 1 \\ 1 - g^{(i)} & \text{if } y^{(i)} = 0 \end{cases} \tag{4.26}$$

### 4.3.15 Simplifying our expression - Piecewise

Our piecewise function is a bit **annoying**, though: is there a way to **simplify** it so that it doesn't have to be **piecewise**?

Our goal is to **combine** our two piecewise cases into a **single** equation. That means one of them needs to **cancel out** whenever the other is true.

Well, let's see what we have to **work** with.

Our **two** cases happen when $y = 0$ or $y = 1$: these are **nice** numbers! Why? Because of the **exponent** rules for these two:

- $c^0 = 1$: an exponent of 0 outputs 1: a factor of 1 in a product might as well **not be there**. It has been effectively **cancelled** out.
- $c^1 = c$: an **exponent** of 1 leaves the factor **unaffected**.

So, let's consider the **first** case, $g$. we can use $g^y$: if $y = 1$, it's **unaffected**. If $y = 0$, the term is **removed**.

We want the **opposite** for $1-g$. We can **swap** 1 and 0 by doing $1-y$. This gives us $(1-g)^{1-y}$.

For one data point:

$$\mathbf{P}\{E\} = \overbrace{g^y}^{y=1} \overbrace{(1-g)^{1-y}}^{y=0} \tag{4.27}$$

We've gotten rid of the piecewise function! Let's add back in the product:

$$\mathbf{P}\{E_{all}\} = \prod_{i=1}^{n} \mathbf{P}\{E_i\} = \prod_{i=1}^{n} g^{(i)y^{(i)}} \left(1 - g^{(i)}\right)^{1-y^{(i)}} \tag{4.28}$$

Looks pretty ugly, but we'll work on that.

### 4.3.16 Getting rid of the product

Our exponents look pretty **ugly**. Can we do something about that?

More importantly, **products** are also pretty unpleasant: we can't use **linearity**!

Linearity makes lots of problems easy to work with, so we try to keep it.

Linearity uses **addition** between variables. What sort of **function** could change a **product** into a **sum**?

Well, we could **list** out different basic functions, to see which ones connect sums and products. It turns out, one **interesting** function is

$$\overbrace{\log ab}^{\text{product}} = \overbrace{\log a + \log b}^{\text{sum}} \tag{4.29}$$

Aha! If we take the **log** of our function, we can turn a **product** into the **sum**!

The below equation looks complicated, but all we've done is swap the product for a sum!

$$\overbrace{\log\left(\prod_{i=1}^{n}\mathbf{P}\{E_i\}\right)}^{\text{product}} = \overbrace{\sum_{i=1}^{n}\log\left(\mathbf{P}\{E_i\}\right)}^{\text{sum}} \tag{4.30}$$

We can also separate our two **factors**:

$$\sum_{i=1}^{n}\left(\log\left(g^{(i)y^{(i)}}\right) + \log\left(\left(1-g^{(i)}\right)^{1-y^{(i)}}\right)\right) \tag{4.31}$$

And **finally**, we can remove the **exponents**:

$$\sum_{i=1}^{n}\left(y^{(i)}\log g^{(i)} + \left(1-y^{(i)}\right)\log\left(1-g^{(i)}\right)\right) \tag{4.32}$$

**Concept 34**

Our **negative log likelihood** (NLL) comes from a couple steps:

- **Use** $y \in \{0,1\}$ instead of $y \in \{-1,+1\}$ so that $y$ and $g$ have **matching** outcomes.
- Get the **chance** the model is right on every **guess**: a **product**.
- Use **exponents** to convert the **piecewise** expression into a single **equation**.
- Take the **log** of our expression to switch from a **product** to a **sum**.
- Take the **negative** to get the **loss** rather than the "**goodness**" of our function.

### 4.3.17 Negative Log Likelihood

Remember, at the **beginning**, we said that we need to take the **negative**: our function represents how **good** our function is, but we want the **loss**.

