

# Outline

Why a column of 1s?

Design matrix and vectorized implementation

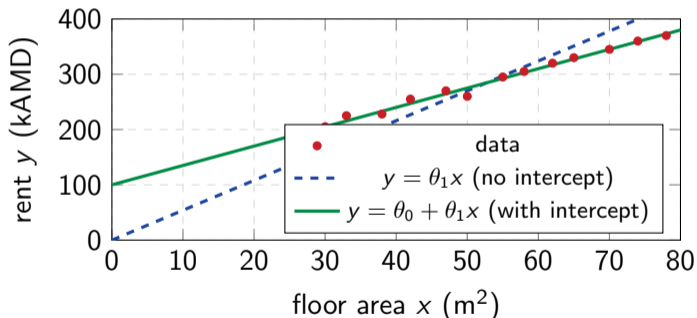
Deriving the normal equation

When OLS fails (and what to do)

Polynomial regression

## The intercept problem - can a line through the origin do it?

Suppose Yerevan studio rent vs. floor area. Even the smallest viable 25 m<sup>2</sup> studio doesn't cost (25 × per-m<sup>2</sup> price): there's a baseline (registration, condition, neighbourhood floor). A line forced through the origin can't absorb that baseline.



## What we learn from the picture

- ▶ The **dashed line** (no intercept) is forced through  $(0, 0)$ . To get near the data cloud it has to be much steeper than the cloud actually slopes.
- ▶ Result: it under-shoots small apartments and over-shoots large ones.
- ▶ The **solid line** (with intercept) is free to start at  $\theta_0 \approx 100$  kAMD and rise gently. Much smaller residuals.

Conclusion: we need an intercept term  $\theta_0$ . The next three frames make  $\theta_0$  disappear into the same dot product as the other coefficients.

## Two parameters, one notation?

**Why we want this:** we're about to vectorize all  $n$  predictions as a single matrix multiplication. A clean dot product makes that one line of NumPy code instead of a loop.

We *want* to write every prediction as:

$$f(\mathbf{x}) = \boldsymbol{\theta}^\top \mathbf{x}.$$

But  $\boldsymbol{\theta} = (\theta_0, \theta_1, \dots, \theta_p)^\top \in \mathbb{R}^{p+1}$  has **one more entry** than  $\mathbf{x} = (x_1, \dots, x_p)^\top \in \mathbb{R}^p$ .

Naive dot product:

$$\boldsymbol{\theta}^\top \mathbf{x} = \theta_1 x_1 + \dots + \theta_p x_p \quad (\text{missing } \theta_0!)$$

## The padding trick: prepend a 1

Replace every feature vector by an **augmented** version:

$$\mathbf{x} = (x_1, \dots, x_p)^\top \longmapsto \tilde{\mathbf{x}} = (1, x_1, \dots, x_p)^\top \in \mathbb{R}^{p+1}.$$

Then:

$$\boldsymbol{\theta}^\top \tilde{\mathbf{x}} = \theta_0 \cdot 1 + \theta_1 x_1 + \dots + \theta_p x_p = \theta_0 + \theta_1 x_1 + \dots + \theta_p x_p.$$

**Now the intercept lives inside  $\boldsymbol{\theta}$ .** No special case in any formula.

*From now on we drop the tilde and just write  $\mathbf{x} \in \mathbb{R}^{p+1}$  with a leading 1.*

## What this buys us, concretely

### Without the trick:

- ▶ Prediction:  $f(\mathbf{x}) = \theta_0 + \sum_{j=1}^p \theta_j x_j$
- ▶ Two cases for the gradient (one for  $\theta_0$ , one for  $\theta_j$ )
- ▶ In code: carry  $\theta_0$  as its own variable

### With the trick:

- ▶ Prediction:  $f(\mathbf{x}) = \boldsymbol{\theta}^\top \mathbf{x}$
- ▶ One gradient formula handles every parameter
- ▶ In code: `X = np.hstack([np.ones((n, 1)), X])`  
once at the top, then everything is `X @ theta`

**Caveat:** regularization (L03) usually does *not* penalize  $\theta_0$ . The penalty term acts on  $\theta_{1:p}$  only - common gotcha when implementing Ridge by hand.

