MY474: Applied Machine Learning for Social Science

Exam Preparation Notes: Theory and Checking code for errors

Cèlia Estruch-Garcia

Last updated: June 13, 2026

The course is essentially answering 1 question: *how do you build a model that generalises to new data?* Each week adds one piece to that puzzle.

Week 1 sets up the problem. The world generates data through some unknown process f, and your job is to approximate it with g(X). The fundamental obstacle is the bias-variance tradeoff — a flexible model fits your training data well (low bias) but wobbles wildly if you'd trained on slightly different data (high variance). You can't eliminate both at once. Everything else is a response to this tension.

Week 2 builds the first real model: logistic regression for binary classification. It introduces two key ideas. First, what does "fitting" mean? — you're finding the parameters β that make your observed data most probable (MLE), which practically means minimising the negative log-likelihood. Second, how do you actually find those parameters? Gradient descent: start somewhere, compute the slope, step downhill, repeat. The sigmoid and the gradient formula are the exam's mechanical core.

Week 3 zooms out and asks: how do we know if our model is any good? It formalises loss (per observation) vs. cost (aggregate), introduces proper evaluation metrics (accuracy, precision, recall, AUC), and — most importantly — explains why you must evaluate on held-out data. A model that memorises training data scores perfectly but fails on anything new. The train/validation/test split is the institutional answer: never let the model see test data until the very end.

Week 4 delivers the practical solution to the bias-variance problem diagnosed in W1. OLS is unbiased by construction, meaning all its error comes from variance (and you can't fix that by tweaking OLS). Regularisation intentionally introduces bias (penalising large coefficients via λR(f)) to $\downarrow$ variance more than enough to compensate. LASSO (L1, diamond constraint) goes further by 0ing out weak predictors entirely (variable selection for free). Ridge (L2, circle) shrinks everything smoothly but never to 0. Crucially, λ itself can't be found by GD (it would always collapse to 0), so you need cross-validation + hyperparameter search (grid, random, Bayesian/TPE) to choose it externally.

Week 1: Concepts in Machine Learning

Bias/variance tradeoff: as you make a model more flexible, bias $\Downarrow$ (lines better adjust points) but variance $\Uparrow$ (won't generalise to new data). The challenge lies in finding a method for which both the variance and the squared bias are low.

- **Variance** *(error across models):* amount by which $\hat{f}$ would change if we estimated it using a different training dataset. $\Rightarrow$ Ideally, $\hat{f}$ should not vary too much between training sets. More flexible methods tend to have high variance — small changes in the training data can lead to large changes in $\hat{f}$. The relative rate of change of these two quantities determines whether the test MSE increases or decreases.

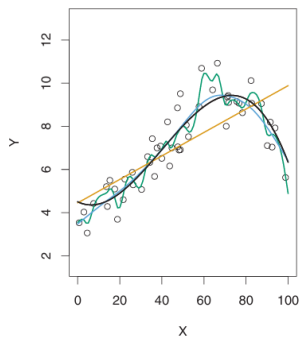


Figure 1: The green least-squares line has high variance: moving any single data point may shift $\hat{f}$ considerably. The orange line is inflexible and has low variance: moving any single observation causes only a small shift.

- **Bias** *(error within one model):* the difference between the average prediction of our model and the true value we are trying to predict. Error introduced by approximating a complex real-world function with a simpler model. For example, linear regression assumes a linear relationship; if the true f is non-linear, the model will have high bias regardless of how much data it sees. Simpler, less flexible models tend to have high bias and low variance.

The expected test error for a new observation x_0 decomposes as:

$$E\Big(y_0 - \hat{f}(x_0)\Big)^2 = \underbrace{\mathrm{Var}\Big(\hat{f}(x_0)\Big)}_{\text{variance}} + \underbrace{\Big[\mathrm{Bias}\Big(\hat{f}(x_0)\Big)\Big]^2}_{\text{squared bias}} + \underbrace{\mathrm{Var}(\epsilon)}_{\text{irreducible error}}$$

where $\mathrm{Var}(\epsilon)$ is irreducible noise in the data that cannot be reduced regardless of model choice. The goal is to minimise the total *reducible* error: $\mathrm{Var}(\hat{f}) + [\mathrm{Bias}(\hat{f})]^2$.

Generalisation: the ability of a model to perform well on data it has never seen during training. A model that scores perfectly on training data is useless if it fails on new data (overfitting). This is why we always hold out a **test set**: data the model never trains on, used only to evaluate final performance.

Cross-validation: a technique to optimise the model using only training data, keeping the test set genuinely untouched. Ex. k-fold cross-validation (Week 3).

Regularisation: a technique that adds a **penalty** to the training loss to discourage the model from assigning large weights to features, forcing it to be simpler and reducing overfitting. (Week 4: L2/Ridge and L1/LASSO)

Feature engineering: the process of transforming, combining, or creating variables to make the signal clearer for the model. Raw data is rarely in the right form. Examples include: taking the log of a skewed variable, encoding a date as "day of week"... (W7).

Model selection: the process of choosing between candidate models (e.g. a decision tree vs. LASSO vs. KNN) in a principled way. You cannot simply pick the model with the best training accuracy (overfitting risk). Instead, cross-validation scores on held-out folds are used to compare models fairly before any test data is touched.

Motivation: Why Machine Learning?

Breiman (2001) identified two modelling cultures:

1. **Deductive/stochastic data model:** "I have a theory about X; I will test it with data."
2. **Inductive/algorithmic model:** 'I have data; what can it tell me about X?' learn from data.

ML largely falls into culture 2 → lets data drive model structure + uses (iterative) algorithms to fine-tune model parameters.

Key Terminology

Term	Meaning
Scalars — lowercase, unbolded	
θ, x	A single number, e.g. $\theta = 3.141$, $x = 1$
Vectors — lowercase, bold	
$\boldsymbol{\theta}$, $\boldsymbol{x}$	An ordered list of scalar values, e.g. $\boldsymbol{\theta} = [0.5, 3, 2]$. They do not carry dimensionality.

