Artificial Intelligence

Lecture 8: Feature Transforms, Cross-Validation & Gradient Descent

Prof. Antonio Khalil Moretti

Week 9

SCIS 432
Spelman College

Check Your Understanding: Simple Linear Regression

Suppose we have fit a simple linear regression model predicting house price (in \$s) from square footage:

$$\hat{y} = 80{,}000 + 1{,}800\,x$$

where x is square footage and $\hat{y}$ is predicted price in \$.

Question 1

A buyer is comparing two houses. House B has **10 more square feet** than House A. According to the model, how much more expensive is House B?

Check Your Understanding: Simple Linear Regression

Suppose we have fit a simple linear regression model predicting house price (in \$s) from square footage:

$$\hat{y} = 80{,}000 + 1{,}800\,x$$

where x is square footage and $\hat{y}$ is predicted price in \$.

Question 1

A buyer is comparing two houses. House B has **10 more square feet** than House A. According to the model, how much more expensive is House B?

Answer:

$\Delta\hat{y} = 1{,}800 \times 10 = \$18{,}000$

Each extra sqft adds \$1,800. The slope β_1 is the *marginal effect* of one unit increase in x.

Question 2

What does the intercept $\beta_0 = 80{,}000$ mean here? Does it make practical sense?

Check Your Understanding: Simple Linear Regression

Suppose we have fit a simple linear regression model predicting house price (in \$s) from square footage:

$$\hat{y} = 80{,}000 + 1{,}800\,x$$

where x is square footage and $\hat{y}$ is predicted price in \$.

Question 1

A buyer is comparing two houses. House B has **10 more square feet** than House A. According to the model, how much more expensive is House B?

Answer:

$\Delta\hat{y} = 1{,}800 \times 10 = \$18{,}000$

Each extra sqft adds \$1,800. The slope β_1 is the *marginal effect* of one unit increase in x.

Question 2

What does the intercept $\beta_0 = 80{,}000$ mean here? Does it make practical sense?

Answer:

It predicts a house with 0 sqft costs \$80,000 — which is *nonsensical*. Intercepts often lack direct interpretation. The model is only valid within the range of observed data.

Check Your Understanding: Multiple Regression Coefficients

We fit a multiple regression model with three features:

$$\hat{y} = 40{,}000 + 1{,}500\,x_1 + 8{,}000\,x_2 - 5{,}000\,x_3$$

where $x_1 =$ sqft, $x_2 =$ bathrooms, $x_3 =$ bedrooms, $\hat{y} =$ price (\$).

Question: What does each coefficient represent? A house has 3 bedrooms, 2 bathrooms, 1200 sqft. We add a bathroom. What happens to the predicted price?

Check Your Understanding: Multiple Regression Coefficients

We fit a multiple regression model with three features:

$$\hat{y} = 40{,}000 + 1{,}500\,x_1 + 8{,}000\,x_2 - 5{,}000\,x_3$$

where x_1 = sqft, x_2 = bathrooms, x_3 = bedrooms, $\hat{y}$ = price (\$).

Question: What does each coefficient represent? A house has 3 bedrooms, 2 bathrooms, 1200 sqft. We add a bathroom. What happens to the predicted price?

Answer:

Each β_i is the effect of feature i **holding all others fixed**:

- β_1=1,500: each extra sqft adds \$1,500, *given fixed bedrooms and bathrooms*
- β_2=8,000: each extra bathroom adds \$8,000, *given fixed sqft and bedrooms*
- β_3=−5,000: each extra bedroom *reduces* price by \$5,000 *given fixed sqft and bathrooms*

Adding one bathroom:

$\Delta\hat{y} = 8{,}000 \times 1 = \$8{,}000$

Why can β_3 be negative?

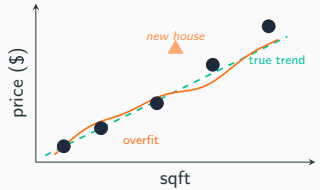
Holding sqft constant, more bedrooms means *smaller* rooms. The coefficient captures the *partial* effect, not the raw correlation.

Multiple regression coefficients can surprise you — they control for all other variables simultaneously.

Check Your Understanding: Overfitting

You train a polynomial regression model on **5 houses**. It predicts all 5 training prices **perfectly**: Train MSE $= 0$.

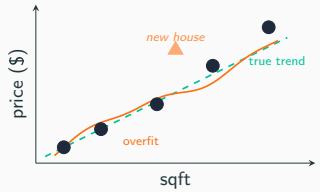
Is this a good model? Should you use it to price a 6th house?



