

Explanatory Notes for 6.390

Shauntclair Ruiz (Current TA)

Fall 2022

Contents

5	Feature Representation	2
5.1	Gaining intuition about feature transformations	7
5.2	Systematic feature construction	11
5.3	Hand-constructing features for real domains	21
5.4	Terms	35

CHAPTER 5

Feature Representation

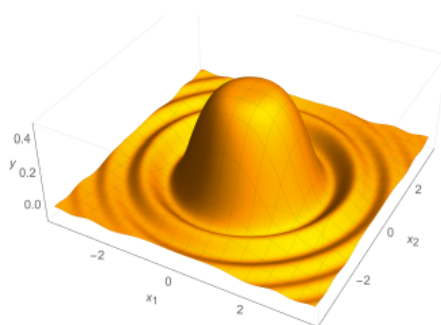
What's still missing?

Last chapter, we used our linear regression model to do classification: we created a "hyper-plane" to **separate** the the data that we placed in each class.

We also mentioned that regularization can increase **structural error**, by limiting what possible θ models we're allowed to use.

But, what if our linear model is already **too limited**? What if we need a more complicated model? This is true in a lot of real-world problems, like vibration:

Our goal was to decrease estimation error, but that's beside the point right now.



This wave doesn't seem particular friendly to a planar approximation.

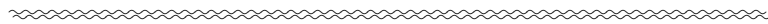
These kinds of situations are called, appropriately, **non-linear**.

Concept 1

Non-linear behavior cannot be accurately represented by any **linear** model.

In order to create an accurate model, we have to use some **nonlinear** operation.

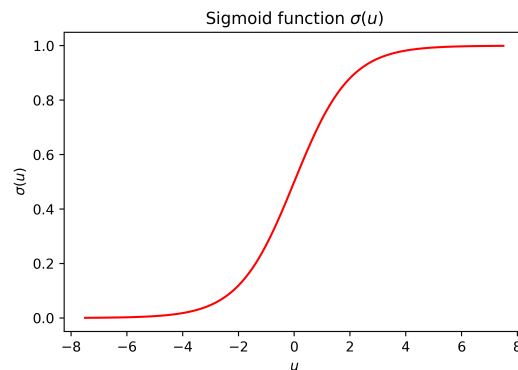
If we could create effective, non-linear models, we might even be able to deal with data that was previously "**linearly inseparable**".



Possible Solutions: Polynomials

Let's try to think of ways to approach this problem. We'll start with a 1-D input, for simplicity.

Upon hearing "non-linear", we might remember the function we introduced last chapter: the **sigmoid**.



Your friendly neighborhood sigmoid.

Can we use this to create a new model class? For now, unfortunately not: remember that we used this in the last chapter, and we still got a **linear** separator. The reasons were discussed there.

Instead, we can get inspiration from our example of "structural error". For now, let's focus on **regression** (though classification isn't too different):

We'll show ways we can use this kind of approach, when we discuss Neural Networks.

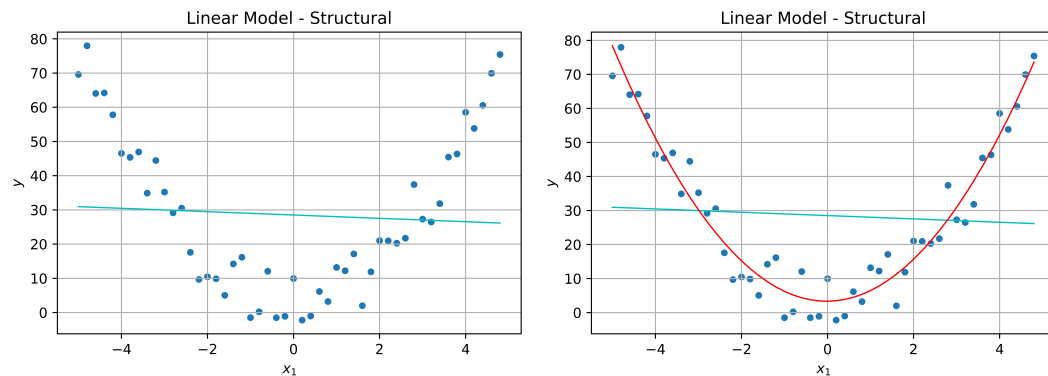


Figure 5.1: A linear function can't represent this dataset. However, a parabola can!

We're still using our input variable x , but this time, we've "**transformed**" it: we have squared x , giving us a model of the form _____

Remember that x is 1-D right now!

$$h(x) = Ax^2 + Bx + C \quad (5.1)$$

It should be clear that this model is more **expressive** than the one before: it can create every model that our linear approach could (just by setting $A = 0$), and it can create new models in a parabola shape. _____

Concept 2

We can make our **linear** model more **expressive** by add a squared term, and turning it into a **parabolic** function.

This concept can be extended even further, to any **polynomial**.

Reminder: "expressiveness" or "richness" of a hypothesis class is how many models it can represent: a more expressive model can handle more different situations.



Transformation

How do we *generalize* this concept? Well, we have a set of constant parameters A, B, C . These are similar to our constants θ_i . Let's change our notation:

$$h(x) = \theta_2 x^2 + \theta_1 x + \theta_0 \quad (5.2)$$

Now, we've got something more familiar. We could imagine extending this to any number of terms $\theta_i x^i$: if we needed a cubic function, for example, we could include $\theta_3 x^3$.

This is starting to look pretty similar to our previous model: in fact, we could even separate out θ as a variable: _____

Notice that θ_0 corresponds to $x^0 = 1$.

$$h(x) = \overbrace{\sum_{i=1}^k \theta_i x^i}^{\text{Polynomial sum}} = \overbrace{\begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_k \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix}}^{\text{Store as vectors}} = \overbrace{\theta^T \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix}}^{\text{Simplify}} \quad (5.3)$$

This really *is* starting to look like our linear transformation! That's helpful: we might be able to use the techniques we developed before.

In fact, we can argue that they're **equivalent**: we've just changed what our input vector is. Consider our new input $\phi(x)$:

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix} \quad h(x) = \theta^T \overbrace{\phi(x)}^{\text{New input}} \quad (5.4)$$

This is called **transforming** our input.

Definition 3

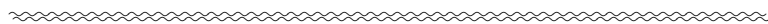
A **transformation** $\phi(x)$ takes our input vector x and converts it into a **new** vector.