Term	Meaning
Matrices — uppercase, bold	
$\boldsymbol{\Theta}$, $\mathbf{X}$	A 2-D array of values, e.g. $\mathbf{X} = \begin{bmatrix} 1 & 5 \\ 24 & -3 \end{bmatrix}$
Indexing	
$\mathbf{x}^{(k)}$	The k-th column of $\mathbf{X}$ (superscript in brackets ⇒ column)
$\boldsymbol{x}_i$	The i-th row of $\mathbf{X}$ (subscript ⇒ row)
$x_i^{(k)}$	The k-th element of the row vector $\boldsymbol{x}_i$
Outcome and predictions	
$\mathbf{X}$	$n \times k$: n rows/observations, k columns/features/predictors
$\boldsymbol{y}$	$n \times 1$ matrix: labels/outputs/responses/outcomes
$\hat{\boldsymbol{y}}$	Predicted values: $\hat{\boldsymbol{y}} \sim g(\mathbf{X})$
Data splits	
Training data	Labelled data used to fit the model
Test data	Held-out data used only to evaluate final performance
3 related concepts	
Model	General structural form; e.g. $y = \alpha + \beta X$. Defines the *type* of relationship.
Hypothesis	Specific parameter combination drawn from **hypothesis set** $\mathcal{H}$: all possible combinations under that model (for linear: $\mathcal{H}_{\text{Linear}} = \{[\alpha, \beta]\}$, ex. $h : \{\alpha = -7.3, \beta = 0.8\}$)
Training algorithm	Procedure that searches $\mathcal{H}$ for best hypothesis, defined by loss function (e.g. minimum sum of squared residuals in linear regression; iterative methods in other models).

Supervised vs Unsupervised Learning

SUPERVISED: relies on 2 components: $\mathbf{X}$ (set of examples / ind. vars) and $\boldsymbol{y}$ (corresponding set of labels/outcomes, not only uni-dimensional).

- **Goal:** learn the mapping $\mathbf{X} \to \boldsymbol{y}$ to predict y_{n+1} for new observations (predict new cases without observing them)
- **More formally:** There is some unknown real-world data-generating process (f^{World}). Using $\mathbf{X}$, we build an approximation:

$$\boldsymbol{y} \sim f^{\text{World}}(\cdot) \quad \Rightarrow \quad \hat{\boldsymbol{y}} \sim g(\mathbf{X})$$

  We assess quality by comparing $\boldsymbol{y}$ to $\hat{\boldsymbol{y}}$. The goal is to make $g(\cdot) \approx f^{\text{World}}(\cdot)$ as closely as possible. $g(\mathbf{X})$ can take any functional form — OLS, decision tree, neural network, etc.
- **As regression:** $g(\mathbf{X}) = \beta_0 + \beta_1 \mathbf{x}^{(1)} + \cdots + \beta_k \mathbf{x}^{(k)}$. We find parameters $\boldsymbol{\beta}$ that minimise the sum of squared residuals:

$$\sum_{i=1}^{n} \big(y_i - g(\boldsymbol{x}_i)\big)^2$$

Within supervised learning, the nature of $\boldsymbol{y}$ determines whether the task is **regression** or **classification**:

Type	y is...	Examples
Regression	**Continuous**/quantitative (outcome=number on a scale)	GDP, vote share, age, time spent on site
Classification*	**Categorical** from a finite set (outcome=label)	Gender, partisan ID, click/no click an ad

**In classification tasks*, the objective is to identify the most likely *class* or *label* for each observation (membership in one of a fixed set of categories).

UNSUPERVISED: only $\mathbf{X}$ is supplied—not always $\boldsymbol{y}$. The objective is to find patterns or structure in the data. Tasks:

Type	Description
Clustering	Assign observations to groups/labels—but unlike classification, these labels (Cluster A, B, ...) are *not* meaningful nor based on any pre-established set. They simply reflect discovered structure in $\mathbf{X}$.
Dimensionality reduction	Reduce the number of dimensions in $\mathbf{X}$. E.g. compress individuals' responses to 20 policy questions into a meaningful 2-D ideological space.

Example: in topic modelling, documents are represented as word-count vectors ($\mathbf{X}$) with no outcome $\boldsymbol{y}$. The goal is to cluster documents by topic (politics, finance, etc.) using patterns in word use—the topic labels are never given in advance.

Parametric vs. Non-Parametric Models

Finally, we must decide what form we give to the model $g(\mathbf{X})$: **parametric** models fix the structure before seeing the data (simpler, more interpretable), while **non-parametric** models let the data determine the structure (+ flexible, but slower and + prone to overfitting).

	Parametric	Non-parametric
Definition	Functional form fixed before data	Structure determined from data
Examples	OLS, logistic regression, LASSO	Non-linear, interaction terms, decision trees, KNN
Pros	Simpler, faster, more interpretable	Flexible, potentially higher accuracy
Cons	May oversimplify f^{World}	Slower, prone to overfitting, less interpretable

Flexibility can come at the expense of interpretability.

The divide between models is not always clean:

- **LASSO**: specifies a regression form (parametric) but can shrink coefficients to exactly zero — partially data-driven
- **Neural networks**: have a defined parameter set but learn highly complex functions

Exam alert. This distinction is tested directly. The correct answer: *parametric models specify the functional form in before seeing the data; non-parametric models allow structure to be determined from the data.*

Common wrong answers to avoid:

- "Non-parametric models have no parameters" — **False.** They often have *many* parameters; the key is that their nature is not fixed before seeing the data.
- "Non-parametric always outperforms parametric" — **False.**
- "Parametric = regression only" — **False.** Both cover regression & classification.