Check Your Understanding: Overfitting

You train a polynomial regression model on **5 houses**. It predicts all 5 training prices **perfectly**: Train MSE $= 0$.

Is this a good model? Should you use it to price a 6th house?



No — this is a bad model.

Lagrange Interpolation Theorem: For any n distinct points $(x_1, y_1), \ldots, (x_n, y_n)$ there exists a *unique* polynomial of degree $\leq n-1$ passing through all of them:

$$p(x) = \sum_{i=1}^{n} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

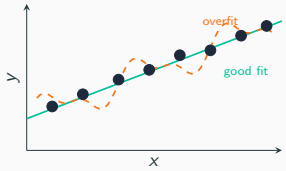
With 5 points a degree-4 polynomial is *guaranteed* to achieve Train MSE $= 0$. This is not skill — it is mathematics.

The model has **memorised noise**. Predictions on new houses will be wildly wrong.

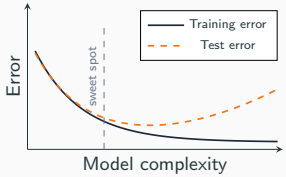
Zero training error is a red flag, not a trophy.

The Generalisation Problem

What we want A model that predicts well on **new, unseen data**. **The danger** Evaluating on the *same* data used to fit gives an **optimistically biased** estimate.



Three regimes

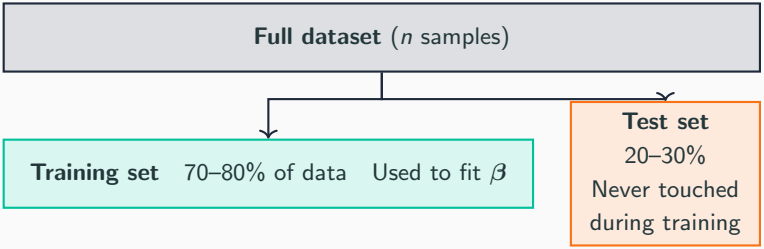


Underfitting: model too simple.

Overfitting: low training error, high test error.

Goal: minimise *test* error.

Train / Test Split

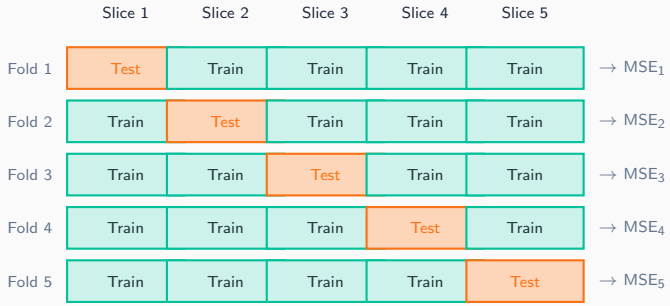


Python

```
from sklearn.model_selection import
train_test_split
X_tr, X_te, y_tr, y_te =
train_test_split(X, y, test_size=0.2,
random_state=42)
```

Key rule: The test set must **never** influence any modelling decision. Violating this is called **data leakage**.

k-Fold Cross-Validation



Algorithm

1. Shuffle and split data into k equal folds
2. For fold i: train on remaining $k-1$ folds, evaluate on fold i
3. Report $\overline{\text{MSE}} = \frac{1}{k}\sum_{i=1}^{k} \text{MSE}_i$

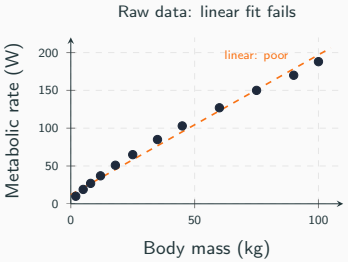
Common: $k = 5$ or $k = 10$.

```
from sklearn.model_selection
import cross_val_score

scores = cross_val_score(
model, X, y, cv=5,
scoring='neg_mean_squared_error'
)

# mean over folds
cv_mse = -scores.mean()
```

Beyond Linearity: Feature Transformations



The problem

Many real relationships are *nonlinear*. A straight line systematically misses the data.

The insight

Create **new features** by transforming x:

$$x \longrightarrow [x,\ x^2,\ x^3,\ \ldots]$$

$$x \longrightarrow [\log x,\ \sqrt{x},\ x^{0.75},\ \ldots]$$

Key message

The model stays *linear in the parameters* — the **same normal equation and gradient descent** apply unchanged.