## One observation $\rightarrow$ matrix-vector product

For a single augmented input  $\mathbf{x}^{(i)} \in \mathbb{R}^{p+1}$ :

$$\hat{y}^{(i)} = \boldsymbol{\theta}^\top \mathbf{x}^{(i)}.$$

Predictions for *all*  $n$  observations:

$$\begin{pmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(n)} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\theta}^\top \mathbf{x}^{(1)} \\ \boldsymbol{\theta}^\top \mathbf{x}^{(2)} \\ \vdots \\ \boldsymbol{\theta}^\top \mathbf{x}^{(n)} \end{pmatrix}.$$

Each entry of the right-hand vector is the dot product of  $\boldsymbol{\theta}$  with one input. We can collect all of them into a single matrix-vector product.

## The design matrix $\mathbf{X}$

Stack the augmented inputs as the **rows** of  $\mathbf{X}$ :

$$\mathbf{X} = \begin{pmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_p^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_p^{(2)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_1^{(n)} & x_2^{(n)} & \cdots & x_p^{(n)} \end{pmatrix} \in \mathbb{R}^{n \times (p+1)}.$$

Shape mnemonic: **rows are observations, columns are features**. First column is always 1 (intercept).

In NumPy: `X.shape == (n, p + 1)`.

## The full prediction in one expression

All  $n$  predictions become a single matrix-vector product:

$$\boxed{\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}} \quad \text{where} \quad \hat{\mathbf{y}} \in \mathbb{R}^n, \quad \mathbf{X} \in \mathbb{R}^{n \times (p+1)}, \quad \boldsymbol{\theta} \in \mathbb{R}^{p+1}.$$

Shape check (do this every time you write a new matrix expression):

$$\underbrace{(n \times (p+1))}_{\mathbf{X}} \cdot \underbrace{((p+1) \times 1)}_{\boldsymbol{\theta}} = \underbrace{(n \times 1)}_{\hat{\mathbf{y}}}.$$

Residuals and SSE also vectorize:

$$\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{y} - \mathbf{X}\boldsymbol{\theta}, \quad \mathcal{R}_{\text{emp}} = \sum_{i=1}^n r_i^2 = \mathbf{r}^T \mathbf{r} = \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2.$$

## Vectorized vs. loop: code

**Slow (Python loop) - teaching example only, DO NOT use in real code:**

```
yhat = np.zeros(n)
for i in range(n):                # row by row
    s = 0.0
    for j in range(p + 1):        # element by element
        s += theta[j] * X[i, j]
    yhat[i] = s
```

**Fast (vectorized NumPy) - the actual implementation:**

```
yhat = X @ theta                # one matrix-vector product
residuals = y - yhat
sse = residuals @ residuals
```

The @ operator is matrix multiplication (covered in `python_libs/02_numpy`).

## Why vectorization wins

Two reasons the second version is dramatically faster:

1. **No per-element Python overhead.** Every iteration of the loop pays the cost of a Python bytecode dispatch. The vectorized version hands the whole job to a single call into compiled linear-algebra code (the BLAS library, written in C / Fortran).
2. **The CPU can process many numbers in parallel.** BLAS routines stream contiguous chunks of memory through SIMD instructions, where one CPU instruction operates on 4-16 floats at once. The Python loop processes one number at a time.

Rule of thumb on this laptop (Intel Iris Xe, 16 GB): for  $n = 10^5$ ,  $p = 50$ , the loop takes seconds; `X @ theta` takes milliseconds.

**Habit to build:** when you catch yourself writing a Python `for` loop over rows of an array, stop and ask “what matrix op does this?”.

## The setup

We want the  $\theta$  that minimizes empirical risk:

$$\hat{\theta} \in \arg \min_{\theta} \mathcal{R}_{\text{emp}}(\theta) = \arg \min_{\theta} \sum_{i=1}^n \left( y^{(i)} - \theta^{\top} \mathbf{x}^{(i)} \right)^2 = \arg \min_{\theta} \|\mathbf{y} - \mathbf{X}\theta\|_2^2.$$

The function  $\mathcal{R}_{\text{emp}}(\theta)$  is a smooth (differentiable everywhere) function with no constraints on  $\theta$ . So the strategy from calculus:

1. Expand the squared norm so we can differentiate.
2. Take the gradient w.r.t.  $\theta$ .
3. Set the gradient to zero  $\rightarrow$  this gives the *normal equations*.
4. Solve the resulting linear system for  $\hat{\theta}$ .