This transformation can be used to:

- Allow our model to handle new, more **complex** situations (more **expressiveness**)
- **Pre-process** our data to make it **easier** for our model to find **patterns**.
- Convert our data into a **usable** format (if, say, the original format doesn't fit into our equations)

Example: Taking our input x and converting it into a polynomial is a **transformation** of our input.

This chapter will focus on these kinds of transformations.



Features

One benefit of only changing out input is that we can continue to use our linear representation: we will be able to optimize a "linear" model θ , over data that has been made **nonlinear**.

These transformations can be complex, especially for multi-dimensional inputs. In this

first case, we only combined one input with **itself**. But, often, we can combine multiple together!

Thus, we should be careful to distinguish each input variable from each other. We often call these "**features**". However, we need to be careful:

Clarification 4

We often use the word **feature** in related (but not identical) contexts:

- A **feature** can be one **aspect** of our **original data**: for example, whether or not something is a cat or a dog, or the height of a patient.
- A **feature** can also be one mathematical **variable** in our **transformed input**. x_i is a feature of the data, while each variable of $\phi(x)$ is a feature of the transformed data.

Just like how we have an input space, we call the collection of possible values for our features the **feature space**.

Example: x in our previous example was a feature of the data, while x^i is a feature of our transformed vector.

Combined, this is why we called this technique the **feature transformation**: we apply some *transform* to the *features* of a data, to create a new set of *features*.

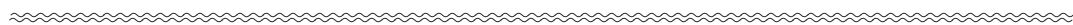
Since these transforms only apply to our features, we can keep the structure of a linear function:

Definition 5

Feature transformation allows us to do **linear** regression or classification on a set of **features** we have **non-linearly transformed**:

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

$\phi(x)$ is our (often nonlinear) transformation of our features x .

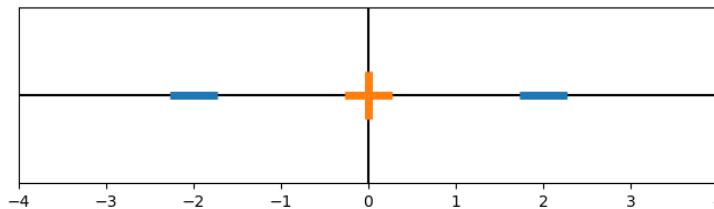


5.1 Gaining intuition about feature transformations

Now that we understand the general idea of feature transformations, we can begin work with them, particularly for classification.

Our goal is often to take data that linear models couldn't handle, and make it **separable**.

So, we'll consider maybe the simplest (solvable) case of a nonlinear data set:



In its current state, there's no one plane that would go through these data points: this is where our transform comes in. We'll try using a polynomial transform with x^2 . It turns out, $-x^2 + 2$ works pretty well.

How do we visualize this? It turns out, there are different perspectives:

Clarification 6

There are **two** different ways we can **graph** a transformation:

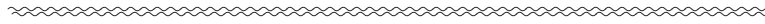
- We transform the **separator**: if our model is $-x^2 + 2$, we just graph that function over the data.
 - This is the approach we used above: we wanted a line that **fit** to our data.
- We transform the **data**: we graph each data point according to our model $-x^2 + 2$.
 - This model allows us to keep a "**linear** separator": we effectively "shift" the data to be linear.

These models are mathematically **equivalent**, and we'll switch the approach we're using based on which is easier/more useful to graph.

It may seem concerning to transform the data, rather than the model. However, keep in mind that:

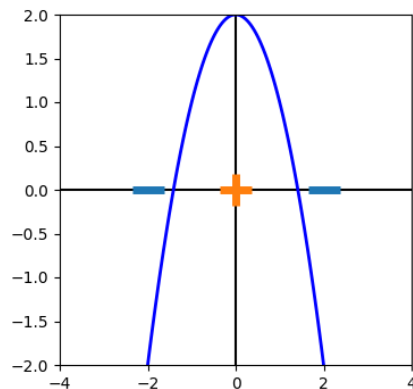
- If you switch perspectives, they're the same: a different model will just "transform" the data when calculating its output.
- Usually, we try to preserve the original structure of the data, so we don't lose information: we just add more.
 - For example, $[1, x, x^2]$ still contains the information x : we just add x^2 .

Example: Let's show both of these in action.



5.1.1 Transforming our separator

First, we transform our linear separator as desired: graphing $-x^2 + 2 = 0$ on our plot.

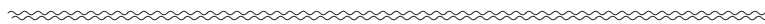


In this version, we've taken our hyperplane separator and transformed it nonlinearly.

In this case, we have assigned $-x^2 + 2 < 0$ as positive.

In this version, we preserve the structure of the data, making it easier to see the original shape. However, it's not as easy to think about the shape and orientation of the "plane" now that it's been deformed.

For example, we don't really have a good "normal" vector, even if we know which side is positive. This is why, to keep our model "linear", we can transform the data, and find the corresponding plane. We'll do that next.

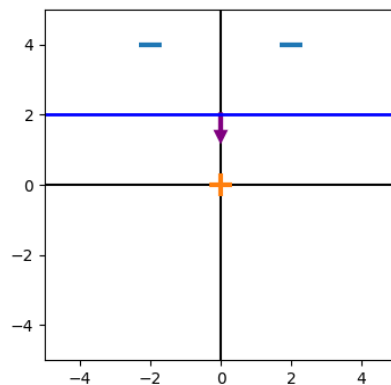


5.1.2 Transforming our data

In this case, every data point gets plotted on $[x, x^2]$. Our hyperplane is given by

$$-x^2 + 2 = \overbrace{\begin{bmatrix} 0 \\ -1 \end{bmatrix}}^{\theta^T} \begin{bmatrix} x \\ x^2 \end{bmatrix} + 2 \quad (5.5)$$

Thus, we get a θ pointing downward, with an offset of 2.



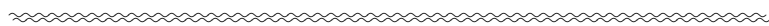
This time, we've transformed our data: the math is totally the same, but now we can identify our separator more easily.

Note that our transformation makes the data linearly separable!

Concept 7

Features transformations allow us to **non-linearly** transform our data, in order to make that data **linearly separable**, or at least, more **accurate** with a linear separator.

Often, we do this by transforming into a **higher dimensional** space.



5.1.3 Positive vs. Negative

While these perspectives are helpful, they can become too complicated with more dimensions/higher-dimensional transformations.