Nonlinearity lives in the features, not the algorithm.

Polynomial Regression: It's Still Linear Regression

Fit $\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$ to n points.

Add polynomial columns to the design matrix:

$$\mathbf{X}_{\text{poly}} = \begin{pmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & x_n^3 \end{pmatrix}$$

Same normal equation as always:

$$\boldsymbol{\beta} = (\mathbf{X}_{\text{poly}}^T \mathbf{X}_{\text{poly}})^{-1} \mathbf{X}_{\text{poly}}^T \mathbf{y}$$

In Python

```
from sklearn.preprocessing
import PolynomialFeatures

poly = PolynomialFeatures(
degree=3,
include_bias=True)
X_p = poly.fit_transform(X)
# adds x^2, x^3 columns

beta = np.linalg.pinv(X_p) @ y
# identical to before
```

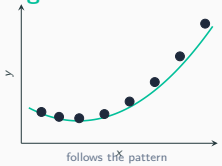
Key point: The algorithm never "knows" whether columns are x, x^2, $\log x$, or $\sqrt{x}$. It just sees numbers.

Overfitting with Polynomials

Degree 1 — Underfit

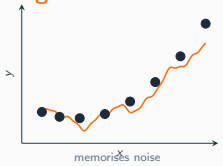
Train MSE	high
Test MSE	high

Degree 3 — Good fit



Train MSE	low
Test MSE	low

Degree 7 — Overfit

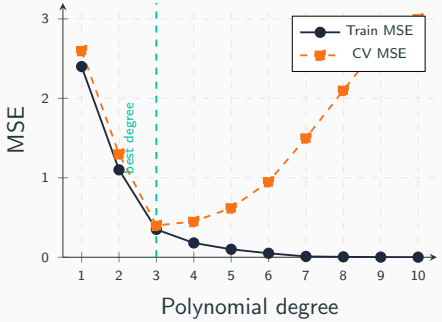


Train MSE	≈ 0
Test MSE	**very high**

Lagrange Interpolation Theorem: For any n distinct points $\exists!$ a degree $\leq n-1$ polynomial through all of them.

This is why we need cross-validation to choose the right polynomial degree.

Using Cross-Validation to Select Polynomial Degree



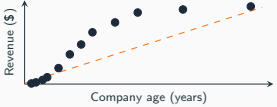
Procedure

1. For each degree $p \in \{1, \ldots, P\}$:
   - Build $\mathbf{X}_{\text{poly}}$ with p columns
   - Run k-fold CV on training set
   - Record $\overline{\text{MSE}}_p$
2. Pick $p^* = \arg\min_p \overline{\text{MSE}}_p$
3. Refit on *all* training data with p^*
4. Evaluate once on test set

```
from sklearn.pipeline import Pipeline
best_deg, best_score = 1, np.inf
for deg in range(1, 11):
pipe = Pipeline([
('poly', PolynomialFeatures(deg)),
('lr', LinearRegression())])
s = -cross_val_score(pipe,
X_tr, y_tr, cv=5,
scoring='neg_mean_squared_error').mean()
if s < best_score:
best_deg, best_score = deg, s
```

Log Transformations

Raw scale — skewed, heteroscedastic



Log scale — linear, homoscedastic

When to use log transforms

- Data spans orders of magnitude (income, population, gene expression)
- Residuals grow with fitted values (heteroscedasticity)
- Multiplicative rather than additive noise

What it buys you

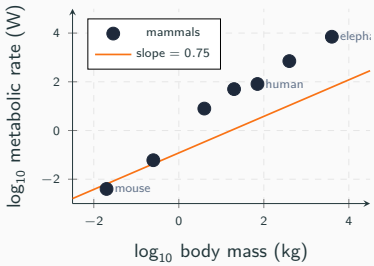
- Compresses large values, spreads small ones
- Stabilises variance across the range
- Often reveals a clean linear relationship

In Python

```
y_log = np.log(y)  # natural log
X_log = np.log(X)
# fit on log-transformed data
beta = np.linalg.pinv(X_log) @ y_log
```

Power Laws

Metabolic scaling (Kleiber's Law)



What is a power law?

$$y = a \cdot x^b$$

Taking log of both sides:

$$\log y = \log a + b \log x$$

A **linear equation** in $\log x$ and $\log y$: slope $= b$, intercept $= \log a$.