## Speedrun: the derivation in 4 lines

Before the careful version, here is the whole thing in 4 lines. Read these once, then we'll justify each step.

$$\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 = \mathbf{y}^\top \mathbf{y} - 2\boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{y} + \boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{X} \boldsymbol{\theta}$$

$$\nabla_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = -2\mathbf{X}^\top \mathbf{y} + 2\mathbf{X}^\top \mathbf{X} \boldsymbol{\theta}$$

$$\text{set to 0: } \mathbf{X}^\top \mathbf{X} \hat{\boldsymbol{\theta}} = \mathbf{X}^\top \mathbf{y} \quad (\text{normal equations})$$

$$\boxed{\hat{\boldsymbol{\theta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}} \quad (\text{OLS estimator})$$

Two facts make this work:

- ▶  $\nabla_{\boldsymbol{\theta}} \mathbf{a}^\top \boldsymbol{\theta} = \mathbf{a}$  and  $\nabla_{\boldsymbol{\theta}} \boldsymbol{\theta}^\top A \boldsymbol{\theta} = 2A\boldsymbol{\theta}$  for symmetric  $A$ .
- ▶  $\mathbf{X}^\top \mathbf{X}$  is symmetric:  $(\mathbf{X}^\top \mathbf{X})^\top = \mathbf{X}^\top (\mathbf{X}^\top)^\top = \mathbf{X}^\top \mathbf{X}$ . ✓

## Step 1: expand the squared norm

Recall:  $\|\mathbf{v}\|_2^2 = \mathbf{v}^\top \mathbf{v}$  for any vector  $\mathbf{v}$  (the squared norm is the dot product with itself).

Apply with  $\mathbf{v} = \mathbf{y} - \mathbf{X}\boldsymbol{\theta}$ :

$$\begin{aligned}\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) &= (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^\top (\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) \\ &= \mathbf{y}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{X}\boldsymbol{\theta} - (\mathbf{X}\boldsymbol{\theta})^\top \mathbf{y} + (\mathbf{X}\boldsymbol{\theta})^\top (\mathbf{X}\boldsymbol{\theta}).\end{aligned}$$

**Why the middle two terms are equal.** Each one is a scalar:

$$\underbrace{\mathbf{y}^\top}_{1 \times n} \underbrace{\mathbf{X}}_{n \times (p+1)} \underbrace{\boldsymbol{\theta}}_{(p+1) \times 1} \rightarrow 1 \times 1 \rightarrow \text{a number.}$$

A scalar equals its own transpose, so  $\mathbf{y}^\top \mathbf{X}\boldsymbol{\theta} = (\mathbf{y}^\top \mathbf{X}\boldsymbol{\theta})^\top = \boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{y}$ . Combine:

$$\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = \mathbf{y}^\top \mathbf{y} - 2\boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{y} + \boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{X}\boldsymbol{\theta}.$$

## Step 2: vector-calculus rules we need

**Two rules.** For constants  $\mathbf{a} \in \mathbb{R}^d$  and symmetric  $A \in \mathbb{R}^{d \times d}$ :

$$\nabla_{\boldsymbol{\theta}} \mathbf{a}^{\top} \boldsymbol{\theta} = \mathbf{a}$$

$$\nabla_{\boldsymbol{\theta}} \boldsymbol{\theta}^{\top} A \boldsymbol{\theta} = 2 A \boldsymbol{\theta}$$

**Where rule 1 comes from** (one-line derivation):

$$\mathbf{a}^{\top} \boldsymbol{\theta} = \sum_j a_j \theta_j \Rightarrow \frac{\partial}{\partial \theta_j} (\mathbf{a}^{\top} \boldsymbol{\theta}) = a_j \Rightarrow \nabla_{\boldsymbol{\theta}} (\mathbf{a}^{\top} \boldsymbol{\theta}) = \mathbf{a}.$$

Rule 2 follows similarly (see math module 07 - multivariate calculus).