With this, our function is in its final form:

**Key Equation 35**

We can get the loss of our **linear logistic classifier (LLC)** using the **negative log likelihood (NLL)** loss function

$$\mathcal{L}_{nll}(g^{(i)},\quad y^{(i)}) = -\left(y^{(i)}\log g^{(i)} + \left(1-y^{(i)}\right)\log\left(1-g^{(i)}\right)\right)$$

Or,

$$-\Big((\text{answer})\log(\text{guess}) + (1-\text{answer})\log\left(1-\text{guess}\right)\Big)$$

Our total loss is

$$\sum_{i=1}^{n} \mathcal{L}_{nll}(g^{(i)}, \quad y^{(i)}) \tag{4.33}$$

Finally, we add our **regularizer**:

$$J_{lr}(\theta, \theta_0; \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} \left( \mathcal{L}_{nll}(g^{(i)}, \qquad y^{(i)}) \right) + \lambda \|\theta\|^2 \tag{4.34}$$

**Key Equation 36**

The full **objective function** for **LLC** is given as

$$J_{lr}(\theta, \theta_0; \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} \left( \mathcal{L}_{nll} \left( \sigma(\theta^T x + \theta_0), \quad y^{(i)} \right) \right) + \lambda \|\theta\|^2$$

Using our **loss** function $\mathcal{L}_{nll}$, and our **logistic** function $\sigma(u)$.

## 4.4 Gradient Descent for Logistic Regression

### 4.4.1 Summary

Now, we have developed all the tool we need to do binary classification with LLC:

- A **linear** model that lets us **combine** our variables,

$$u(x) = \theta^T x + \theta_0 \tag{4.35}$$

- A **logistic** model that lets us get the **probability** of a classification,

$$\sigma(u) = \frac{1}{1 + e^{-u}} \tag{4.36}$$

- A **threshold value** we use to determine how to **classify** our data,

$$h(x; \theta) = \begin{cases} +1 & \text{if } \sigma(u(x)) > \sigma_{\text{thresh}} \\ 0 & \text{otherwise} \end{cases} \tag{4.37}$$

- A **loss function** NLL we use to **evaluate** our model performance:

$$\mathcal{L}_{nll}(g^{(i)}, \quad y^{(i)}) = -\left(y^{(i)} \log g^{(i)} + \left(1 - y^{(i)}\right) \log \left(1 - g^{(i)}\right)\right)$$

- And an **objective function** we can **optimize**:

$$J_{lr}(\theta, \theta_0; \mathcal{D}) = \lambda \|\theta\|^2 + \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_{nll}(g^{(i)}, \quad y^{(i)}) \tag{4.38}$$

We have everything we need to do optimization.

### 4.4.2 The problem: Gradient Descent

We want to do **gradient descent** to minimize $J_{lr}$

$$R(\theta) + J_{lr}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_{nll}(g^{(i)}, \quad y^{(i)}) \tag{4.39}$$

We want repeatedly **adjust** our model $\Theta = (\theta, \theta_0)$ to improve $J_{lr}$. To do that, we want the gradients for $\theta$ and $\theta_0$. Let's start with $\theta$.

$$\nabla_\theta J_{lr} = \frac{\partial J_{lr}}{\partial \theta} \tag{4.40}$$

First, $J_{lr}$ has **two** terms, so we'll separate them.

$$\nabla_\theta J_{lr} = \frac{\partial R}{\partial \theta} + \frac{1}{n}\sum_{i=1}^{n} \frac{\partial \mathcal{L}_{NLL}}{\partial \theta}(g^{(i)}, \quad y^{(i)}) \tag{4.41}$$

The regularization term is pretty easy, because we did it last chapter:

$$\frac{\partial R}{\partial \theta} = 2\lambda\theta \tag{4.42}$$

But what about our first term?

### 4.4.3 Getting the gradient: Chain Rule

Now, we just need to do

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta}(g, y) \tag{4.43}$$

With our $\mathcal{L}_{NLL}$ term, we run into an issue: how do we take the **derivative**? The function is very, very deeply **nested**. In our case:

$x$ **affects** $u(x)$. $u(x)$ **affects** $\sigma(u)$. $\sigma(u) = g$ **affects** $\mathcal{L}_{NLL}(g, y)$, which finally **affects** $J(\theta, \theta_0)$.

How do we represent this **chain** of functions? With the **chain rule**:

$$\frac{\partial A}{\partial C} = \frac{\partial A}{\partial B} \cdot \frac{\partial B}{\partial C} \tag{4.44}$$

So, we'll build up a **chain rule** for our needs. We'll use $g = \sigma(u)$.