Kleiber's Law ($b \approx 3/4$):

$M \propto m^{0.75}$ across all mammals — from mice to elephants, a 7-order-of-magnitude range.

Other power laws

- Earthquake frequency vs. magnitude (Gutenberg–Richter)
- City population vs. rank (Zipf's law)
- Gene expression levels across cells

A straight line on a log-log plot is the signature of a power law.

Solving a System of Equations

The normal equation requires:

$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

This is equivalent to solving d linear systems of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} = \mathbf{X}^T\mathbf{X}$ is $d \times d$:

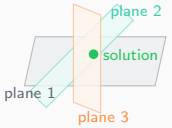
$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Three questions:

1. *When* does a unique solution exist?
2. *How* do we find it efficiently?
3. *Why* is naive inversion $\mathcal{O}(d^3)$?

The geometry

Each equation $a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 = b_i$ defines a *plane* in 3D.

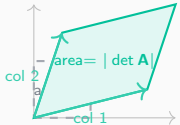


A unique solution exists when the three planes meet at **exactly one point**. Fails when planes are parallel or coincident — detected by the **determinant**.

The Determinant

Geometric meaning

$|\det(\mathbf{A})|$ = **volume scaling factor**: how much $\mathbf{A}$ stretches or shrinks space.



What $\det(\mathbf{A})$ tells you

- $\det \neq 0 \Rightarrow$ **invertible**, unique solution exists
- $\det = 0 \Rightarrow$ **singular**: no unique solution
- Sign = orientation; magnitude = volume scaling

For 2×2: $\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$

Invertible:

$$\det \begin{pmatrix} 3 & 1 \\ 2 & 4 \end{pmatrix} = 12 - 2 = 10 \neq 0 \checkmark$$

Singular:

$$\det \begin{pmatrix} 2 & 4 \\ 1 & 2 \end{pmatrix} = 4 - 4 = 0 \times$$

Row 2 is $\frac{1}{2}\times$ row 1 — linearly dependent.

The Leibniz Formula for the Determinant

$$\det(\mathbf{A}) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

S_n = all permutations of $\{1, \ldots, n\}$; $\operatorname{sgn}(\sigma) = +1$ (even # swaps) or -1 (odd # swaps).

All $3! = 6$ permutations of S_3:

σ	$\operatorname{sgn}(\sigma)$	Term	Note
$(1, 2, 3)$	$+1$	$+a_{11}a_{22}a_{33}$	identity (0 swaps)
$(1, 3, 2)$	-1	$-a_{11}a_{23}a_{32}$	swap cols 2,3
$(2, 1, 3)$	-1	$-a_{12}a_{21}a_{33}$	swap cols 1,2
$(2, 3, 1)$	$+1$	$+a_{12}a_{23}a_{31}$	cycle right (2 swaps)
$(3, 1, 2)$	$+1$	$+a_{13}a_{21}a_{32}$	cycle left (2 swaps)
$(3, 2, 1)$	-1	$-a_{13}a_{22}a_{31}$	full reversal (3 swaps)

$$\det(\mathbf{A}) = a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31}$$

For $n \times n$: $n!$ terms — $\mathcal{O}(n!)$ to evaluate. For $n = 20$: over 2 quintillion terms. Elimination gives us $\mathcal{O}(n^3)$.

Gaussian Elimination and Back Substitution

Augmented matrix $[\mathbf{A} \mid \mathbf{b}]$

Start: $\left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right)$

$R_2 \leftarrow R_2 + \frac{3}{2}R_1$; $R_3 \leftarrow R_3 + R_1$:

$\left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right)$

$R_3 \leftarrow R_3 - 4R_2$:

$\left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right)$

Back-substitute: $x_3{=}-1$, $x_2{=}4$, $x_1{=}2$.

Why this beats inverting A

- Elimination: $\mathcal{O}(d^3)$ with a *small* constant
- Full $\mathbf{A}^{-1}$: also $\mathcal{O}(d^3)$ but $\sim 3\times$ more work
- `np.linalg.solve` never forms $\mathbf{A}^{-1}$

Always prefer `solve` over `inv`

```
# DON'T:
beta = np.linalg.inv(A) @ b

# DO:
beta = np.linalg.solve(A, b)

# Normal equation:
A = X.T @ X
b_rhs = X.T @ y
beta = np.linalg.solve(A, b_rhs)
```

`solve` is faster, more stable, avoids forming the inverse.