**1D sanity check.** If  $d = 1$  and  $a, A$  are scalars,

$$\frac{d}{d\theta} (a\theta) = a, \quad \frac{d}{d\theta} (A\theta^2) = 2A\theta. \quad \checkmark$$

And  $A = \mathbf{X}^{\top} \mathbf{X}$  is symmetric because  $(\mathbf{X}^{\top} \mathbf{X})^{\top} = \mathbf{X}^{\top} \mathbf{X}$ , so rule 2 applies.

### Step 3: take the gradient

Differentiate  $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = \mathbf{y}^\top \mathbf{y} - 2\boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{y} + \boldsymbol{\theta}^\top (\mathbf{X}^\top \mathbf{X}) \boldsymbol{\theta}$  term-by-term:

- ▶  $\nabla_{\boldsymbol{\theta}} \mathbf{y}^\top \mathbf{y} = 0$  (no  $\boldsymbol{\theta}$ )
- ▶  $\nabla_{\boldsymbol{\theta}} (-2\boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{y}) = -2\mathbf{X}^\top \mathbf{y}$  (rule 1 with  $\mathbf{a} = \mathbf{X}^\top \mathbf{y}$ )
- ▶  $\nabla_{\boldsymbol{\theta}} \boldsymbol{\theta}^\top (\mathbf{X}^\top \mathbf{X}) \boldsymbol{\theta} = 2(\mathbf{X}^\top \mathbf{X}) \boldsymbol{\theta}$  (rule 2)

Sum:

$$\nabla_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = -2\mathbf{X}^\top \mathbf{y} + 2\mathbf{X}^\top \mathbf{X} \boldsymbol{\theta}.$$

## Step 4: set the gradient to zero

At a stationary point of  $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ :

$$-2\mathbf{X}^T \mathbf{y} + 2\mathbf{X}^T \mathbf{X} \boldsymbol{\theta} = 0 \quad \implies \quad \boxed{\mathbf{X}^T \mathbf{X} \hat{\boldsymbol{\theta}} = \mathbf{X}^T \mathbf{y}}$$

These are the **normal equations** - a linear system in  $\boldsymbol{\theta}$ .

**Why “normal”?** The residual  $\mathbf{r} = \mathbf{y} - \mathbf{X}\hat{\boldsymbol{\theta}}$  ends up *orthogonal* (normal) to every column of  $\mathbf{X}$ . The system above says exactly

$$\mathbf{X}^T \mathbf{r} = 0.$$

Geometrically: each column of  $\mathbf{X}$  is a “feature direction”. The optimal residual has no remaining correlation with any feature - if it did, we could reduce the loss by moving  $\boldsymbol{\theta}$  along that direction.

## Step 5: solve for $\hat{\theta}$

If  $\mathbf{X}^\top \mathbf{X}$  is invertible, left-multiply both sides of  $\mathbf{X}^\top \mathbf{X} \hat{\theta} = \mathbf{X}^\top \mathbf{y}$  by  $(\mathbf{X}^\top \mathbf{X})^{-1}$ :

$$\hat{\theta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

This is the **ordinary least squares (OLS) estimator**.

**Why this is the global minimum (not just a stationary point).** The Hessian of  $\mathcal{R}_{\text{emp}}(\theta)$  is

$$\nabla_{\theta}^2 \mathcal{R}_{\text{emp}}(\theta) = 2 \mathbf{X}^\top \mathbf{X}.$$

$\mathbf{X}^\top \mathbf{X}$  is positive semi-definite, meaning the loss surface is a *bowl* (never curves down). A stationary point of a bowl is the bottom. So  $\hat{\theta}$  is the global minimum.

Bonus: closed-form  $\rightarrow$  no iteration, no learning rate, no convergence questions. The price:  $\mathbf{X}^\top \mathbf{X}$  has to be invertible (next section).