In an effort to simplify, we might ask ourselves, "what do we really want to know"? In the end, all we typically care about is classification: which data points are positive or negative?

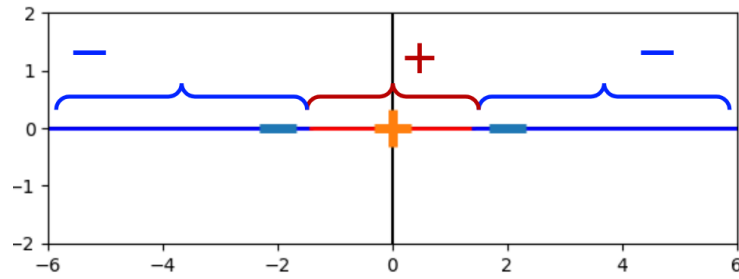
So, we'll create a third representation to correspond to that.

Concept 8

A third, **simplified** representation of our transformation doesn't show how it affects our data points or classifier. Instead, we just show the **result**: which regions are classified as positive, and which are classified as negative?

This allows us to see which points get **classified** in which way, without considering the high-dimensional details of the model itself.

Example: We can graph this for our sample data:



This way, we can stay in a 1-D space, while showing the information we need!

Note that the points where we switch between positive and negative, $\pm\sqrt{2}$, are the points corresponding to $-x^2 + 2 = 0$: they're the only part of the separator surface visible in our 1D plot.

They match our nonlinear hyperplane separator from section 5.1.1

5.2 Systematic feature construction

Now that we've established feature transformations, let's consider a couple options for how we'd want to do it, and how we can generalize to higher dimensions.

Here, we'll present two common ways to construct features, in a way that's consistent across problems, or "systematic".

We could also call this "problem independent": it works regardless of what kind of problem you have. Though, that doesn't mean problem time won't affect performance.

5.2.1 Polynomial Basis

At the start of this chapter, we introduced the idea of polynomial transformations. If a linear function isn't "expressive" enough to solve a problem, then we can create a more complex model, based on how many x^i we include. This can be written as:

$$h(x) = \sum_{i=1}^k \theta_i x^i = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_k \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix} \quad (5.6)$$

Another word for this might be a "polynomial **basis**".

Definition 9

A "**basis**" is the set of basic, **distinct** elements where every **linear** combination creates a unique item in the **space**.