LU Decomposition: How `numpy` Really Solves It

The key idea

Gaussian elimination implicitly factors $\mathbf{A}$:

$$\mathbf{PA} = \mathbf{LU}$$

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}$$

- $\mathbf{P}$: permutation matrix (row swaps for stability)
- $\mathbf{L}$: lower triangular (elimination multipliers)
- $\mathbf{U}$: upper triangular (row-echelon form)

Solve $\mathbf{Ax} = \mathbf{b}$ in two cheap steps:

1. Forward sub: solve $\mathbf{Lz} = \mathbf{Pb}$ $\quad \mathcal{O}(d^2)$
2. Back sub: solve $\mathbf{Ux} = \mathbf{z}$ $\quad \mathcal{O}(d^2)$

$\mathcal{O}(d^3)$ paid once for factorisation. Each new $\mathbf{b}$ costs only $\mathcal{O}(d^2)$.

In Python:

```
from scipy.linalg import lu_factor,
lu_solve

# Factorise once --- O(d^3)
lu, piv = lu_factor(A)

# Each solve --- O(d^2)
x1 = lu_solve((lu, piv), b1)
x2 = lu_solve((lu, piv), b2)
```

When does this fail?

If $\det(\mathbf{A}) = 0$, $\mathbf{U}$ has a zero on the diagonal — elimination breaks down. This is exactly the **multicollinearity** problem in regression.

Motivation: When the Normal Equation Is Too Slow

Normal equation

$$\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

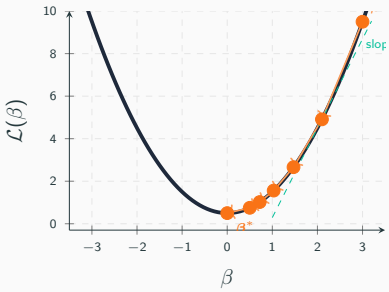
- ✓ Exact, closed-form
- ✓ No hyperparameters
- ✗ Inverting $d \times d$: $\mathcal{O}(d^3)$
- ✗ Infeasible for large d or n
- ✗ Only works for least-squares loss

Gradient descent Iteratively moves downhill on the loss surface:

$$\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} - \eta \, \nabla_{\boldsymbol{\beta}} \, \mathcal{L}(\boldsymbol{\beta}^{(t)})$$

- ✓ Scales to millions of parameters
- ✓ Works for any differentiable loss
- ✓ Foundation of deep learning
- ✗ Requires tuning η
- ✗ Approximate (iterative)

Gradient Descent: The Intuition



The update rule

$$\beta^{(t+1)} = \beta^{(t)} - \eta \cdot \left.\frac{d\mathcal{L}}{d\beta}\right|_{\beta^{(t)}}$$

Why subtract?

Gradient points *uphill*. Subtracting moves *downhill*. **Learning rate** η

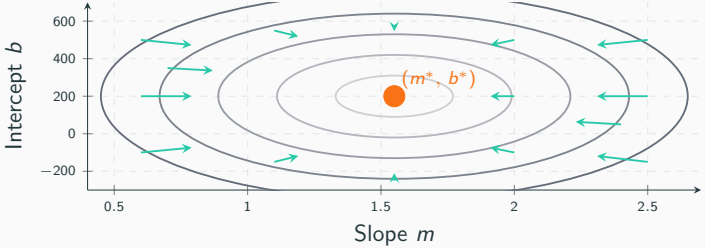
- Too large $\Rightarrow$ overshoots
- Too small $\Rightarrow$ very slow
- Just right $\Rightarrow$ smooth convergence

Typical: 10^{-4} *to* 10^{-1}.

Gradient of the SSE: Quiver Plot

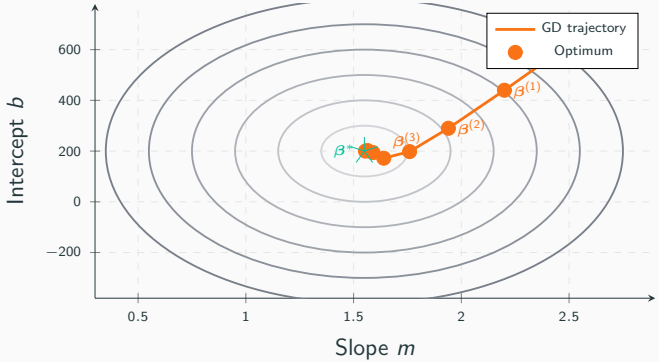
For $\mathcal{L}(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2$:

$$\nabla_{\boldsymbol{\beta}} \mathcal{L} = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \implies \boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} + 2\eta\, \mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}^{(t)})$$