Sample Question 5 — MSE calculation. Given $\boldsymbol{y} = [3, 5, 8, 9, 1]^\top$ and $\hat{\boldsymbol{y}} = [2, 6, 6, 8, 1]^\top$:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1^2 + (-1)^2 + 2^2 + 1^2 + 0^2}{5} = \frac{1+1+4+1+0}{5} = \frac{7}{5} = \mathbf{1.4}$$

Common errors:

- Taking absolute values instead of squaring ⇒ MAE, not MSE
- Taking the square root of the result ⇒ RMSE ($\sqrt{1.4} \approx 1.183$)
- Dividing by $n - 1$ instead of n ⇒ sample variance convention ($7/4 = 1.75$)

Week 2: Prediction vs. Explanation, Binary Classification, Gradient Descent

So far we have seen *what* ML is and *what types* of problems exist. Now we need to understand *why* we use models. With the same logistic regression, we can do two very different things: explain *why* someone is a Democrat (by looking at the $\hat{\boldsymbol{\beta}}$, the coefficients) or predict *whether* someone is a Democrat (by looking at $\hat{\boldsymbol{y}}$, the prediction). Explanation seeks the truth about the world; prediction seeks to get new cases right. Topic 2 builds the tools for making binary predictions: first the logistic model (how to transform a straight line into a probability between 0 and 1), then how to measure whether predictions are good (MLE and NLL), and finally how to find the best parameters iteratively (gradient descent).

Explain vs. Predict

Both tasks can use the **same model** (e.g. logistic regr), but focus is fundamentally different:

	Explanation ($\hat{\beta}$)	**Prediction ($\hat{y}$)**
Goal	Test a pre-stated theory	Predict future outcomes given past observations
Evaluate by	Inference: could this hold in the population?	Performance on "unseen" examples
Favours	Parsimony; seek the "true" model	Fidelity; seek good predictions
RHS/LHS?	Cares about $\hat{\boldsymbol{\beta}}$ (right-hand side)	Cares about $\hat{\boldsymbol{y}}$ (left-hand side)

Note: even explanatory models are typically associational, not causal. Establishing causality requires specific designs (experiments, RDD, etc.) covered in other courses.

In equation $\hat{\boldsymbol{y}} = \alpha + \hat{\boldsymbol{\beta}}\mathbf{X}$, social-science explanation focuses on $\hat{\boldsymbol{\beta}}$ (what drives the outcome?), while ML prediction focuses on $\hat{\boldsymbol{y}}$ (how accurately can we label new cases?)→ Prediction & explanation are **complements**, not substitutes.

Why prediction is useful in social science:

- Useful in its own right (e.g. forecasting economic outcomes, quantifying risk)
- Fills in missing data (e.g. inferring online demographics)
- Aids causal inference: e.g. in matching, we use ML to predict the *propensity score* (probability of being treated) — we just want accurate predictions, not an interpretable model
- Predictive models can provide benchmarks for causal theories

Logistic regression is both. The same fitted model can report $\hat{\boldsymbol{\beta}}$ for explanation ('ideology is strongest predictor of party membership') and generate $\hat{\boldsymbol{y}}$ for prediction ('this individual has a 99.6% chance of being a Democrat'). The distinction is in the *purpose*, not the model.

The Logistic Model for Binary Classification

When $y_i \in \{0, 1\}$ (e.g. Democrat vs. Republican), linear regression is inappropriate because it can produce predictions outside $[0, 1]$. Logistic regression solves this with two components:

1. **A linear model space** that spans any real value: $\alpha + \boldsymbol{\beta}\mathbf{X} \in (-\infty, +\infty)$
2. **An inverse-link function** (sigmoid) that maps this back to $[0, 1]$

We model the *log-odds* (logit) of the outcome:

$$\text{logit}(\boldsymbol{y}) = \log \frac{P(\boldsymbol{y} = 1)}{1 - P(\boldsymbol{y} = 1)} = \alpha + \boldsymbol{\beta}\mathbf{X}$$

Fitted logit values for observation i: $\text{logit}(\hat{y}_i) = \hat{\alpha} + \hat{\boldsymbol{\beta}}\boldsymbol{x}_i$

Probabilities are recovered using the **sigmoid function** $S(x) = \frac{1}{1+e^{-x}}$, applied to the fitted log-odds. To see where this comes from, rearrange the logit:

$$\begin{aligned}
\log \frac{p}{1-p} &= \alpha + \boldsymbol{\beta}\mathbf{X} \\
\log\left(\frac{1}{p} - 1\right) &= -(\alpha + \boldsymbol{\beta}\mathbf{X}) \quad \text{(negate both sides, since } \tfrac{1-p}{p} = \tfrac{1}{p} - 1\text{)} \\
\frac{1}{p} &= 1 + e^{-(\alpha+\boldsymbol{\beta}\mathbf{X})} \quad \text{(exponentiate both sides)} \\
p = P(y_i = 1) &= \frac{1}{1 + e^{-(\hat{\alpha}+\hat{\boldsymbol{\beta}}\boldsymbol{x}_i)}}
\end{aligned}$$

Exam — computing probability from logistic regression. Given $\mathbf{x}_i = [x_1, x_2, x_3]$ and $\boldsymbol{\beta} = [b_1, b_2, b_3]$ (with $a = 0$):

1. Compute log-odds: Log-Odds $= b_1x_1 + b_2x_2 + b_3x_3$
2. Apply sigmoid: $p(Y) = \frac{1}{1+e^{-\text{Log-Odds}}}$

Example: $\mathbf{x} = [1, 2, 3]$, $\boldsymbol{\beta} = [1, -2, 3]$: Log-Odds $= 1 - 4 + 9 = 6$, so $p = 1/(1 + e^{-6}) = 0.998$.

```
def sigmoid(z):
z = np.asarray(z)
z = np.clip(z, -35, 35)
return 1 / (1 + np.exp(-z))
```

Training a binary classification model - Maximum Likelihood Estimation

Goal: find β^* (best hypothesis) that makes our data X most probable to be observed → Called **likelihood** of the data, which we aim to maximise:

$$\mathcal{L}(\beta^*) = \prod_{i=1}^{n} p(\boldsymbol{x}_i \mid \beta^*)$$

This is the **joint probability** of the observed data conditional on a set of parameter values $(x_1, x_2, \ldots)$. Given β^*, we want predicted probabilities close to 1 for Democrats ($y_i = 1$) and close to 0 for Republicans ($y_i = 0$).

