Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2022

Concept 1

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

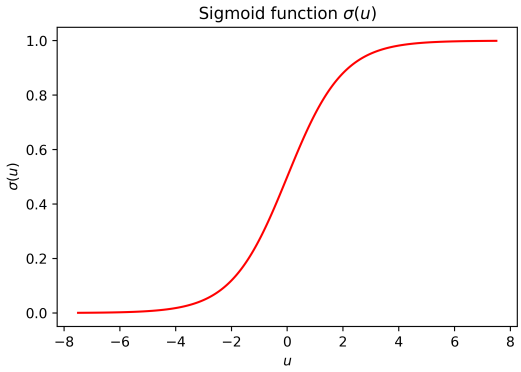
In order to create an accurate model, we have to use some **nonlinear** operation.

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.

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

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

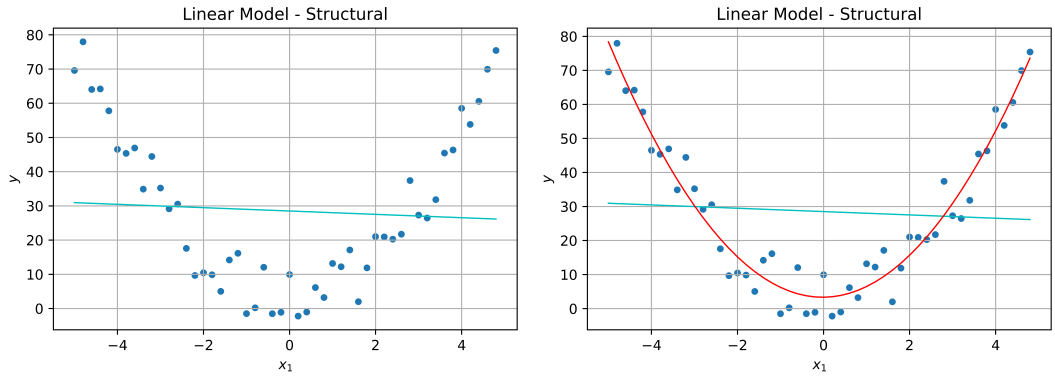


Figure 1: A linear function can't represent this dataset. However, a parabola can!

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!

$$h(x) = Ax^2 + Bx + C \tag{1}$$

It should be clear that his 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.

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

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**.

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:

$$h(x) = \theta_2 x^2 + \theta_1 x + \theta_0 \tag{2}$$

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 $x^0 = 1$.

$$h(x) = \overbrace{\sum_{i=1}^{k} \theta_i x^i}^{\text{Polynomial sum}} = \overbrace{\begin{bmatrix}\theta_0\\ \theta_1\\ \theta_2\\ \vdots\\ \theta_k\end{bmatrix} \cdot \begin{bmatrix}1\\ x\\ x^2\\ \vdots\\ x^k\end{bmatrix}}^{\text{Store as vectors}} = \overbrace{\theta^T}^{\text{Simplify}} \begin{bmatrix}1\\ x\\ x^2\\ \vdots\\ x^k\end{bmatrix} \tag{3}$$

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 $\phi(x)$:

$$\phi(x) = \begin{bmatrix}1\\ x\\ x^2\\ \vdots\\ x^k\end{bmatrix} \qquad h(x) = \theta^T \overbrace{\phi(x)}^{\text{New input}} \tag{4}$$

This is called **transforming** our input.

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:

$$h(x) = \sum_{i=1}^{k} \theta_i x^i = \begin{bmatrix}\theta_0\\ \theta_1\\ \theta_2\\ \vdots\\ \theta_k\end{bmatrix} \cdot \begin{bmatrix}1\\ x\\ x^2\\ \vdots\\ x^k\end{bmatrix} \tag{5}$$

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

Definition 3

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".

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 4

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.

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 5

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 **outweights** 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 6

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

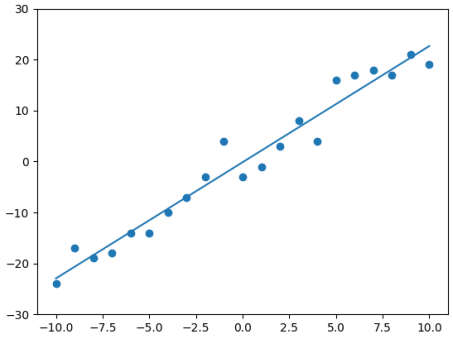
Because they can take on so many different, **complex** functions, they can very very closely **match** the original data set.

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

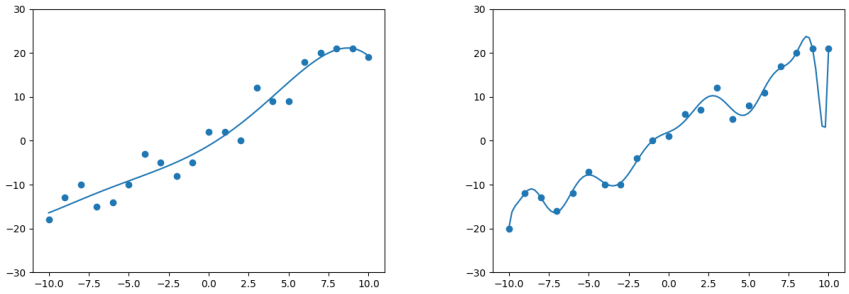
Let's see this in action: we'll generate some data based on $2x + 1$, while applying some random noise to it. We'll see the optimized linear regression model for each.