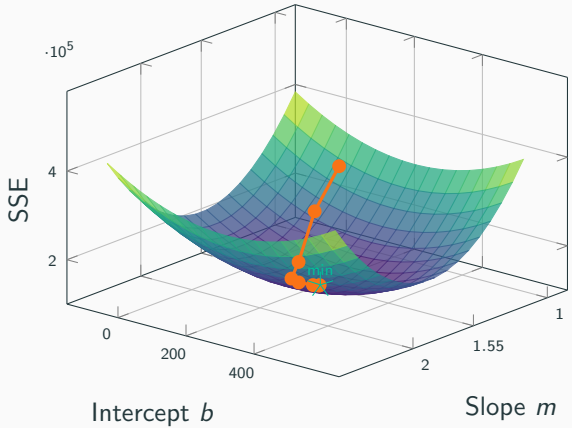
Each teal arrow shows the negative gradient direction. All arrows converge toward the minimum.

Gradient Descent Trajectory on the SSE Contour Map



The trajectory overshoots slightly before settling. Smaller η gives smoother but slower convergence.

Gradient Descent on the 3-D SSE Surface



Reading the surface

The SSE is a **paraboloid** in 2-D parameter space.

Red trace: GD rolling from $\boldsymbol{\beta}^{(0)}$ to $\boldsymbol{\beta}^*$.

Linear regression:
Convex bowl — GD finds the global minimum.

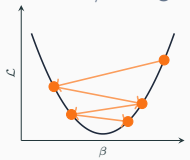
Neural networks:
Non-convex with many local minima and saddle points.

Bowl elongated along b-axis: b has less effect per unit than m.

Effect of Learning Rate η

Too large ($\eta = 0.9$)

Oscillates / diverges

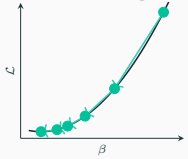


Too small ($\eta = 0.01$)

Very slow

Just right ($\eta = 0.3$)

Smooth convergence



Rule of thumb: start with $\eta = 0.01$, halve if loss oscillates, double if convergence is very slow.

Modern practice: adaptive optimisers (Adam, RMSProp) tune η per parameter automatically.

Gradient Descent for Linear Regression: Full Algorithm

Python implementation

```
# X:(n,d), y:(n,1), eta:  learning rate
# Initialise
beta = np.zeros((X.shape[1], 1))

for t in range(n_iters):
residuals = y - X @ beta
# gradient of SSE
grad = -2 * X.T @ residuals
# parameter update
beta = beta - eta * grad

# approx (X^TX)^-1 X^Ty
```

Convergence Stop when:

$$\|\nabla_\beta \mathcal{L}\| < \varepsilon$$

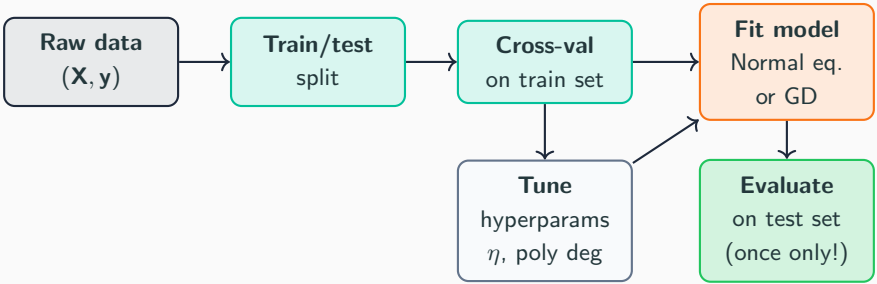
or after a fixed number of iterations.

Variants

- **Batch GD:** all n samples per step
- **SGD:** 1 sample per step — noisy but fast
- **Mini-batch:** B samples — standard in deep learning

Convex losses (SSE): all variants converge to the global minimum.
Non-convex losses (deep nets): convergence to a local min or saddle.

Putting It All Together



Key principles

- Always split *before* any preprocessing or feature engineering
- Use CV to select polynomial degree, η, and other hyperparameters
- Report test performance *exactly once*

Next lecture

- Regularisation (Ridge, Lasso) — controlling polynomial complexity
- Logistic regression: from regression to classification
- Mini-batch SGD and adaptive optimisers