But there's a complication: $p(\boldsymbol{x}_i \mid \beta^*)$ always returns the probability of being a Democrat — so multiplying all predictions together penalises correct Republican predictions (low p drags the product toward 0). We need a formula that uses p when $y_i = 1$ and $1 - p$ when $y_i = 0$, handling each case separately:

$$\textit{Logistic likelihood:} \quad \mathcal{L}_{\beta,i} = p(\boldsymbol{x}_i \mid \beta^*)^{y_i} \times \big(1 - p(\boldsymbol{x}_i \mid \beta^*)\big)^{(1-y_i)}$$

When $y_i = 1$: the exponent on the second term is 0, so it turns to 1 — leaving just $p(\boldsymbol{x}_i \mid \beta^*)$, the probability of being a Democrat.
When $y_i = 0$: the exponent on the first term is 0, so *that* turns to 1 — leaving just $1 - p(\boldsymbol{x}_i \mid \beta^*)$, the probability of being a Republican.
In both cases, 1 term "switches off". Simplifying (since $p(\boldsymbol{x}_i \mid \beta^*) = \hat{y}_i$) → $\mathcal{L}_{\beta^*,i} = \hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i}$

The full MLE optimisation problem — from the hypothesis set Θ, find the parameter values that maximise the likelihood function:

$$\arg\max_{\beta \in \Theta} \; \mathcal{L}(\beta) = \prod_{i=1}^{N} \hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i}$$

Numeric overflow and the log-likelihood

- **Problem:** each $p(\boldsymbol{x}_i \mid \beta^*)$ is a small number between 0 and 1. Multiplying many of them together produces a result so small the computer can no longer store it precisely.
- **Solution1:** take the log. Since log is strictly increasing, maximising $\log \mathcal{L}$ is equivalent to maximising $\mathcal{L}$. The product becomes a sum: $\log(a \times b) = \log a + \log b$, so no underflow.
- **Solution2:** minimise the **negative log-likelihood** (NLL): multiply by -1 to turn the maximisation into a minimisation problem (convention only — both are equivalent):

$$\arg\min_{\beta \in \Theta} \; \mathcal{L}(\beta) = -\sum_{i=1}^{N} \log\Big(\hat{y}_i^{y_i} \times (1-\hat{y}_i)^{1-y_i}\Big) = -\sum_{i=1}^{N} \big[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)\big]$$

Exam — why log-likelihood, not likelihood? The primary reason is **(a): the log turns the product into a sum**, which is far easier for computers to handle (no numeric underflow from multiplying many small probabilities together).

The negative log-likelihood (NLL) is a *separate* step: we negate it to turn maximisation into minimisation — that is a convention, not the original motivation for taking the log.

So: log → summation (numerical stability). Negative → minimisation (convention).

Gradient Descent

Intuition: imagine you have a loss function (a measure of how poorly the model is doing). If we were to draw it, it would look like a parabola with a minimum point, which is where we want to get to. The problem is that we do not know where this minimum is. The only thing we can do is look at where we are now and ask: does the curve here go up or down? This is the **gradient** — the slope of the curve at that point. If it goes upward to the right, we move to the left, and vice versa: always in the **opposite** direction to the slope. We repeat this many times and, little by little, we get closer to the minimum.

Why not solve analytically? With linear regression, there is a direct formula to find the best parameters (OLS uses linear algebra). With logistic regression there is not — the sigmoid makes the problem non-linear, so no closed-form solution exists. We need an **iterative algorithm**: find β^* step by step.

The algorithm (4 steps): imagine you are on a mountain blindfolded trying to reach the lowest point. You feel the slope, take a step downhill, and repeat. Here: the "mountain" is the loss function $Q(\beta)$, the "slope" is the gradient, and the "steps" are the updates to β.

1. Choose a starting value for β (e.g. $\beta = 0$) — we are not yet at the minimum
2. Calculate the **gradient** at that point: steepness and direction of the curve at β
3. Adjust β in the **opposite direction** to the sign of the gradient (opposite sign always)
4. Recalculate the gradient and repeat from step 2

Generalising to all k parameters: adjust each β_k by subtracting its corresponding gradient element:

$$\underbrace{\beta_k}_{\text{new value}} = \underbrace{\beta_k}_{\text{current value}} - \underbrace{\frac{\partial Q(\boldsymbol{\beta})}{\partial \beta_k}}_{\text{gradient}}$$

Logistic regression gradient: to perform the update $\beta_k \leftarrow \beta_k - \frac{\partial Q}{\partial \beta_k}$, we need the partial derivative. The larger the gap between prediction and reality, the larger it is (bigger step).