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = \frac{\partial \mathcal{L}_{NLL}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \theta} \tag{4.45}$$

Sigma contains $u$, so we'll use that instead:

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = \frac{\partial \mathcal{L}_{NLL}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial u} \cdot \frac{\partial u}{\partial \theta} \tag{4.46}$$

This is our full **chain rule**!

**Key Equation 37**

The **gradient** of **NLL** can be calculated using the **chain rule**:

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = \frac{\partial \mathcal{L}_{NLL}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial u} \cdot \frac{\partial u}{\partial \theta} \tag{4.47}$$

### 4.4.4 Getting our individual derivatives

We can take the derivative of each of these objects. First, let's look at $\mathcal{L}_{NLL}$

$$\mathcal{L}_{nll}(\sigma, y) = -\Big(y \log \sigma + (1-y)\log(1-\sigma)\Big)$$

And we'll use $\frac{d}{dx}\log(x) = \frac{1}{x}$

$$\boxed{\frac{\partial \mathcal{L}_{NLL}}{\partial \sigma} = -\left(\frac{y}{\sigma} - \frac{1-y}{1-\sigma}\right)} \tag{4.48}$$

Now, we look at $\sigma(u)$:

$$\sigma(u) = \frac{1}{1+e^{-u}} \tag{4.49}$$

If we take the derivative, we can get:

$$\frac{\partial \sigma}{\partial u} = \frac{-e^{-u}}{\left(1+e^{-u}\right)^2} \tag{4.50}$$

Which we can rewrite, conveniently, as

Try this yourself if you're curious!

$$\boxed{\frac{\partial \sigma}{\partial u} = \sigma(1-\sigma)} \tag{4.51}$$

Finally, our last derivative:

$$u = \theta^T x + \theta_0 \tag{4.52}$$

$$\boxed{\frac{\partial u}{\partial \theta} = x} \tag{4.53}$$

### 4.4.5 Simplifying our chain rule

So, now, we can put together our chain rule:

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = \frac{\partial \mathcal{L}_{NLL}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial u} \cdot \frac{\partial u}{\partial \theta} \tag{4.54}$$

Plug in the derivatives:

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = -\left(\frac{y}{\sigma} - \frac{1-y}{1-\sigma}\right) \cdot \sigma(1-\sigma) \cdot x \tag{4.55}$$

Simplify:

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = \Big((1-y)\sigma - y(1-\sigma)\Big) \cdot x \tag{4.56}$$

And finally, we sum the terms. We can do the $\theta_0$ gradient at the same time: the only difference is that $\frac{\partial u}{\partial \theta_0} = 1$, instead of x.

**Key Equation 38**

The **gradients** of NLL for gradient descent are

$$\nabla_\theta \mathcal{L}_{NLL} = (\sigma - y)x$$

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta_0} = (\sigma - y)$$

We can plug this into $J_{lr}$:

One comment we didn't make: remember that $R(\theta)$ won't show up in the $\theta_0$ derivative!

$$\nabla_\theta J_{lr} = \frac{1}{n}\sum_{i=1}^{n}\left(\left(g^{(i)} - y^{(i)}\right)x^{(i)}\right) + 2\lambda\theta \tag{4.57}$$

$$\frac{\partial J_{lr}}{\partial \theta_0} = \frac{1}{n}\sum_{i=1}^{n}\left(g^{(i)} - y^{(i)}\right) \tag{4.58}$$

We can use this to do **gradient descent**!

$$\theta_{new} = \theta_{old} - \eta\nabla_\theta J_{lr}(\theta_{old}) \tag{4.59}$$

In $\theta^{(t)}$ notation:

$$\theta^{(t)} = \theta^{(t-1)} - \eta\left(\nabla_\theta J_{lr}(\theta^{(t-1)})\right) \tag{4.60}$$

$$\theta_0^{(t)} = \theta_0^{(t-1)} - \eta\left(\frac{\partial J_{lr}(\theta^{(t-1)})}{\partial \theta_0}\right) \tag{4.61}$$

This also corresponds to some basic math within Neural Networks, which we will return to **later** in the course.

## 4.5 Handling Multiple Classes

Now, we have developed a **binary** classifier, using logistic regression. But, many (almost all) problems have more than two classes!

