

LINEAR MODELS

Complete Mathematical & Conceptual Guide

Regression · Gradient Descent · Regularization · Classification

Extra-deep focus on the math behind every equation and hyperparameter

Linear models are the foundation of modern ML. Master these, master everything that comes after.

01 Ordinary Least Squares (OLS) — The Normal Equation

The problem

You have data points $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ and you want to find a line $y = \theta_0 + \theta_1 x$ that fits best. What does 'best' mean? The line that **minimises the sum of squared errors**.

The solution: the normal equation

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

This formula computes the optimal parameters θ in one shot. But what does it actually do?

■ DEEP DIVE — Deriving the normal equation from scratch

Step 1: Set up the problem. We want to minimise the **sum of squared errors (SSE)**:

$$SSE(\theta) = \sum (y_i - (\theta_0 + \theta_1 x_i))^2 = \|X\theta - y\|^2$$

Step 2: Take the derivative with respect to θ . At the minimum, the derivative is zero:

$$d(SSE)/d\theta = -2X^T(y - X\theta) = 0$$

Step 3: Rearrange to solve for θ .

$$X^T X \theta = X^T y$$

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

That's it! No iteration, no learning rates — just linear algebra.

Computational cost

Operation	Cost
$X^T X$ (matrix multiply)	$O(n^2 m)$ where n =features, m =samples
Invert $(n \times n)$ matrix	$O(n^3)$ — the bottleneck
Total for OLS	$O(n^3 + n^2 m)$

Why OLS doesn't scale

For 10,000 features, inverting a 10,000×10,000 matrix means **10^{12} floating-point operations**. For 100,000 features: **10^{14} operations** (petascale). This is why gradient descent, which avoids matrix inversion, became the industry standard for large datasets.

But for datasets that *do* fit in memory (thousands to tens of thousands of samples, <100 features), OLS is fast, exact, and interpretable.

02 Gradient Descent — Iterative Optimization

The core idea

Instead of solving for θ directly, we start with a random guess and iteratively improve it by stepping in the direction of **decreasing cost**. This direction is the **negative gradient**.

The update rule

$$\theta := \theta - \eta \nabla J(\theta)$$

Where:

- η (eta) = learning rate, controls step size
- ∇J = gradient of the cost function — points uphill, toward worse cost
- $-\nabla J$ = direction of steepest descent

■ DEEP DIVE — What the gradient actually is

For linear regression with cost $J = \text{MSE} = (1/m) \|X\theta - y\|^2$:

$$\nabla J = (2/m) X^T (X\theta - y)$$

This vector has one entry per parameter. Each entry tells you: "**change θ by this amount to reduce cost the most.**" The larger the gradient component, the more that parameter contributed to the error.

The **negative gradient** points downhill — we move in that direction by step size η .

If a parameter has a large negative gradient, increasing it reduces cost → we subtract that gradient. If it has a positive gradient, decreasing it reduces cost → we subtract (which makes it smaller).

The learning rate (η) — a critical hyperparameter

If η is...	What happens
Too small	Slow convergence, might get stuck in local minima
Too large	Overshooting, cost oscillates or diverges (gets worse!)
Just right	Smooth, monotonic decrease toward the minimum

How to choose a learning rate

There's no universal rule — it depends on the scale of your features and the curvature of your cost function. **Standard approach:** start with $\eta=0.01$ and observe the cost curve. If it oscillates, reduce η . If it decreases smoothly but slowly, increase η . Adaptive methods like Adam automatically tune η per parameter.

For polynomial and complex problems, you may want to decay η over time: $\eta(t) = \eta / (1 + t)$ so early iterations take big steps and later iterations refine.

03 Batch vs. Stochastic Gradient Descent

Batch GD: use all m samples for each update

```

For each epoch:
grad = (2/m) *  $\sum (X\theta - y)$  # All m rows
 $\theta = \theta - \eta \cdot \text{grad}$ 

```

Pros: Stable, exact gradient, simple analysis

Cons: Slow for huge datasets, needs full matrix in memory

Stochastic GD: use one random sample at a time

```

For each epoch:
For each sample i:
grad_i = 2 *  $x_i \cdot (x_i \theta - y_i)$  # Just one row
 $\theta = \theta - \eta \cdot \text{grad}_i$ 

```

Pros: Fast per iteration, scales to huge datasets, can escape local minima

Cons: Noisy updates, cost doesn't always decrease, needs learning rate schedule

■ DEEP DIVE — Why noise in SGD is actually helpful

Because each sample gives a slightly different gradient, SGD's path is jagged — sometimes it overshoots, sometimes it undercorrects. This randomness is **actually beneficial**:

- **Escape local minima:** Batch GD can get stuck on a plateau where the gradient is zero but it's not the global minimum. SGD's noise jolts it forward.
- **Generalisation:** Updates that only fit one sample well can 'overfit' to that sample, but the next sample pulls in a different direction, forcing the model to find a compromise — the true generalisable direction.