Exam — logistic regression gradient. the partial derivative used in the gradient descent update has a surprisingly simple form (EXAM):

$$\frac{\partial Q^{\text{Logit}}}{\partial \beta_k} = \underbrace{\big(g(\mathbf{X}) - \boldsymbol{y}\big)}_{\text{prediction}-\text{observed}} \times \underbrace{\boldsymbol{x}^{(k)}}_{k\text{-th feature dimension}}$$

The full gradient vector ∇ stacks one such derivative per parameter:

$$\nabla = \begin{bmatrix} \partial Q^{\text{Logit}}(\boldsymbol{\theta})/\partial \beta_1 \\ \partial Q^{\text{Logit}}(\boldsymbol{\theta})/\partial \beta_2 \\ \vdots \\ \partial Q^{\text{Logit}}(\boldsymbol{\theta})/\partial \beta_k \end{bmatrix}$$

Learning rate: the gradient tells you which direction to step, but not how big the step should be. Too big and you jump over the minimum; too small and you barely move (or get stuck in local minima). We control step size by multiplying the gradient by λ:

$$\underbrace{\theta_{\text{new}}}_{\text{new value}} = \underbrace{\theta}_{\text{current value}} - \underbrace{\lambda}_{\text{learning rate}} \underbrace{\nabla}_{\text{gradient}}$$

λ is a **hyperparameter** chosen by the researcher:

- **Too large:** you keep jumping over the bottom of the valley — the algorithm never settles
- **Too small:** you inch forward so slowly you may never reach the global minimum

Gradient descent is an *optimiser*. It is not the cost function (that is the NLL), not the objective function (same as cost), and not a regulariser (L1/L2 penalties are regularisers). It is the *algorithm that searches for the minimum* of the loss.

Biggest problem for gradient descent in logistic regression? Answer: (a) an overly large learning rate. A large λ causes parameter updates to overshoot the minimum, potentially diverging entirely. Since the logistic loss surface is convex (one global minimum), this is the primary failure mode.

Why the others are wrong:

- (b) Small λ: slow but still converges — inefficient, not catastrophic
- (c) Categorical features: need encoding, but do not break gradient descent
- (d) Regularisation: changes the objective but does not prevent convergence
- (e) Large sample size: increases computation time but improves gradient estimates

Stochastic gradient descent (SGD): with standard GD, you look at *every* observation before taking a single step — precise, but very slow on large datasets. SGD instead takes a step after *each* observation: faster (N updates per pass through the data), but noisier since each step uses less information. Observations are processed in a **random order** each epoch — this ensures no part of the dataset systematically influences the updates more than another.

Method	**Update rule**
Gradient descent	$\beta_k \leftarrow \beta_k - \lambda \sum_{i=1}^{n} (g_{\beta_k}(\boldsymbol{x}_i) - y_i)\boldsymbol{x}_i^{(k)}$
SGD	$\beta_k \leftarrow \beta_k - \lambda (g_{\beta_k}(\boldsymbol{x}_i) - y_i)\boldsymbol{x}_i^{(k)}, \quad \text{for } i \in \{1, \ldots, N\}$
	`coefs = coefs - l_rate * (yhat_i - y[i]) * row_vec`

Punchline: (S)GD is not specific to logistic regression. Any model where you can define a loss function and compute a gradient can be trained this way — LASSO, neural networks, and more. This is why gradient descent is one of the most important ideas in all of ML.

SGD update. Given $x = 7$, $y = 14$, $\beta = 7.3$ (linear regression, since y is continuous).

SGD update rule: $\beta \leftarrow \beta - \lambda(g(x_i) - y_i) \cdot x_i$

$$\begin{aligned} \text{Prediction:} \quad & g(x) = \beta \cdot x = 7.3 \times 7 = 51.1 \\ \text{Error:} \quad & g(x) - y = 51.1 - 14 = 37.1 \\ \text{Gradient:} \quad & (g(x) - y) \cdot x = 37.1 \times 7 = 259.7 \\ \text{Update:} \quad & \beta_{\text{new}} = 7.3 - \lambda \times 259.7 \end{aligned}$$

The final value requires λ. E.g. with $\lambda = 0.01$: $\beta_{\text{new}} = 7.3 - 2.597 = \mathbf{4.703}$.

Seminar 1: Gradient Descent (Algorithm from Scratch)

Builds logistic regression with SGD from scratch, confirming that the update rule from the notes works in practice. The key result: after 50 epochs starting from $\beta = [0, 0, 0]$, the algorithm recovers coefficients very close to the true $[3, 1, -2]$.

Standard sklearn pipeline (used throughout the course):

```
model = LogisticRegression()    # 1. create
model.fit(X_train, y_train)     # 2. fit on TRAIN only
model.predict(X_test)           # 3. predict on TEST
accuracy_score(y_test, y_pred)  # 4. evaluate: test labels vs test predictions
```

Exam: spot errors in code. (conceptual, not syntactic). Trace these 3 checks in order:

1. What data is `.fit()` called on? → must be **train only**
2. What data is `.predict()` called on? → must be **test**
3. What labels are used in the evaluation (accuracy) function? → must be **the labels for the same observations you predicted on**

Data leakage (most common error):

```
scaler.fit_transform(X_test)   # WRONG: refits scaler on test data
scaler.fit_transform(X_train)  # RIGHT: fit on train...
scaler.transform(X_test)       #        ...only transform test
```

Rule: anything with `.fit` should only ever see training data.

Train/test mixup:

```
model.fit(X_test, y_test)       # WRONG: training on test data
model.predict(X_train)          # WRONG: predicting on train
accuracy_score(y_train, y_pred) # WRONG: comparing against wrong labels
```

3 lines of a correct pipeline: fit → train, predict → test, evaluate → test labels vs test predictions.

Week 3: Loss, evaluation metrics, test-train split

We generalize the concept of **loss** that we saw in W2 in the context of logistic regression. Now we turn it into a general **framework** for any model. We need the loss for two things: (1) to train the model (to guide gradient descent), and (2) to *interpret* how well it works.

The function we use for training does not have to be the same as the one we use for evaluation. Furthermore, the big problem in ML is that if we always evaluate on the same data we trained on, the model memorizes instead of learning. This is why we split the data into training/validation/test.

Loss — Revisited as a General Framework

How do we assess/compare model performance (once we've found the best hypothesis)?
How good is our model on $\mathbf{X}'$ *not used to train the data?*

Any model has a loss function $\mathcal{L}(\hat{y}_i, y_i)$ **which:**

1. Measures how good the regression/classification is for a given example ($\mathbf{x}$)
2. Assigns some penalty for deviations from the "true" value/label (y_i)

When $y_i = \hat{y}_i$, the loss $= 0$. Otherwise $\mathcal{L} > 0$.