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

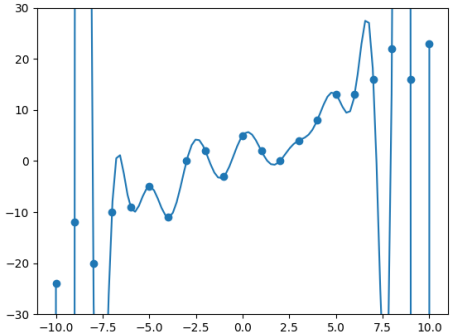
For ease, we'll exclude regularization: it does help mitigate this problem, but it doesn't totally solve it.



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.

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

$$\theta^T x + \theta_0 = \theta_1 x_1 + \theta_2 x_2 + \theta_0 \tag{6}$$

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 7

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:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_2 & x_2x_1 & x_2^2 \end{bmatrix}^T \tag{7}$$

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

You don't need to memorize these.

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & x_2 & x_2x_1 & x_2x_1^2 & x_2^2 & x^2x_1 & x^3 \end{bmatrix}^T \tag{8}$$

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:

$$\binom{(d+1)+k-1}{k} = \binom{d+k}{k} = \frac{(d+k)!}{d!k!} \tag{9}$$

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

Summary of Polynomial Basis

Definition 8

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

$$\prod_{i=1}^{d} x_i^{c_i} \tag{10}$$

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

Creating features such as:

$$x_1^k, \quad x_1 x_2, \quad x_2 x_3^3 x_6, \quad 1, \quad \ldots \tag{11}$$

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	$[1, x]^T$	$[1, x_1, \ldots, x_d]^T$
2	$[1, x, x^2]^T$	$[1, x_1, \ldots, x_d, x_1^2, x_1 x_2, \ldots]^T$
3	$[1, x, x^2, x^3]^T$	$[1, x_1, \ldots, x_1^3, x_1 x_2, \ldots, x_1 x_2 x_3, \ldots]^T$
$\vdots$	$\vdots$	$\vdots$

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!

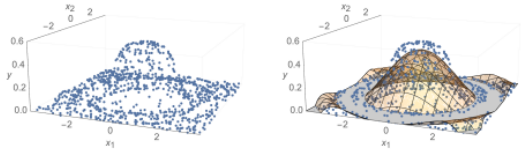
Concept 9

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

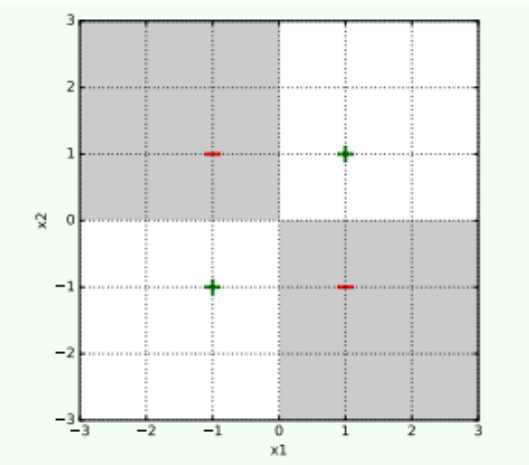
So, after we transform our data, we can use normal techniques (OLS, gradient descent, SGD) to fit our model.

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

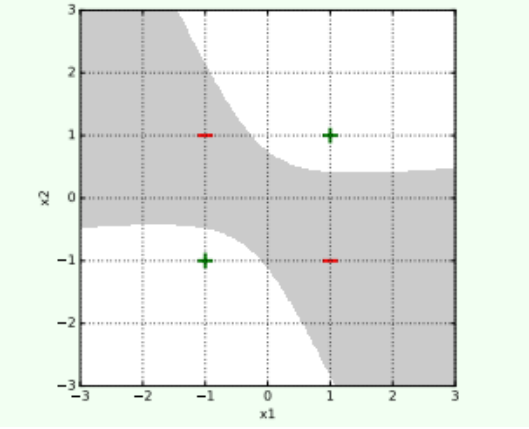
We'll train our model for various situations, to see what it can do. Different problems require different orders k, still.



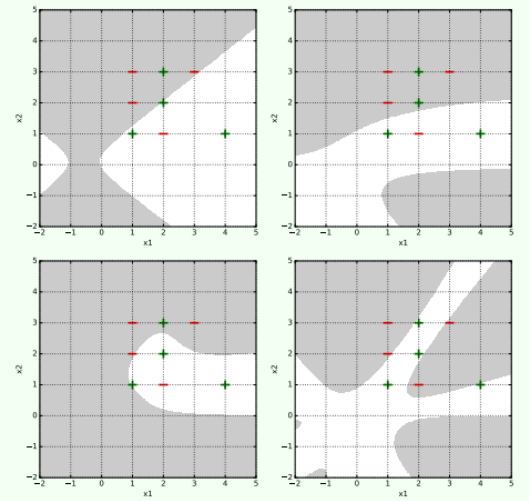
We start with the waveform we showed at the start of the chapter: on the left, we see a bunch of datapoints we want to take a regression over. With $k = 8$, we get a pretty good result.



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 $k = 5$.