This is why SGD, despite being noisy, often outperforms batch GD in practice on real data.

Aspect	Batch GD	Stochastic GD
Computation per update	$O(nm)$ — full matrix	$O(n)$ — one row
Gradient	Exact	Noisy but unbiased
Memory	Requires all data	One sample at a time
Convergence	Smooth monotonic decrease	Jagged, but often faster
Good for	Small-medium datasets	Large datasets, online learning

04 Polynomial Regression & The Overfitting Trap

The pitch: fit non-linear data

Real data rarely follows a straight line. Polynomial features let us fit curves: instead of $[1, x]$, use $[1, x, x^2, x^3, \dots]$. Linear regression on polynomial features = **polynomial regression**. Still linear-in-parameters!

The problem: overfitting

A degree-10 polynomial will fit training data perfectly (all 11 parameters plus intercept can match 11 training points exactly). But on new data, it's terrible — it's fitting the **noise** in the training set, not the true underlying

pattern.

■ DEEP DIVE — Bias vs. Variance vs. Irreducible Error

Bias: How wrong the model is *on average* across different datasets. A simple model (line) on curved data has high bias — it's systematically off, no matter what. A complex model (high-degree polynomial) has low bias — it can fit any curve.

Variance: How much the model's predictions change when trained on different datasets. A simple model has low variance — the line stays roughly the same. A complex model has high variance — small changes in data cause wild swings in the curve.

Irreducible error: Noise in the data itself — no model can do better than this, no matter how complex.

Total error = Bias² + Variance + Irreducible error.

The **bias-variance tradeoff**: as you increase complexity, bias ↓ but variance ↑. The sweet spot is where total error is minimised.

Learning curves reveal the problem

Plot training error and validation error (from k-fold CV) against training set size:

- If validation error stays high while training error is low → **overfitting** (high variance)
- If both training and validation error are high → **underfitting** (high bias)

Overfitting improves with more data (right side of curve). Underfitting improves by using a more complex model.

05 Regularization — Controlling Model Complexity

The idea: penalise complexity

Instead of minimising just MSE, add a penalty that grows with coefficient size:

$$J(\theta) = \text{MSE} + \alpha \times (\text{penalty on coefficients})$$

The α hyperparameter controls the tradeoff:

- $\alpha = 0$: No penalty, pure least squares (can overfit)
- α large: Heavy penalty, coefficients shrink toward zero (underfits)

L2 Regularization (Ridge)

$$J(\theta) = \text{MSE} + \alpha \times \sum \theta_j^2$$

Ridge regression penalises the **sum of squares** of coefficients. This shrinks all coefficients smoothly toward zero, but rarely to exactly zero.

■ DEEP DIVE — Why L2 shrinks smoothly

If θ_j is large, θ_j^2 is very large — the penalty blows up. To minimise J , the algorithm reduces θ_j . As θ_j approaches zero, θ_j^2 approaches zero quadratically — the marginal penalty decreases, so the algorithm stops short of zero.

Advantage: Ridge handles correlated features well — it distributes weight across correlated features rather than picking one arbitrarily.

Disadvantage: It never performs feature selection — all features stay in the model (though weakly weighted).

L1 Regularization (Lasso)

$$J(\theta) = \text{MSE} + \alpha \times \sum |\theta_j|$$

Lasso regression penalises the **sum of absolute values**. This creates a 'corner' at $\theta_j=0$, which is where the penalty often bottoms out — coefficients snap to exactly zero.

■ DEEP DIVE — Why L1 produces sparsity

The absolute value $|\theta_j|$ has a sharp gradient (slope) everywhere except at zero, where it's undefined. When the MSE gradient pushes a coefficient toward zero, the L1 penalty's sharp corner often wins, driving it all the way to zero.

This is a **geometric effect**: the optimal θ sits at the intersection of the MSE contours and the L1 penalty diamond. That intersection often lies on a corner (axis), not inside the diamond — i.e., on a zero coordinate.

Advantage: Automatic feature selection — weak features are zeroed out, simplifying the model.

Disadvantage: Unstable with correlated features — it'll arbitrarily pick one and zero the others.

ElasticNet: the best of both

$$J(\theta) = \text{MSE} + \alpha \times (\text{L1 ratio} \times \sum |\theta_j| + (1 - \text{L1 ratio}) \times \sum \theta_j^2)$$

Combines Ridge and Lasso. L1 ratio = 0 gives Ridge, 1 gives Lasso, 0.5 is balanced.

Why use ElasticNet? Correlated features + need for sparsity. L1 picks one of correlated features, L2 distributes weight — together they pick one, but with stable gradients.