Two reasons we need loss:

1. **To fit the model:** guide gradient descent through the hypothesis set. Requires a *differentiable* function.
2. **To interpret performance:** once the β^* is found, how well does it actually do?

Key insight: these two uses do not have to use the same function. You might train with BCE (differentiable, smooth) and report with accuracy (interpretable). The loss used to fit and the loss used to evaluate are separable choices.

Loss vs. Cost:

- **Loss** $\mathcal{L}(\hat{y}_i, y_i)$: difference between prediction and actually observed data; computed *per observation*. Quantifies the error for a single example.
- **Cost** $Q(\boldsymbol{\beta})$: the *aggregate* loss over the entire dataset, e.g. $\sum_{i=1}^{N} \mathcal{L}(\hat{y}_i, y_i)$ or its mean. This is the curve we minimise with gradient descent.

When $y_i = \hat{y}_i$, loss $\mathcal{L}(\hat{y}_i, y_i) = 0$. Otherwise, $\mathcal{L} > 0$, and the more wrong, the bigger the penalty.

A Naive Classification Loss and Why it Fails

The simplest approach: count the number of incorrect classifications:

$$\mathcal{L}(\hat{y}, y) = \sum_{i=1}^{N} \mathbf{1}(\hat{y}_i \neq y_i)$$

Why this is naive:

1. **Ignores how "close" the classifier got.** A predicted probability of 0.49 vs 0.51 changes the loss massively (0 vs 1), but the model is barely different. A 0.95 prediction on a true positive is penalised identically to a 0.51. The distance to the truth is invisible.
2. **Treats all errors as equal.** sometimes, false negatives are far worse than false positives (e.g. missing a cancer diagnosis vs. a spam email). This loss function has no mechanism to weight them differently.

Binary Cross-Entropy (BCE)

Solves problem (1) above, but not (2): to penalise false negatives more than false positives, add a weight w in front of each term.

Since most classifiers output a *probability of the positive class*, BCE assesses how close that probability is to the true label:

$$\mathcal{L}_{\text{BCE}} = -\Bigg[\underbrace{y_i}_{\text{true label}} \times \underbrace{\log(g(\mathbf{x}_i))}_{\text{log of predicted prob.}} + \underbrace{(1 - y_i)}_{\text{"punishment" side}} \times \log(1 - g(\mathbf{x}_i)) \Bigg]$$

The closer to the true value, the smaller each component will be. The logic-gate mechanism works just like in MLE (Week 2):

- $y_i = 1$: right term $= 0$, left survives $\rightarrow -\log(\hat{p})$. Closer to 1 $\Rightarrow$ smaller loss.
- $y_i = 0$: left term $= 0$, right survives $\rightarrow -\log(1-\hat{p})$. Closer to 0 $\Rightarrow$ smaller loss. Conditional on being class 0, not 1.

Loss Functions: A Taxonomy

Loss is separable from the model. In conventional statistics the 2 are bundled together (OLS = linear model + L2 loss). In ML we distinguish them: the same model can be optimised with different loss functions, yielding different estimators. You can even **chain** loss functions: e.g. a standard fit loss plus a penalty term ⇒ this is how regularisation works (W4).

Loss Function	**Task**	$\mathcal{L}(y, g(\mathbf{x}))$
ℓ_1 (absolute)	Regression	$\lvert y_i - g(\mathbf{x}_i)\rvert$
ℓ_2 (quadratic)	Regression	$(y_i - g(\mathbf{x}_i))^2$
0–1 classification loss	Classification	$\mathbb{I}[g(\mathbf{x}_i) \neq y_i] = 1$ if error, $= 0$ otherwise
Binary cross-entropy	Classification	$-[y_i \log(g(\mathbf{x}_i)) + (1 - y_i) \log(1 - g(\mathbf{x}_i))]$
Cross-entropy (multi-class)	Classification	$-\sum_{c=1}^{M} y_{i,c} \log(g(\mathbf{x}_i)_c)$ summation over all classes; BCE is M=2 special case

Note on ℓ_2 vs MSE: ℓ_2 is the *squared error* at the observation level. MSE is the *mean* of those squared errors across the dataset — i.e. the cost.

Choosing the right loss function. Context: labelling 1M individuals into 5 employment categories (full-time, self-employed, part-time, unemployed, not seeking). The correct loss function is **cross-entropy** (multiclass), not binary cross-entropy (only 2 classes) and not L2-loss (regression, not classification).

Why not hinge loss? Hinge loss assigns $\{-1, +1\}$ labels in a binary SVM context. It can be extended to multiclass, but this requires a more complex formulation — cross-entropy is the simpler and more natural choice here.

Key rule: **binary** outcome → BCE; $M > 2$ **categories** → cross-entropy (BCE is just the $M = 2$ special case).

Model Fit Metrics

We now ask: once the model is trained, how do we *report* its quality?

With a continuous y → RMSE (average difference between predicted and actual value):

$$\text{RMSE}(\mathbf{y}, \hat{\mathbf{y}}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

Interpretable in the *units of the outcome*. Lower is better. Still a cost metric, not a loss.

Classification: Accuracy: out of all predictions, what fraction did you get right? So if you have 100 people and your model correctly classifies 75, accuracy = 75%.

$$\text{Accuracy}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}(y_i = \hat{y}_i)$$

Useful and interpretable ("75% accurate"). **But requires balanced classes.** With imbalanced classes it is dangerously misleading:

Imbalanced class example (Adult dataset): in the 1994 US Census, 76% earn ≤50K (label = 0) and 24% above (label = 1).

Imagine the laziest possible model: it ignores all features (age, education, etc.) and predicts 0 for everyone. → It gets all 76 low earners right, and the 24 high earners wrong: Accuracy = 76/100 = 76

Looks decent! But the model has learned nothing and never identifies a high earner. This is the **majority class baseline**: the score you get by always predicting the most common class. *Always check your model's accuracy beats it by a meaningful margin.* ⇒ This is why we need metrics (below) that evaluate performance within each class separately.