In this case, each term x^i is part of the "basis": we can use any combination

$$\sum_i \theta_i x^i$$

to create a unique polynomial. This allows us to cover our "feature space".

~~~~~

#### 5.2.1.1 Order

An important question to ask is, "how many terms do we include"?

We categorize our polynomials based on the highest exponent included: this is called the **order**.

#### Definition 10

**Order**  $k$ , also known as **degree**, is the **largest** exponent allowed in our **polynomial**.

Every higher exponential  $x^j$  can be thought of as having a coefficient  $\theta_k = 0$ : as far as we're concerned, it **doesn't exist**.

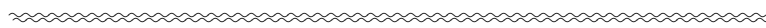
**Example:** We can compare different orders, by looking at the feature vector they create:

Here's a table of the first few:

| Order    | $d = 1$              | Example                     |
|----------|----------------------|-----------------------------|
| 0        | $[1]$                | 3.5                         |
| 1        | $[1, x]^T$           | $2.5x - 1$                  |
| 2        | $[1, x, x^2]^T$      | $4.1x^2 - 10x + 1$          |
| 3        | $[1, x, x^2, x^3]^T$ | $x^3 + 8x^2 + x - \sqrt{2}$ |
| $\vdots$ | $\vdots$             | $\vdots$                    |

Note that, while we chose every coefficient to be nonzero here, they don't have to be!  $-x^2 + 2$  from before is a valid second-order polynomial.

The order we choose is an important decision choice.



### 5.2.1.2 Overfitting with order

It's difficult to know how many terms to include in our polynomial, but we run into two problems if our order is **too high**:

- It becomes time-consuming to calculate, with little benefit
- We start overfitting more and more.

The first part makes sense: with more terms, we have to do more multiplications, more additions, etc.

#### Concept 11

More **complex models** tend to be more **expensive** to train, and slower to use. This is a trade-off for more **accuracy**.

Usually, there's a point where cost **outweighs** benefits. A problem is rarely perfectly solved, even by an excellent model, so you can't just continue until it's "perfect".

But what about the second part? Why do we increase overfitting?

With a higher order, our polynomial becomes more complex: it can take on more shapes, which are increasingly complex and perfectly fit to the data.

This can cause our data to overlook obvious patterns, and instead create a very precise shape that is paying attention to the noise in our model.

**Concept 12**

**High-order polynomials** are very vulnerable to **overfitting**.

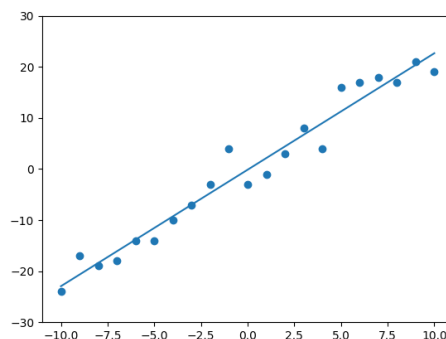
Because they can take on so many different, **complex** functions, they can very very closely **match** the original data set.

This can cause the model to "learn" noise, and **miss** broader and simpler patterns that actually exist. It may fail to learn something broad and useful, while **memorizing** the dataset with its expressiveness.

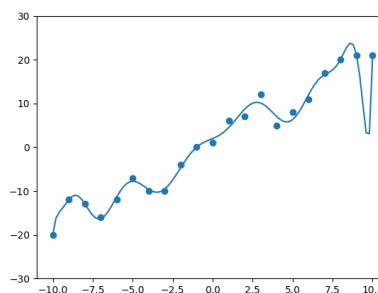
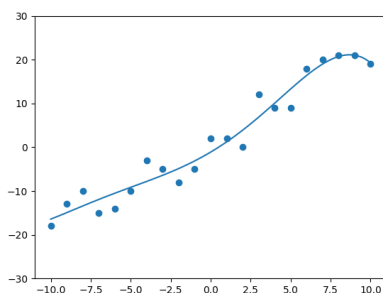
Let's see this in action: we'll generate some data based on  $2x + 1$ , while applying some random noise to it. We'll see the optimized linear regression model for each.

Rather than transform the data, we'll transform the separator: this really highlights the overfitting effect.

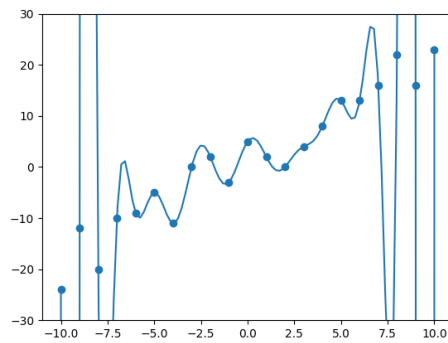
For ease, we'll exclude regularization: it does help mitigate this problem, but it doesn't totally solve it.



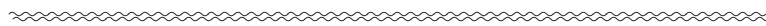
Here's the 1st order solution: in this case, correct for the underlying distribution. It fits our data fine.



5th and 15th order. The left model looks suspicious, and the right is way overfit. It's very unlikely that we know such an intricate pattern, from so little data.



20th order. We have one order for each data point: now, our model is capable of doing regression going through every single data point: as overfit as physically possible, perfectly matching the data.



### 5.2.1.3 Higher dimensions

Until now, we've only been focusing on the 1-D case of data. Let's change that. Let's consider a 2D dataset  $[x_1, x_2]^T$ .

Our typical model

$$\theta^T x + \theta_0 = \theta_1 x_1 + \theta_2 x_2 + \theta_0 \quad (5.7)$$

Is "order 1": the largest exponent is 1. This is still a "linear" model.

If we want to move up to order 2, we increase the max exponent, adding  $x_1^2$  and  $x_2^2$  to the basis.

However, this doesn't take full advantage of the expressiveness of our model: this only creates parabolas aligned with the  $x_1$  and  $x_2$  axes. How do we create other options?

Well, we created these options by multiplying  $x_1$  with another  $x_1$ . It seems like we could logically expand to multiplying  $x_1$  by  $x_2$ .

#### Definition 13

For **higher dimension**  $d > 1$  **polynomials**, we allow for multiplication **between variables**  $x_i$  and  $x_j$ .

The **order** of the polynomial is the maximum number of times you can **multiply variables** together.

For order  $k$ , the **sum of exponents** must be **less than or equal to** the order.

So, for  $d = 2$ , order=2, we get the basis:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_2 & x_2 x_1 & x_2^2 \end{bmatrix}^T \quad (5.8)$$

For  $d = 2$ ,  $\text{order}=3$ , it starts getting a bit messy: we have 10 different basis terms.

You don't need to memorize these.

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & x_2 & x_2 x_1 & x_2 x_1^2 & x_2^2 & x_2^2 x_1 & x_2^3 \end{bmatrix}^T \quad (5.9)$$

#### 5.2.1.4 Total number of features

How many do we have in general? Well, every term results from multiplying variables **at most**  $k$  times. Or, the exponents **add up** to at most  $k$ .

If we count 1 as a factor, we can say that the exponents always add up to  $k$  (since  $1^j$  has no effect). So, we have  $d + 1$  different numbers, which add up to exactly  $k$ .

We have  $d$  variables, so  $d + 1$  if we include 1 as a variable.

This is a well-known problem in combinatorics: how many ways are there to add up  $m$  numbers to total  $n$ ? The solution to this problem gives us:

$$\binom{(d+1) + k - 1}{k} = \binom{d+k}{k} = \frac{(d+k)!}{d!k!} \quad (5.10)$$

Explaining the math here is beside the point of this course. If you're curious, search up "stars and bars math", or visit [here](#).

#### 5.2.1.5 Summary of Polynomial Basis

##### Definition 14

The **polynomial basis** of order  $k$  and dimension  $d$  includes **every feature**

$$\prod_{i=1}^d x_i^{c_i} \quad (5.11)$$

Where all of the integer exponents  $c_i \geq 0$  add up to **at most**  $k$ .