**Example:** Different animals, genres of movies, sub-types of disease, etc.

### 4.5.1 Approaches to multi-class classification

So, we need to a way to do **multi-classing**. Consider two main approaches:

- **Train** many binary classifiers on different **classes** and **combine** them into a single model.
  - There are several ways to **combine** these **classifiers**. We won't go over them here, but some **names**: OVO (one-versus-one), OVA (one-versus-all).
- **M**ake **one** classifier that handles the multi-class problem by itself.
  - This model will be A **modified** version of logistic regression, using a variant of NLL.

The **latter** approach is what we will use in this **next** section.

### 4.5.2 Extending our Approach: One-Hot Encoding

Rather than being **restricted** to classes 0 and 1, we'll have $k$ **distinct** classes. Our **hypothesis** will be

$$h : \mathbb{R}^d \to \{C_1, C_2, C_3, \dots C_k\}$$

Where $C_i$ is the $i^{\text{th}}$ class. Meaning, we want to **output** one of those $k$ **classes**.

Because we'll be using our computer to do **math** to get the **answer**, we need to represent this with **numbers**. Before, we would simply **label** with 0 or 1.

We could return $\{1, 2, 3, 4, 5...k\}$ for each **label**. But this is **not** a good idea: it implies that there's a natural **order** to the classes, which isn't necessarily true.

If we don't **actually** think $C_1$ is closer to $C_2$ than to $C_5$, we probably shouldn't represent them with numbers that are **closer** to each other.

Instead, each class needs to be a **separate** variable. We can store them in a vector:

$$\begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_k \end{bmatrix} \tag{4.62}$$

So, our **label** will be

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix} \tag{4.63}$$

In binary classification, we used 0 or 1 to indicate whether we fit into one **class**. So, that's how we'll do each class: 0 if our data point is **not** in this class, 1 if it **is**.

This approach is called **one-hot encoding**.

**Definition 39**

**One-hot encoding** is a way to represent **discrete** information about a data point.

Our $k$ classes are stored in a length-$k$ column **vector**. For **each** variable in the vector,

- The value is **0** if our data point is **not in that class**.
- The value is **1** if our data point is **in that class**.

In one-hot encoding, items are **never** labelled as being in **two** classes at the **same time**.

**Example:** Suppose that we want to classify **furniture** as table, bed, couch, or chair.

$$\begin{bmatrix} \text{table} \\ \text{bed} \\ \text{couch} \\ \text{chair} \end{bmatrix} \tag{4.64}$$

For each class:

$$y_{\text{chair}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad y_{\text{table}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad y_{\text{couch}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad y_{\text{bed}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{4.65}$$

### 4.5.3 Probabilities in multi-class

So, we now know our **problem**: we're taking in a data point $x \in \mathbb{R}^d$, and **outputting** one of the classes as a **one-hot vector**.

So, now that we know what sorts of data we're **expecting**, we need to decide on the formats of our **answer**.

We'll be returning a vector of length-$k$: **one** for each **class**. When we were doing **binary** classification, we estimated the **probability** of the positive class.

So, it should make sense to do the same **here**: for each class, we'll return the estimated **probability** of our data point being in that class.

$$g = \begin{bmatrix} \mathbf{P}\{x \text{ in } C_1\} \\ \mathbf{P}\{x \text{ in } C_2\} \\ \vdots \\ \mathbf{P}\{x \text{ in } C_k\} \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_k \end{bmatrix} \tag{4.66}$$

We can add an **additional** rule: the probabilities need to add up to **one**: we should assume our point ends up in some class or **another**.

$$g_1 + g_2 + ... + g_k = 1 \tag{4.67}$$

**Concept 40**

The different terms of our **multi-class** guess $g_i$ represent the **probability** of our data point being in class $C_i$.

Because we should assume our data point is in **some** class, all of these probabilities have to **add** to 1.

**Scaling** values to add up to 1 is called **normalization**. How do we do that?

Well, let's say each class gets a **value** of $r_i$, before being **normalized**: we have done some other math we won't worry about.

To make the total 1, we'll **scale** our terms by a factor C:

$$C(r_1 + r_2 + ...r_k) = C\left(\sum_{i=1}^{k} r_i\right) = 1 \tag{4.68}$$

We can get our factor C just by dividing:

$$C = \frac{1}{\sum r_i} \tag{4.69}$$

We've got $g_i$ now!

$$g = \begin{bmatrix} r_1 / \sum r_i \\ r_2 / \sum r_i \\ \vdots \\ r_k / \sum r_i \end{bmatrix} \tag{4.70}$$

### 4.5.4 Turning sigmoid multi-class

Now, we just need to compute $r_i$ terms to plug in. To do that, we'll see how we did it using sigmoid:

$$g = \sigma(u) = \frac{1}{1+e^{-u}} \tag{4.71}$$

This function is 0 to 1, which is good for being a probability.