06 Logistic Regression — From Regression to Classification

The problem with linear regression for classification

If you use linear regression for binary classification (0/1 targets), the fitted line will output values way above 1 and way below 0 for extreme x-values. These aren't probabilities. We need a function that always outputs between 0 and 1.

The solution: the sigmoid function

$$\sigma(z) = 1 / (1 + e^{(-z)})$$

Where $z = \theta^T x$ is the linear prediction. The sigmoid squashes any input to (0, 1):

- $z \rightarrow -\infty$: $\sigma(z) \rightarrow 0$
- $z = 0$: $\sigma(z) = 0.5$
- $z \rightarrow +\infty$: $\sigma(z) \rightarrow 1$

■ DEEP DIVE — Why the sigmoid works for probability

The sigmoid is the output of **logistic regression**, which models $P(y=1|x) = \sigma(\theta^T x)$. It interprets the linear score $\theta^T x$ as a **log-odds**:

$$\log(P(y=1)/P(y=0)) = \theta^T x$$

This equation is the **logit** — it's invertible:

$$P(y=1) = e^{(\theta^T x)} / (1 + e^{(\theta^T x)}) = \sigma(\theta^T x)$$

So the sigmoid naturally falls out of the assumption that the log-odds are linear in x . This is why it's used everywhere in classification.

The cost function: log loss (cross-entropy)

$$J(\theta) = -(1/m) \sum [y_i \log(\sigma_i) + (1-y_i) \log(1-\sigma_i)]$$

This is **binary cross-entropy**. Why not MSE (squared error)?

MSE has a **problem**: $\sigma(z)$ is S-shaped and flat at the extremes. When the prediction is very wrong ($\sigma=0.01$ but $y=1$), the MSE gradient is tiny — learning is slow. Log loss has a sharp gradient everywhere — the further from the truth, the steeper the penalty.

Decision boundary at $\sigma = 0.5$

The default decision boundary is where $\sigma(\theta^T x) = 0.5$, which happens when $\theta^T x = 0$. But you can shift this! If you want **high recall** (catch more positives), lower the threshold to 0.3. If you want **high precision** (fewer false alarms), raise it to 0.7. The sigmoid outputs and k-fold CV PR curves let you find the right cutoff for your use case.

07 Softmax — Multiclass Logistic Regression

From binary to K classes

For binary classification, we output $P(y=1|x)$ via sigmoid. For multiclass, we output a probability for *each* class. Softmax is the generalisation of sigmoid to K classes.

The softmax function

$$P(y=k|x) = e^{(\theta_k^T x)} / \sum_j e^{(\theta_j^T x)}$$

For each class k , we have a coefficient vector θ_k . Softmax computes a probability vector that sums to 1 and is all non-negative.

■ DEEP DIVE — How softmax works

Step 1: Compute K linear scores: $z_k = \theta_k^T x$

Step 2: Exponentiate to make positive: $e^{(z_k)}$

Step 3: Normalise so they sum to 1: divide by the sum

The exponential is key — it **amplifies differences**: if $z_1 = 2.0$ and $z_2 = 2.1$, then $e^{(2.0)} = 7.4$ and $e^{(2.1)} = 8.2$ (only 10% apart). But if $z_1 = 0$ and $z_2 = 0.1$, then $e^0 = 1$ and $e^{(0.1)} = 1.1$ (still 10% apart). So softmax is **temperature-dependent** — confident scores don't 'squash' like sigmoid does.

At test time, we **predict the class with highest probability** — equivalently, highest score z_k .

Cost function: categorical cross-entropy

$$J(\theta) = -(1/m) \sum_i \sum_k y_{\{i,k\}} \log(P(y=k|x_i))$$

For each sample i , only the true class k^* has $y_{\{i,k^*\}}=1$; others are 0. So the sum simplifies to just $-\log(P(y=k^*|x_i))$. The model is penalised for low probability on the true class.

08 Summary & Model Zoo

Model	Task	Link function	Cost function
Linear Regression	Continuous y	Identity	MSE
Ridge / Lasso / ElasticNet	Regression + regularization	Identity	MSE + penalty
Logistic (binary)	Binary classification	Sigmoid	Log loss
Logistic (multiclass)	Multiclass	Softmax	Cross-entropy

All use the same optimisation

- Small datasets ($n < 10,000$): Normal equation (OLS) for fast exact solution
- Medium datasets: Batch gradient descent with learning rate schedule
- Large datasets: Stochastic / mini-batch GD with decaying η
- All need regularization (Ridge/Lasso/ElasticNet) to avoid overfitting

That's linear models. Master these principles and you're ready for neural networks (which chain linear $\rightarrow$ nonlinearity), SVMs (which use kernels for implicit feature expansion), and boosting (which ensembles weak linear learners).