## Worked numerical example

Let  $\mathbf{X} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix}$ ,  $\mathbf{y} = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix}$ . Compute  $\hat{\boldsymbol{\theta}}$  by hand.

$$\mathbf{X}^T \mathbf{X} = \begin{pmatrix} 3 & 6 \\ 6 & 14 \end{pmatrix}, \quad \mathbf{X}^T \mathbf{y} = \begin{pmatrix} 10 \\ 23 \end{pmatrix}$$

$$\det(\mathbf{X}^T \mathbf{X}) = 3 \cdot 14 - 6 \cdot 6 = 6$$

$$(\mathbf{X}^T \mathbf{X})^{-1} = \frac{1}{6} \begin{pmatrix} 14 & -6 \\ -6 & 3 \end{pmatrix}$$

$$\hat{\boldsymbol{\theta}} = \frac{1}{6} \begin{pmatrix} 14 & -6 \\ -6 & 3 \end{pmatrix} \begin{pmatrix} 10 \\ 23 \end{pmatrix} = \frac{1}{6} \begin{pmatrix} 2 \\ 9 \end{pmatrix} = \begin{pmatrix} 1/3 \\ 3/2 \end{pmatrix}.$$

**Fitted line:**  $\hat{y} = \frac{1}{3} + \frac{3}{2}x$ .

**Residuals.**  $\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\theta}} = (\frac{11}{6}, \frac{10}{3}, \frac{29}{6})$ , so

$$\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}} = (\frac{1}{6}, -\frac{2}{6}, \frac{1}{6}).$$

**Orthogonality check:**  $\mathbf{X}^T \mathbf{r}$  should be 0.

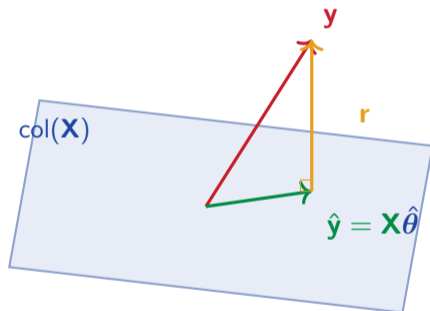
$$\frac{1}{6} - \frac{2}{6} + \frac{1}{6} = 0 \quad \checkmark$$

$$1 \cdot \frac{1}{6} + 2 \cdot (-\frac{2}{6}) + 3 \cdot \frac{1}{6} = 0 \quad \checkmark$$

Both columns of  $\mathbf{X}$  are orthogonal to  $\mathbf{r}$ , exactly as the normal equations require.

## Geometric picture

$\hat{y} = \mathbf{X}\hat{\theta}$  is the **orthogonal projection** of  $y$  onto the **column space** of  $\mathbf{X}$  (the set of all vectors  $\mathbf{X}\theta$  as  $\theta$  ranges over  $\mathbb{R}^{p+1}$  - the subspace reachable from our features).



**OLS in one sentence:** pick  $\theta$  so that the residual is perpendicular to every feature direction.

## When does $\mathbf{X}^\top \mathbf{X}$ fail to be invertible?

**Fact (from linear algebra):**  $\mathbf{X}^\top \mathbf{X}$  is invertible exactly when the  $p + 1$  columns of  $\mathbf{X}$  are linearly independent.

Three common ways the columns become *dependent*:

- ▶ **Multicollinearity.** Two features are essentially the same. Example: weight in kg and weight in pounds (one is a constant multiple of the other).
- ▶ **Redundant encoding (the dummy trap).** Including *all* dummy columns for a categorical variable AND an intercept. The dummies always sum to the all-ones column.
- ▶  $p \geq n$ . More features than observations. Columns live in  $\mathbb{R}^n$  - you can't fit  $p + 1 > n$  linearly independent vectors in  $\mathbb{R}^n$ .

## What to do when it fails

Options, in order of practicality:

1. **Drop the redundant column.** Cheap fix for the dummy trap and for obvious unit-of-measure duplicates.
2. **Use the pseudo-inverse:** `theta = np.linalg.pinv(X) @ y`. Works via SVD; returns the minimum-norm solution when the system is under-determined.
3. **Regularize.** Replace  $\mathbf{X}^\top \mathbf{X}$  with  $\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}$ . Always invertible for  $\lambda > 0$ . This is **Ridge regression** - L03.
4. **Gradient descent.** Works even when the closed form doesn't, and scales to large  $n, p$  - L02.