Just for our convenience, we'll switch to positive exponents: all we have to do is multiply by $e^u/e^u$.

Negative numbers are easy to mess up in algebra.

$$g = \frac{e^u}{1+e^u} \tag{4.72}$$

We'll think of **binary** classification as a special case of **multi-class** classification. The above probability could be thought of as $g_1$: the chance of our first class.

**Concept 41**

**Binary classification** is a **special** case of **multi-class** classification with only **two** classes.

So, we can use it to figure out the **general** case.

So, what was our **second** probability, $1-g$? This will be our second class, $g_2$.

$$g_2 = 1 - g = \frac{1}{1+e^u} \tag{4.73}$$

This follows an $r_i/(\sum r_i)$ format: the numerators (1 and $e^u$) add to **equal** the denominator $(1+e^u)$.

$$g = \begin{bmatrix} 1/(1+e^u) \\ e^u/(1+e^u) \end{bmatrix} \tag{4.74}$$

How do we **extend** this to **more** classes? Well, 1 and $e^u$ are **different** functions: this a problem. We want to be able to **generalize** to many $r_i$.

How do they make them **equivalent**? We could say $1 = e^0$. So, we could treat both terms as **exponentials**!

$$g_1 = \frac{e^u}{e^0+e^u} \tag{4.75}$$

We can do this for an **arbitrary** number of terms. We'll treat them as **exponentials**, just like for $e^u$ and $e^0$

$$g_i = \frac{r_i}{\sum r_j} = \frac{e^{u_i}}{\sum e^{u_j}} \tag{4.76}$$

### 4.5.5 Our Linear Classifiers

What are each of those $u_i$ terms? When we were doing **binary**, it was a **linear regression** function:

$$u(x) = \theta^T x + \theta_0 \tag{4.77}$$

We can do this for multiple different $u_i$ by just creating a different linear classifier $(\theta, \theta_0)$ for each one. We'll represent each vector as $\theta_{(i)}$.

$$\theta_{(1)} = \begin{bmatrix} \theta_{1(1)} \\ \theta_{2(1)} \\ \vdots \\ \theta_{d(1)} \end{bmatrix} \qquad \theta_{(2)} = \begin{bmatrix} \theta_{1(2)} \\ \theta_{2(2)} \\ \vdots \\ \theta_{d(2)} \end{bmatrix} \qquad \theta_{(k)} = \begin{bmatrix} \theta_{1(k)} \\ \theta_{2(k)} \\ \vdots \\ \theta_{d(k)} \end{bmatrix} \tag{4.78}$$

And each can be used on our input:

$$u_1(x) = \theta_{(1)}^T x + \theta_{0(1)} \qquad u_2(x) = \theta_{(2)}^T x + \theta_{0(2)} \dots \tag{4.79}$$

But this is tedious. Instead, we can combine them all into a $(d \times k)$ **matrix**.

k classes, so we need k classifiers. We'll stack them side-by-side like how we stacked multiple data points to create X.

$$\theta = \begin{bmatrix} \theta_{(1)} & \theta_{(2)} & \cdots & \theta_{(k)} \end{bmatrix} = \begin{bmatrix} \theta_{1(1)} & \theta_{1(2)} & \cdots & \theta_{1(k)} \\ \theta_{2(1)} & \theta_{2(2)} & \cdots & \theta_{2(k)} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{d(1)} & \theta_{d(2)} & \cdots & \theta_{d(k)} \end{bmatrix} \tag{4.80}$$

And our constants, $\theta_0$, in a $(k \times 1)$ matrix:

$$\theta_0 = \begin{bmatrix} \theta_{0(1)} \\ \theta_{0(2)} \\ \vdots \\ \theta_{0(k)} \end{bmatrix} \tag{4.81}$$

**Concept 42**

We can combine **multiple classifiers** $\Theta_{(i)} = \left(\theta_{(i)}, \theta_{0(i)}\right)$ into large **matrices** $\theta$ and $\theta_0$ to compute **multiple** outputs $u_i$ at the **same** time.