Creating features such as:

$$x_1^k, x_1 x_2, x_2 x_3^3 x_6, 1, \dots \quad (5.12)$$

We can represent this in a table:

This table is different from the one we saw earlier!



| Order    | $d = 1$              | in general ( $d > 1$ )                                              |
|----------|----------------------|---------------------------------------------------------------------|
| 0        | $[1]$                | $[1]$                                                               |
| 1        | $[1, x]^T$           | $[1, x_1, \dots, x_d]^T$                                            |
| 2        | $[1, x, x^2]^T$      | $[1, x_1, \dots, x_d, x_1^2, x_1 x_2, \dots]^T$                     |
| 3        | $[1, x, x^2, x^3]^T$ | $[1, x_1, \dots, x_d, x_1^3, x_1 x_2, \dots, x_1 x_2 x_3, \dots]^T$ |
| $\vdots$ | $\vdots$             | $\vdots$                                                            |

### 5.2.1.6 Polynomial Basis in Action

Below, we'll show examples of how polynomial basis being used, to demonstrate just how useful it is.

But how do we use feature transformations? The best part: remember that we can view it as a linear separator? We can train it just the same way!

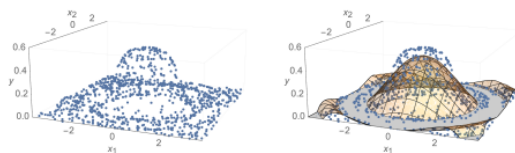
#### Concept 15

**Feature transformations don't change** how we train our model. We can still treat our model as a **linear** vector  $\theta$ , even if our data has been **non-linearly** transformed.

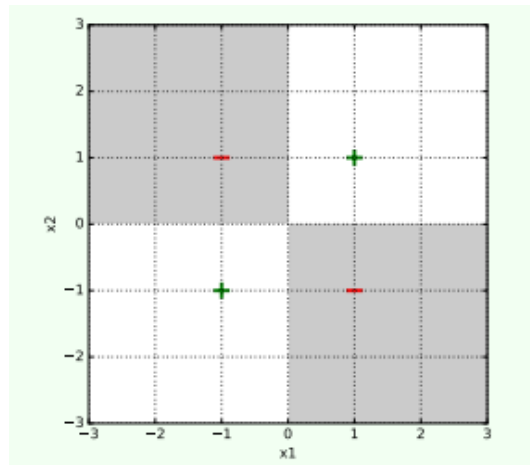
So, after we transform our data, we can use normal techniques (OLS, gradient descent, SGD) to fit our model.

In this situation, the benefits of regularization become more clear: by preventing  $\theta$  from becoming too large, we discourage a surface that is too "extreme", with larger changes across its surface.

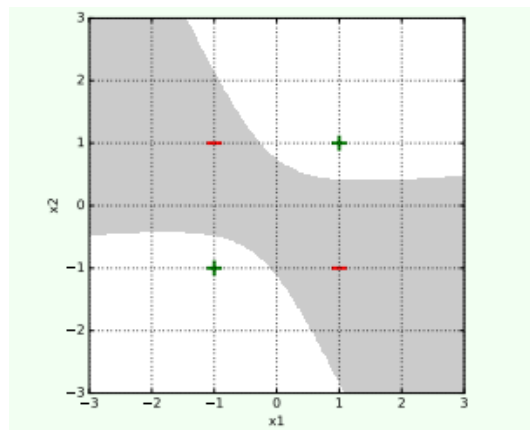
We'll train our model for various situations, to see what it can do. Different problems require different orders  $k$ , still.



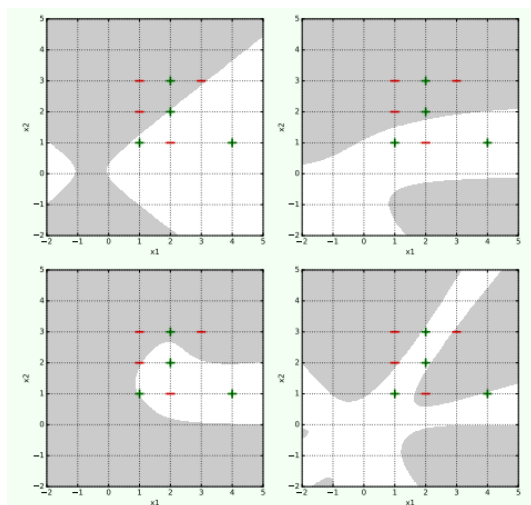
We start with the waveform we showed at the start of the chapter: on the left, we see a bunch of datapoints we want to take a regression over. With  $k = 8$ , we get a pretty good result.



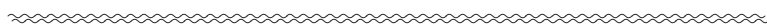
This is the classic "xor" problem: a typical case of "linearly unseparable". With  $k = 2$ , we can classify it well with the chosen model  $4x_1x_2 = 0$ .



This time we use gradient descent and a random initialization to get a less 'perfect', but still effective classification.



This dataset is pretty brutal: we try with  $k = 2, 3, 4$ , and finally 5. The shapes we get are... complex, to say the least. But, we successfully solve it with  $k = 5$ .



## 5.2.2 Radial Basis

Finally, we consider an alternative way to create a feature space.

- With the "polynomial basis" approach, we **combined features** to create more complex surfaces to **fit** the structure of the data.
- This "radial basis" approach, on the other hand, **combines data points** to **learn** more about the structure of the data.

What do we mean by that? Well, let's consider what we mean by "structure": when we're judging data, what sorts of patterns are we looking for?

Often, we're looking to see, "what data is near/similar to other data?" Similar data is more likely to behave similarly, after all.

So, it might be useful to include distance between data points as a feature: how do we implement this? Well, let's do this one-by-one: we'll create a feature for the distance to a single data point  $p$ .

We'll come back to these ideas when we talk about clustering!

We start with squared distance, for smoothness reasons.

$$\|p - x\|^2 \quad (5.13)$$

This feature would *grow* as data points get further apart, though. We want to see what data is *close*: the opposite.

We could use a function like  $\frac{1}{u}$ . However, this would explode to infinity as distances shrink: not good.

$e^{-u}$  is a better fit: it approaches 1 when  $u = 0$ , and, relatedly, it tends to drop off more smoothly and gradually than  $1/u$ .

Finally, we add a coefficient  $\beta$  to the exponent to give us more control: it will tell us how quickly our function decays with distance.

The word "decay" is used commonly for exponential decrease.

**Definition 16**

We define the **radial basis function**

$$f_p(x) = e^{-\beta \|p-x\|^2}$$

As a **feature** in the RBF feature transformation.

This transformation takes a data point  $p$  and provides a feature  $f_p(x)$  that represents "**closeness**" of  $x$  to  $p$ .

Note some useful properties of this transform:

- For small distances, this feature creates a **connection** between  $p$  and our data point: representing some local "structure" of **closeness**.
- If points are far away, this effect gradually **vanishes**: points which are **far** away have very little to do with each other.
- $\beta$  controls what is considered "close" and "far":
  - if  $\beta$  is large, points have to be very close for an effect.
  - if  $\beta$  is small, we have a larger "neighborhood" of points with a relevant  $f_p(x)$ .

**Definition 17**

The **radial basis functions (RBF)** transform takes each of the data points in the input, and uses it to create a set of **radial basis function** features.

Collectively, they make the **feature space**:

$$\phi(x) = [f_{x^{(1)}}(x), f_{x^{(2)}}(x), \dots, f_{x^{(n)}}(x)]^T$$

Where:

$$f_p(x) = e^{-\beta \|p - x\|^2}$$

This transform allows us to represent "closeness" within our dataset. With it, we can compare new data points to some "reference" points  $x^{(i)}$ .

It's often used to allow us to represent our dataset in a way that is approximate, but still useful.

This general idea is useful for problems like:

- Function approximation,
- Optimization,
- Reducing noise in signals

This approach is not limited to the "squared distance" idea of closeness, either: if you can come up with another way to define distance, you can use the same approach.

Reminder that "noise" just refers to anything undesired in the signal. Usually added by randomness or the environment.

These ways to define distance are called "distance metrics".

## 5.3 Hand-constructing features for real domains

So far, we've focused on transformations that handle two of our main problems (which have a lot of overlap):

Borrowed from the transformation definition above.

- Allow our model to handle new, more **complex** situations (more **expressiveness**)
- **Pre-process** our data to make it **easier** for our model to find **patterns**.

Now, we'd like to turn our attention to the last of the three:

- Convert our data into a **usable** format (if, say, the original format doesn't fit into our equations)

One challenge with our models are they rely on computation and calculation. This usually require our input to be something like a **number**.

But we don't always receive data in this way: words, brands, colors, and odd others data types, are often presented instead. Frequently, we even need to **adjust** our numerical inputs.

The **transformations** in this section address these kinds of problems. We take data that is informative, but not currently usable, and converting them to something we can **compute** with.

### Concept 18

Often, we have to **convert** between data types, in order to do the machine learning work we want to do.

This requires using the appropriate **transforms** to get data we can work with, without **losing** important **information**.

As mentioned above, we have to be **careful**: if we use the wrong data type, we can **lose** crucial information that our model could have made use of.



### 5.3.1 Discrete Features

One of the most common issues with data types is figuring out what to do with **discrete** features: ones that are broken up into categories.

These categories may or may not have an order, or some other important information. We need to use the right data type to keep as much information as possible. This will allow our model to more easily discover patterns.

We'll make the following assumption:

**Clarification 19**

In this textbook/course, we assume that all **input vectors**  $x$  should be **real-valued vectors** (or:  $x \in \mathbb{R}^d$ )

And now, we go through some common examples of feature transformations:

**5.3.1.1 Numeric**

First, we consider the case where our pre-processing **feature** is "*almost*" in a number format: each class could reasonably correspond to a **number**.

**Definition 20**

In the **numeric** transformation, we convert each of our  $k$  classes into a **number**.

- This approach is only appropriate if each class is roughly "numeric": it fits appropriately into the **real numbers**.
  - We have a clear **ordering**, and
  - The numbers have the **structure** of real numbers: **distance** between points, or the idea of **adding/multiplying**, makes sense.

**Example:** There are many ways to do this. Here, we evenly distribute values evenly between 0 and 1: \_\_\_\_\_

$$\left[ \frac{1}{k} \quad \frac{2}{k} \quad \dots \quad \frac{k-1}{k} \quad 1 \right], \quad \text{Class } i \longrightarrow \frac{i}{k} \quad (5.14)$$

Remember: which way you transform should reflect the nature of your data!

**5.3.1.2 Thermometer Code**

Next, we'll relax how number-like our feature is. This time, we don't need our data to behave like a number, but it does have an **ordering**. \_\_\_\_\_

Some examples:

- Results of an opinion poll:
  - "Strongly Agree", "Agree", "Neutral", "Disagree", "Strongly Disagree"
- Education level:
  - "Below High School", "High School Degree", "Associates Degree", "Bachelors", "Advanced Degree"
- Ranking of athletes

By "relax", we mean we'll remove some requirements for our feature, like being able to add them together.

In this case, we can't just use numbers  $\{1, 2, 3, \dots\}$ . Why not?

Because that implies that there's a specific "scaling" between points: Is the #1 athlete twice as good as the #2 athlete? Maybe, but that's not what the ranking tells us!

### Concept 21

Data that is **ordered** but not **numerical** cannot be represented with a **single real number**.

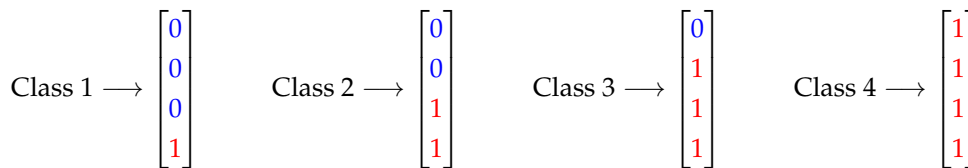
Otherwise, we might consider one element to be a certain amount "larger" or "smaller" than another, when that's not what **ordering** means.

**Example:** Suppose we assign  $\{1, 2, 3\}$  for  $\{\text{Disagree}, \text{Neutral}, \text{Agree}\}$ . The person who writes 'agree' is doesn't "agree three times as much" as the person who writes 'disagree'!

But, we still want to keep that ordering: counting up from one element to the next. How do we create an order without creating an exact, numeric difference?

Just now, we tried to count by increasing a single variable. But, there's another way to count: counting up using multiple different variables!

This approach is more similar to counting on your fingers.



This version allows us to avoid the problems we had before: this doesn't behave the same way as a **numerical** value.

To better understand what's going on, here's another way to frame it:

$\text{Class}(x)$  is just shorthand for, "which class is  $x$  in?"

$$\phi(x) = \begin{bmatrix} \text{Class}(x) \geq 4 \\ \text{Class}(x) \geq 3 \\ \text{Class}(x) \geq 2 \\ \text{Class}(x) \geq 1 \end{bmatrix} \quad (5.15)$$

**Example:** Suppose  $x$  is in class 3. The bottom three statements are all true, the top one is false: so we get  $[0, 1, 1, 1]^T$ .

This helps us understand why this encoding is so useful:

- We aren't directly "adding" variables to each other: they stay separated by **index**.
- When using a linear model  $\theta^T \phi(x)$ , each class matches a different  $\theta_i$ .
- Despite not behaving like numbers, "higher" classes in the order still keep track of all of the classes below them.

$\theta_i$  scales the  $i^{\text{th}}$  variable. So, each class can be scaled differently!



- **Example:** Class 2-4 all share the feature  $\text{Class}(x) \geq 2$  (equivalent to  $\text{Class}(x) > 1$ ).

This technique is called **thermometer encoding**.

### Definition 22

**Thermometer encoding** is a **feature transform** where we take each class and turn it into a feature vector  $\phi(x)$  where

$$\text{Class 1} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \text{Class 2} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 1 \end{bmatrix}, \quad \text{Class 3} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad \text{Class } k \rightarrow \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

- The **length of the vector** is the **number of classes**  $k$  we have.
- The  $i^{\text{th}}$  class has  $i$  **ones**.
- This transformation is only appropriate if the data
  - Is **ordered**,
  - But not **real number-compatible**: we can't add the values, or compare the "amount" of each feature.

**Example:** We reuse our example from earlier:

$$\phi(x_{\text{Class 1}}) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \phi(x_{\text{Class 2}}) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \quad \phi(x_{\text{Class 3}}) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad \phi(x_{\text{Class 4}}) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