## Computational cost

Forming and solving the normal equations costs:

- ▶ Building  $\mathbf{X}^\top \mathbf{X}$ :  $\mathcal{O}(np^2)$  multiplications.
- ▶ Solving the  $(p + 1) \times (p + 1)$  system (LU / Cholesky):  $\mathcal{O}(p^3)$ .
- ▶ Total:  $\mathcal{O}(np^2 + p^3)$ .

Big- $\mathcal{O}$  implications:

- ▶ Fine for  $p \lesssim 10^3$ . A laptop solves it instantly.
- ▶ Bad for  $p \gtrsim 10^4$ . The  $p^3$  term dominates. Use gradient descent (L02) or stochastic methods.
- ▶ Numerically: prefer `np.linalg.lstsq(X, y)` or `np.linalg.solve(X.T @ X, X.T @ y)` over an explicit `np.linalg.inv(...)` call. Same answer, much better conditioning.

## Same model, different features

Linear regression is linear **in the parameters**  $\theta$ , not in the inputs  $\mathbf{x}$ . Nothing stops us from feeding it nonlinear functions of  $x$ .

For a single feature  $x \in \mathbb{R}$ , the degree- $d$  polynomial model is

$$f(\mathbf{x}) = \theta_0 + \theta_1 x + \theta_2 x^2 + \dots + \theta_d x^d.$$

**“But  $x^2$  is nonlinear!”** Yes - the *fitted function* is nonlinear in  $x$ . But the *optimization problem* is still linear regression: we treat  $x^2$  as just another column of the design matrix. Define the basis expansion

$$\phi(x) = (1, x, x^2, \dots, x^d)^\top \in \mathbb{R}^{d+1},$$

then  $f(\mathbf{x}) = \boldsymbol{\theta}^\top \phi(x)$  - linear in  $\boldsymbol{\theta}$ .

## Polynomial design matrix

Stack the basis-expanded inputs into a new design matrix:

$$\mathbf{X}_\phi = \begin{pmatrix} 1 & x^{(1)} & (x^{(1)})^2 & \dots & (x^{(1)})^d \\ 1 & x^{(2)} & (x^{(2)})^2 & \dots & (x^{(2)})^d \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x^{(n)} & (x^{(n)})^2 & \dots & (x^{(n)})^d \end{pmatrix} \in \mathbb{R}^{n \times (d+1)}.$$

**Key observation:** from OLS's perspective  $\mathbf{X}_\phi$  is just another design matrix. The normal equation is unchanged:

$$\hat{\theta} = (\mathbf{X}_\phi^\top \mathbf{X}_\phi)^{-1} \mathbf{X}_\phi^\top \mathbf{y}.$$

In code: `from sklearn.preprocessing import PolynomialFeatures`. For multiple features, the same idea also generates *interaction* terms like  $x_1 x_2$ ,  $x_1^2 x_2$ .

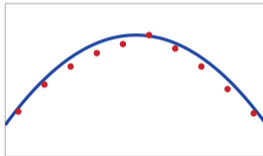
## Predict-first: which curve generalizes?

Three fits on the same 10 noisy samples from  $y = \sin(\pi x) + \varepsilon$ ,  $x \in [0, 1]$ .

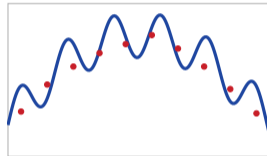
curve A



curve B



curve C



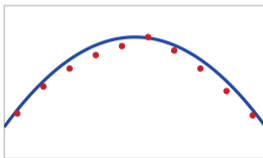
## Predict-first: which curve generalizes?

Three fits on the same 10 noisy samples from  $y = \sin(\pi x) + \varepsilon$ ,  $x \in [0, 1]$ .

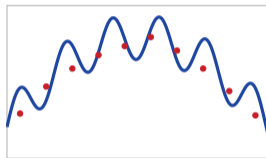
curve A



curve B



curve C

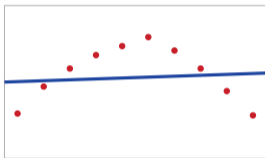


**Predict, then read on:** (1) Which has the smallest training error? (2) Which has the smallest test error? (3) Which would you deploy?