Confusion Matrix: summary of predictions vs. actual outcomes:

	Actual Positive	**Actual Negative**
Predicted Positive	True Positive (TP)	False Positive (FP)
Predicted Negative	False Negative (FN)	True Negative (TN)

Performance metrics from confusion matrix (not memorize). Which one use depends on context:

Metric	**Formula**	**Intuition**
Recall / TPR / Sensitivity	$\frac{TP}{TP+FN}$	Of all actual positives, how many did you catch?
Precision	$\frac{TP}{TP+FP}$	Of all predicted positives, how many were actually positive?
False Positive Rate (FPR)	$\frac{FP}{FP+TN}$	Of all actual negatives, how many did you incorrectly flag?
True Negative Rate (TNR) / Specificity	$\frac{TN}{TN+FP}$	Of all actual negatives, how many did you correctly identify?

Confusion matrix gives you TP, FP, FN, TN for 1 **specific threshold** (e.g. predict positive if $\hat{p} > 0.5$. From those counts you derive TPR, Precision etc. → but these only tell you how the model performs at that one cut-off.

Receiver Operator Curve: ROC curve asks: what if I moved that threshold? For every possible threshold (0.01, 0.02, ... 0.99), compute the TPR and FPR — then plot them against each other. So the ROC curve is essentially the confusion matrix metrics (specifically TPR vs FPR) computed at every possible threshold simultaneously → it visualises model performance across the full range.

Area Under the (RO)Curve: AUC then collapses the entire ROC curve into one number — how much area is under it. A random model splits positives and negatives equally at every threshold, so its curve follows the diagonal → AUC = 0.5.

- AUC = 1: perfect model
- AUC = 0.5: random chance (model follows the diagonal)
- AUC = 0: perfectly wrong

Each point on the ROC curve corresponds to the TPR/FPR values from the confusion matrix at one specific threshold. The AUC summarises the whole curve in a single number: how likely is it that the model scores a random positive higher than a random negative?

Overfitting and the Train/Test Split

Overfitting: the model has learned the training data *too well* — including its noise. Low training error but fails to generalise to out-of-sample data (in regression terms: the model has too many parameters).

Underfitting: opposite—the model is too simple for the true data-generating process (ex. not enough parameters). Higher training error *and* poor out-of-sample performance.

Implication: training error alone is not a reliable judge of model quality ⇒ This is why we always evaluate on held-out data (below).

To detect overfitting, we hold out data the model never trains on, and ensuring test splits are representative of the data (achieved by random sampling):

Split	Purpose
Training	Fit the model (gradient descent sees only this data) [70-80%]
Validation	Mid-stage tuning; check performance and adjust hyperparameters without touching the test set
Test	Final evaluation of the best model. Used only *once*, at the very end.

Time series warning: for temporal data, no random sampling → Split chronologically (train on past, test on future). Otherwise you leak future information into training.
Related: **concept drift** — the relationship between variables may change over time (e.g. pre- vs. post-2020 data), causing the model to degrade out-of-sample.

Bias–Variance Trade-off

Recall from W1: as model complexity increases, bias ↓ but variance ↑. The MSE decomposes as $\text{Var}(\hat{f}) + \text{Bias}(\hat{f})^2 + \text{Var}(\varepsilon)$, where only the first two terms are reducible.
Here we ask: *what does this mean concretely for OLS and prediction?*

Bias (of predictions): systematic difference between average prediction and truth.

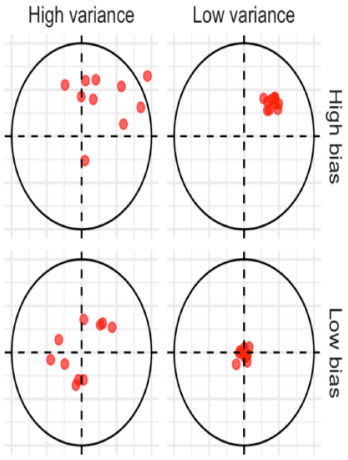
$$\text{Bias}_{\mathbf{y}} = \mathbb{E}[\hat{\mathbf{y}}] - \mathbf{y}$$

High bias ⇒ model not sensitive enough to the data (underfitting). OLS is unbiased under Gauss-Markov: $(\mathbb{E}[\hat{\boldsymbol{\beta}}_{\text{OLS}}] - \boldsymbol{\beta}) = 0$.

Variance (of predictions): how sensitive predictions are to the particular training dataset.

$$\mathbb{V}_{\hat{\mathbf{y}}} = \mathbb{E}\big[(\mathbb{E}[\hat{\mathbf{y}}] - \hat{\mathbf{y}})^2\big]$$

High variance ⇒ resampling yields very different predictions (overfitting). OLS has the **lowest variance of any unbiased linear estimator** (BLUE, Gauss-Markov).



So can't we just choose a low-variance, low-bias strategy? No, they are linked (reducing variance forces bias up and vice versa). Assuming we could calculate the MSE on some test data $\mathbf{X}'$ given a trained model $\hat{f}$, and decomposing it further:

$$\text{MSE} = \mathbb{E}\big[(\hat{f}(\mathbf{X}') - \mathbf{y}')^2\big] \;\rightarrow\; \text{MSE} = \mathbb{E}\big[(\hat{f}(\mathbf{X}') - \mathbb{E}[\hat{\mathbf{y}}])^2\big] + \big(\mathbb{E}[\hat{\mathbf{y}}] - \mathbf{y}'\big)^2 = \underbrace{\mathbb{V}_{\hat{\mathbf{y}}}}_{\text{Variance}} + \underbrace{\text{Bias}^2}_{\text{Bias}^2}$$

Bias-variance tradeoff explains why we might **not want to use OLS for prediction tasks**:

- **OLS is unbiased** (under GM assumptions) ⇒ MSE is driven entirely by variance. Complex OLS models are sensitive to training data and may perform poorly out-of-sample.
- You cannot intentionally introduce a little bias to reduce variance, because the estimator is constrained to be unbiased.