~~~~~

5.3.1.3 One-hot Code

We introduced this technique in the **previous** chapter:

When there's no clear way to **simplify** our data, we accept the current discrete classes, and **convert** them to a number-like form.

- Examples:

- Colors: {Red, Orange, Yellow, Green, Blue, Purple}
- Animals: {Dog, Cat, Bird, Spider, Fish, Scorpion}
- Companies: {Walmart, Costco, McDonald's, Twitter}

We can't use thermometer code, because that suggests a natural **order**. And we definitely can't use real numbers.

Example: {Brown, Pink, Green} doesn't necessarily have an obvious order: you could force one, but there's no reason to.

But, we can use one idea from thermometer code: each class in a different variable.

$$\begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_k \end{bmatrix} \quad (5.16)$$

But in this case, we don't "build up" our vector: we replace $\text{Class}(x) \geq 4$ with $\text{Class}(x) = 4$.

$$\phi(x) = \begin{bmatrix} \text{Class}(x) = 4 \\ \text{Class}(x) = 3 \\ \text{Class}(x) = 2 \\ \text{Class}(x) = 1 \end{bmatrix} \quad (5.17)$$

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

Definition 23

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**.

$$\text{Class 1} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \text{Class 2} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad \text{Class 3} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad \text{Class } k \rightarrow \begin{bmatrix} 1 \\ \vdots \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

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

- This transformation is only appropriate if the data is
 - Does not have another **structure** we can reduce it to: it's neither like a **real number** nor **ordered**
 - We don't have an **alternative** representation that contains more (accurate) information.

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

$$\begin{bmatrix} \text{table} \\ \text{bed} \\ \text{couch} \\ \text{chair} \end{bmatrix} \quad (5.18)$$

For each class:

$$\mathbf{y}_{\text{chair}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \mathbf{y}_{\text{table}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{y}_{\text{couch}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \mathbf{y}_{\text{bed}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad (5.19)$$

~~~~~

#### 5.3.1.4 One-hot versus Thermometer

One common question is, "why can't we use one-hot for ordered data? We could sort the indices so they're in order".

However, there's a problem with this logic: the computer **doesn't care** about the order of the variables in an array: it contains no information!

Why is that? If the vector has an order, shouldn't that **affect** the model?

Well, remember that our model is represented by

$$\theta^T x = \sum_i \theta_i x_i \quad (5.20)$$

The vector format  $\theta^T x$  is just a way to **condense** our equation: the ordering goes away when we compute the sum.

#### Concept 24

**Order** of elements in a vector **don't** affect the behavior of our model.

This is because a linear model is a **sum**, and sums are the same regardless of **order**.

If our model has the same math regardless of order, then it can't use order information.

**Example:** We'll take a vector, and rearrange it.

Despite shuffling, these two equations are equivalent!

$$\theta^T \phi(x) = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \theta_4 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \longrightarrow (\theta^T)^*(\phi(x))^* = \begin{bmatrix} \theta_3 & \theta_1 & \theta_4 & \theta_2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

The math is the same, despite changing order: our model knows nothing about ordering.

#### Clarification 25

**One hot encoding cannot** encode information about ordering.

**Thermometer encoding** is required to **represent ordered objects**.

Why is thermometer encoding able to of representing ordering? Let's try shuffling it, too.

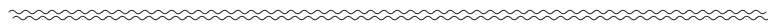
$$\theta^T \phi(x) = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \theta_4 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad (5.21)$$

$$(\theta^T)^*(\phi(x))^* = \begin{bmatrix} \theta_3 & \theta_1 & \theta_4 & \theta_2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad (5.22)$$

Even though we've changed the order, we still know this is the **third** in the order, because we have **three** 1's!

### Concept 26

Even though the **order of elements** in a vector **doesn't matter**, we can retrieve the order of **thermometer coding** based on the **number of 1's in the vector**.



#### 5.3.1.5 Factored Code

Now, we move away from number-like properties. Instead, what other patterns of our feature could be **useful**?

Sometimes, a single feature will contain multiple pieces of information. Separating those pieces (or **factors**) from each other can make it easier for our machine to understand.

- A **car** is often described by the "make" (brand) and "model" (which exact type of car by that brand).
  - These could be broken into two **features**: "make" is one feature, "model" is another.
  - **Example**: "Nissan Altima" becomes "Make: Nissan" and "Model: Altima".
- Most **blood types** are in the following categories: {A+, A-, B+, B-, AB+, AB-, O+, O-}.
  - You could factor this based on the letter, and positive/negative: {A, B, AB, O} and {+, -}.
  - Since "O" means we contain neither A nor B, we could factor the first feature further: {A, not A}, {B, not B}
  - Example: Using the first factoring, A- becomes [A, -]. Using the second it becomes [A, not B, -].
- **Addresses** have many parts: street number, zip code, state, etc.
  - Each of these can be given their own factor.

**Definition 27**

**Factored code** is a **feature transformation** where we take one **discrete class** and break it up into other discrete classifications, called **factors**.

$$\text{Class } m \text{ and } n \longrightarrow \text{Class } m, \text{Class } n \quad (5.23)$$

- This transformation is only appropriate if
  - We have some feature(s) that can be **broken up** into **simpler**, meaningful parts.

Often, we apply **other** feature transformations after factored coding.

Note the final comment: often, we turn a discrete class into multiple new discrete classes.

But, we still need to convert these into a usable, numeric-vector form!

**Example:** We can re-use our blood type example from above.

$$\phi(x) = \begin{bmatrix} x \text{ contains A} \\ x \text{ contains B} \\ x \text{ is +} \end{bmatrix} \quad (5.24)$$

Each of these are binary features. For example:

$$\phi(AB-) = \begin{bmatrix} \text{True} \\ \text{True} \\ \text{False} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \quad (5.25)$$

### 5.3.1.6 Binary Code

One possible way to encode data is to **compress** data using a **binary code**.

This might be tempting, because  $k$  values can be represented by  $\log_2(k)$  values.

**Example:** Suppose you have the one-hot code for 6, and want to represent it with binary:

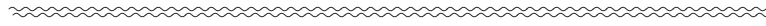
$$\text{Class } 6 \xrightarrow{\text{One-hot}} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}^T \xrightarrow{\text{Binary}} \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^T \quad (5.26)$$

**Please do not do this**

**Concept 28**

Using **binary code** to compress your features is usually a **bad idea**.

This forces your model to spend resources learning how to **decode** the binary code, before it can do the task you want it to!

**5.3.2 Text**

Just now, we showed different ways to transform **discrete** features.

Another very common data type we work with is **language**: bodies of text, online articles, corpora, etc.

Later in this course, we will discuss more powerful ways to analyze text, such as **sequential models**, and **transformers**.

Obligatory chatgpt reference.

There's a very simple encoding that we'll focus on here: the **bag of words** approach.

This approach is meant to be as simple as possible: for each word, we ask ourselves, "if this word in the text?", and answer yes (1) or no (0) for every single word.

**Definition 29**

The **bag of words** feature transformation takes a body of text, and creates a **feature** for every **word**: is that word in the text, or not?

$$\phi(x) = \begin{bmatrix} \text{Word 1 in } x \\ \text{Word 2 in } x \\ \vdots \\ \text{Word } k \text{ in } x \end{bmatrix} \quad (5.27)$$

This approach is used for **bodies of text**.

**Example:** Consider the following sentence: "She read a book."

With the words: {She, he, a, read, tired, water, book}

We get:

$$\phi(\text{"She read a book."}) = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \quad (5.28)$$

A couple weaknesses to this approach:

- Ignores the order of words and syntax of the sentence.
- Doesn't encode meaning directly.

- Duplicate words are only included once.
- It doesn't create much structure for our model to use.

But, it's very easy to implement.

### 5.3.3 Numeric values

Now, on to the (typically) more manageable data type:

#### Concept 30

Typically, if your feature is **already a numeric value**, then we usually want to **keep it as a data value**.

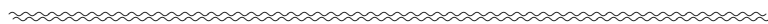
**Example:** Heart rate, stock price, distance, reaction time, etc.

However, this may not be true if there is some difference between different ranges of numbers:

- Being below or above the age of 18 (or 21) for legal reasons
- Temperature above or below boiling
- Different age ranges of children might need different range sizes: the difference between ages 1-2 is very different from ages 7-8.

#### Concept 31

Sometimes, if there are distinct **breakpoints**/boundaries between different values of a numerical feature, we might use **discrete** features to represent those.



#### 5.3.3.1 Standardizing Values

We still aren't done, if our data is numeric. We likely want to **scale** our features, so that they all tend to be in similar ranges.

Why is that? If some features are much **larger** than others, then they will have a much larger impact on the answer.

For example, suppose we have  $x_1 = 4000$ ,  $x_2 = 7$ :

$$h(x) = \theta^T x = 4000\theta_1 + 7\theta_2 \quad (5.29)$$

The first term is going to have a way bigger impact on  $h(x)$ . If we change  $x_1$  by 10%, that's going to be bigger than if we changed  $x_2$  by 100%! \_\_\_\_\_

$$\begin{aligned} 4000 * 10\% &= 400 \\ 7 * 100\% &= 7 \end{aligned}$$



**Concept 32**

If one **feature** is much **larger** than **another** feature, it will tend to have a much **larger** effect on the result.

This is often a bad thing: just because one feature is **larger**, doesn't mean it's more **important**!

**Example:** Income might be in the range of tens of thousands (10,000-100,000), while age is a two-digit number(20-100). Income will be weighed more heavily.

How do we solve that problem? We need to do two things:

- **Shift** the data so that our range is not too high/low. Our goal is to have it centered on 0.
  - We want it centered on 0 so we can distinguish between the above-average and below-average data points.
  - We do this by subtracting the **mean**, or the **average** of all of our data points.

Plus, it's easier to get all of our data to 0, rather than picking some arbitrary value.

$$\phi_1(x) = x - \bar{x} \quad (5.30)$$

- Scale the **range** of possible values, so they all vary by roughly the same amount.
- : So, if one variable tends to vary by a **larger** amount, it doesn't have a bigger impact on the result.

$$\phi(x_i) = \frac{x_i - \bar{x}_i}{\sigma_i} \quad (5.31)$$

Where  $\sigma$  is the **standard deviation**.

If you are interested, we define **standard deviation** below.

Note that each feature has its own  $\sigma_i$ : we have to compute this equation for each feature.

**Definition 33**

To make sure that all of our data is **on the same size scale**, we **normalize/standardize** our dataset using the operation

$$\phi(x_i) = \frac{x_i - \bar{x}_i}{\sigma_i}$$

For every variable  $x_i$  in a data point  $x$ .

- $\bar{x}_i$  is the **mean** of  $x_i$
- $\sigma_i$  is the **standard deviation** of  $x_i$

This results in a dataset which has

- A mean  $\bar{x}_i$  of **0**
- A standard deviation  $\sigma_i$  of **1**

So, all of our features have the same **average**, and **vary** by the same amount.

This prevents some features getting prioritized because they're on different size scales.

**Example:** Suppose we have 1-D data  $x = [1, 2, 3, 4, 5, 6]$

The mean is

$$\bar{x} = \frac{1 + 2 + 3 + 4 + 5 + 6}{6} = 3.5 \quad (5.32)$$

And the standard deviation is

$$\sigma = \sqrt{\frac{2.5^2 + 1.5^2 + .5^2 + .5^2 + 1.5^2 + 2.5^2}{6}} = \sqrt{\frac{35}{12}} \approx 1.7078 \quad (5.33)$$