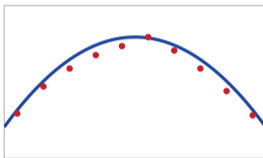
## Predict-first: which curve generalizes?

Three fits on the same 10 noisy samples from  $y = \sin(\pi x) + \varepsilon$ ,  $x \in [0, 1]$ .

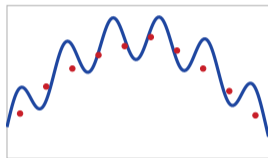
curve A



curve B



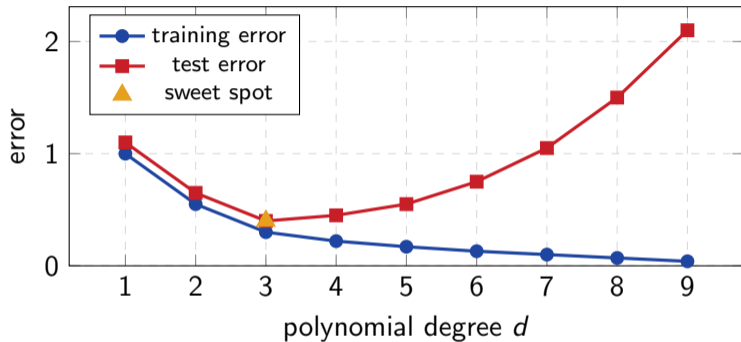
curve C



**Predict, then read on:** (1) Which has the smallest training error? (2) Which has the smallest test error? (3) Which would you deploy?

Answers: A is degree 1 (*underfit*, biggest train AND test error). C is degree 9 (*overfit*, tiny train error, biggest test error). **B** is degree 3 - small train error AND small test error. Deploy B.

## The U-shape of test error



Training error keeps falling. Test error has a **U-shape**. Minimum at the sweet spot. *Numbers illustrative; exact values depend on noise level and sample size.*

## What polynomial regression teaches us

1. **Linearity is in  $\theta$ , not in  $x$ .** Almost every model in classical ML (kernel methods, basis splines, fixed-feature neural networks) is “linear regression on engineered features”.
2. **More features = more flexibility = more overfitting risk.** The U-shape is the bias-variance tradeoff (formalized in L04).
3. **We need a way to penalize complexity.** If we can't drop features by hand, can we penalize their size? → **regularization** (L03).
4. **We need a way to evaluate honestly.** Training error is not enough. → **hold-out + cross-validation** (L04, L05).

## Wrap-up

- ▶ Pad inputs with a 1 so the intercept lives inside  $\theta$ . One formula everywhere.
- ▶ Design matrix  $\mathbf{X}$ : rows are observations, columns are features. Predictions vectorize as  $\hat{\mathbf{y}} = \mathbf{X}\theta$ .
- ▶ Normal equation  $(\mathbf{X}^\top \mathbf{X}) \hat{\theta} = \mathbf{X}^\top \mathbf{y}$  comes from  $\nabla \mathcal{R}_{\text{emp}}(\theta) = 0$ . Geometrically: residual perpendicular to every feature column.
- ▶ OLS fails when  $\mathbf{X}^\top \mathbf{X}$  is singular. Fixes: drop a column, pseudo-inverse, ridge, or gradient descent.
- ▶ Polynomial regression is OLS with  $\phi(x) = (1, x, x^2, \dots, x^d)$ . Same algebra, more flexibility - and more overfitting risk.

### HW01b - implement and verify.

1. Implement OLS from scratch using only `np.linalg.solve(X.T @ X, X.T @ y)`.
2. Verify against `sklearn.linear_model.LinearRegression`:  
`np.allclose(theta_mine, sk.coef_, atol=1e-10)`.
3. Extend to polynomial degree 3 by hand-constructing  $\mathbf{X}_\phi$ . Compare to `PolynomialFeatures`.
4. Bonus: time the loop version vs. the vectorized version for  $n = 10^5$ ,  $p = 50$  and report the ratio.