This will put all of our terms into a $(1 \times k)$ vector $u$.

$$u(x) = \theta^T x + \theta_0 = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_k \end{bmatrix} \tag{4.82}$$

### 4.5.6 Softmax

We now have all the pieces we need. Our **linear regression** for each class:

$$\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_k \end{bmatrix} = \theta^T x + \theta_0 \tag{4.83}$$

The **exponential** terms, to get **logistic** behavior:

$$r_i = e^{u_i} \tag{4.84}$$

The **averaging** to get probability = 1:

$$g = \begin{bmatrix} r_1 / \sum r_i \\ r_2 / \sum r_i \\ \vdots \\ r_k / \sum r_i \end{bmatrix} \tag{4.85}$$

And so, our multiclass function is...

**Definition 43**

The **softmax function** allows us to calculate the probability of a point being in each class:

$$g = \begin{bmatrix} e^{u_1} / \sum e^{u_i} \\ e^{u_2} / \sum e^{u_i} \\ \vdots \\ e^{u_k} / \sum e^{u_i} \end{bmatrix}$$

Where

$$u_i(x) = \theta_{(i)}^T x + \theta_{0(i)} \tag{4.86}$$

If we are forced to make a **choice**, we choose the class with the **highest probability**: we return a **one-hot encoding**.

### 4.5.7 NLLM

One loose end left to tie up: our **loss function**. We need to evaluate our hypothesis, and be able to improve it.

For **binary classification**, we did **NLL**:

$$\mathcal{L}_{nll}(g, y) = -\Big(y \log g + (1-y) \log (1-g)\Big)$$

How do we make this work in **general**? Well, we want to make our two terms have a **similar** form, so we can generalize to more classes.

- $g$ and $1-g$ are both probabilities: we can think of them as $g_1$ and $g_2$, respectively.
- If $g = g_1$, then we would expect $y = y_1$. And indeed: it gives a 1 if we're in the first class (+1).
  - Similarly, $1 - y = y_2$.

$$\mathcal{L}_{nll}(g, y) = -\Big(y_1 \log g_1 + (y_2) \log (g_2)\Big)$$

They have the **same** format now! Much tidier. And it tracks: when one **label** is correct, the other term is $y_j = 0$, and **vanishes**.

Does this **generalize** well? It turns out it does: with **one-hot encoding**, the correct label is **always** $y_j = 1$, and the incorrect labels are **all** $y_j = 0$.

So, we'll write it out:

**Key Equation 44**

The **loss** function for **multi-class** classification, **Negative Log Likelihood Multiclass (NLLM)**, is written as:

$$\mathcal{L}_{NLLM}(g, y) = -\sum_{j=1}^{k} y_j \log(g_j)$$

Because of **one-hot encoding**, all terms except one have $y_j = 0$, and thus **vanish**.

Using all of these functions, we can finally do gradient descent on our multi-class classifier. However, we won't go through that work in these notes.

## 4.6 Prediction Accuracy and Validation (WIP)

We've been working in **probabilities**, but in the end, the goal is usually to make a **decision** or **prediction**: which class do we **pick**?

In general, we just pick the class we predict with the **highest** probability.

And in the real world, we don't care about how close we were to right - we just care about how often we **were** right.

So, we use **accuracy**.

**Definition 45**

The **accuracy** of our model is the **percentage** of the time we get the **right** answer.

We can write this as

$$A(h; D) = 1 - \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(h(x^{()}), y^{(i)}\right) \tag{4.87}$$

Where $\mathcal{L}$ is 0-1 loss (**counting** the number of **wrong** answers)

Or, "one minus how often we get the answer **wrong**".

## 4.7 Terms

- Class
- Classification
- Label
- Binary Classification
- 0-1 Loss
- Linear Classifier
- Separator
- Orientation
- Boundary
- Normal Vector
- Dot Product (Conceptual)
- Linear Separator
- Sign Function
- Hyperplane
- Separability
- Non-separate data
- Sigmoid Function
- Logistic Regression
- Prediction Threshold
- Linear Logistic Classifier (LLC)
- Negative Log Likelihood (NLL)
- Multi-class Classification
- One-Hot Encoding
- Normalization
- Softmax Function
- Negative Log Likelihood Multi-Class (NLLM)
- Accuracy