~~~~~

5.3.3.2 Variance and Standard Deviation (Optional)

This section describes the origin of σ above. Feel free to skip if you're familiar.

In order to scale our data, we need a measure of how much our data **varies**. So, if our data varies by more, we can scale it down, and vice versa.

We can measure this using the **variance**.

Definition 34

We can measure how spread out/varying our data with **variance**

$$\sigma^2 = \sum_i \frac{(x^{(i)} - \bar{x})^2}{n} \quad (5.34)$$

In other words, the **average squared distance** from the **mean**.

Why do we square the terms? Same reason we square our loss:

- We want only positive values, for distance.
- We don't want to use absolute value, for smoothness.

We also get nicer statistical properties we won't discuss here.

However, this is too large: we want something similar to "average distance from the mean".

This is the average **squared** distance.

So, we take a square root!

Definition 35

A more common way to measure how our data varies is using **standard deviation** σ

$$\sigma = \sqrt{\sigma^2} = \sqrt{\sum_i \frac{(x - \bar{x})^2}{n}}$$

This term is **not** the average distance from the mean, but can be used for **scaling** our data in the same way.

This term allows us to scale our data appropriately. If our data varies by a larger amount, σ will be larger. So, $\frac{1}{\sigma}$ will cancel that variance out.

5.4 Terms

- Non-linear
- Transformation
- Feature
- Feature Transformation
- Polynomial Basis
- Order/Degree of a polynomial
- Radial Basis
- Discrete Feature
- Numeric transformation
- Thermometer Code
- One-hot Code
- Factored Code
- Binary Code
- Bag of Words
- Standardization
- Normalization
- Standard Deviation