⇒ This is the motivation for **regularisation** (Week 4: LASSO, Ridge), which deliberately introduce bias in exchange for lower variance and better out-of-sample performance.

Exam — key distinctions to know:

Loss vs. Cost: loss is per observation; cost is the aggregate over the dataset (or its mean).

Training error vs. test error: training error measures performance on seen data; test error measures generalisation. A model can have 0 training error (overfitting) while failing on new data.

Why accuracy misleads with imbalanced classes: predicting the majority class always achieves high accuracy. Always compare model accuracy to the majority-class baseline.

AUC interpretation: AUC = 0.5 means the model is no better than random. AUC = 1 means perfect discrimination.

Bias-variance trade-off in one sentence: reducing one component of reducible test error necessarily increases the other; the goal is to find the model complexity that minimises their sum.

Why OLS can be bad for prediction: unbiasedness means all MSE comes from variance; complex OLS models are sensitive to training data and can generalise poorly.

Week 4: Hyperparameters, LASSO and Regularisation

So far we have seen that OLS has high variance (it does not generalize well) and is unbiased by construction (all the error comes from variance) → We introduce **regularization**: intentionally adding bias to the model to reduce variance (↓ total loss curve) and improve out-of-sample predictions. We modify the objective function by adding a penalty term λ R(f) that penalizes large coefficients. Depending on how we define R(·), we obtain **LASSO** (L1), **Ridge** (L2) or **Elastic Net** (a mixture). Once we have the model, we need to choose the value of the hyperparameter λ —which cannot be optimized with gradient descent— and for this we need **k-fold cross-validation** combined with hyperparameter optimization (grid, random, or TPE/Bayesian).

Sample Question — matching key concepts.

- Trying to improve model performance by reducing complexity → **Regularisation**
- Reducing parameter values to constrain model complexity → **Shrinkage** (a specific form of regularisation)
- Total penalty for bad predictions across the dataset → **Cost**
- Absolute difference between true and predicted value → **L1-Loss**
- Penalty for a bad prediction (single observation) → **Loss**
- How different estimated parameters would be across datasets → **Variance**

Note: shrinkage is a subset of regularisation (complexity reduced by shrinking β towards 0); cost is the aggregate loss (W3 distinction: loss per obs, cost over dataset).

Regularisation and Overfitting

OLS models have *low bias* (adjusts perfectly to training data) but *high variance* (does not fit new data, fails to generalise).

Solution: **REGULARISATION PROCESS**: deliberately add bias to ↓ variance (limit overfitting) and ↓ the total MSE on out-of-sample data. Aims to yield better predictions on $\mathbf{X}'$.

Penalises complexity (constrains the model to be simpler, ex. OLS with many parameters) → Minimization function (Kleinberg et al. 2015), **general regularised optimisation**:

S'optimitza el model (fins ara sense restriccions), però aquí $\lambda R(f)$ és un cost additional per tenir coeficients grans. Si el model vol pujar un coeficient per guanyar una mica d'ajust, ha de pagar un preu. Això el força a ser més simple, menys sensible al soroll de training data, i per tant a generalitzar millor.

$$\arg\min_f \underbrace{\sum_{i=1}^{N}(y_i - f(\mathbf{x}_i))^2}_{\text{sum of squared errors}} + \underbrace{\lambda R(f)}_{\text{regularisation}}$$

where f is any fitted algorithm (regression, random forest,...), $R(\cdot)$ is the **regularisation function** applied to that model, and λ is the **penalty scalar** (hyperparameter set by the researcher before training). Com més gran λ, més aggressivament 'shrinks' els coeficients cap a zero.

In OLS $\lambda = \frac{1}{\infty} = 0$: no regularisation, error is explained entirely by variance (you recover the OLS without penalization).

$R(\cdot)$ **as Shrinkage**: to ↓ **complexity** of the model, **shrink** coefficient estimates towards 0. [An OLS model with k parameters estimates a non-zero β for every predictor regardless of its importance. Shrinkage moves $\beta_0, \ldots, \beta_k$ towards 0, reducing sensitivity to individual training observations and therefore variance.]

Goal: ↓ variance error by *more* than the ↑ in bias error, yielding a net improvement in out-of-sample MSE.

Note on standardisation: predictors must be standardised before fitting LASSO/Ridge. The penalty is applied to raw coefficient magnitudes, so a predictor on a large scale (e.g. GDP in trillions) will have a tiny coefficient and be under-penalised, while a binary predictor will be over-penalised. Standardising puts all coefficients on the same footing, so that the penalty treats coefficients comparably. → Feature engineering (Week 7).

Type of shrinkage1: LASSO (Least Absolute Shrinkage and Selection Operator)

L1 norm — for any coefficient in your model, sum up the absolute values of the coefficients:

$$\|\boldsymbol{\beta}\|_1 = \sum_j |\beta_j|$$

This is a *score of complexity*: the higher this number, the more the coefficients are moving the prediction, i.e. the more complex the model.

We can restrict the size of this norm and **add it to our objective function** as a constraint (LASSO sets $R(f) = \|\hat{\boldsymbol{\beta}}\|_1$):

$$\arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{N}(y_i - \mathbf{x}_i\boldsymbol{\beta})^2 \quad \text{subject to } \|\boldsymbol{\beta}\|_1 \leq t \quad \text{(L1 norm)}$$

We still aim to minimise the difference between observed and predicted, but $\boldsymbol{\beta}$ (L1 norm) should be less than a certain value t. The "subject to" constraint has no direct mathematical operator, so we convert it by replacing it with a + term (what was a constraint becomes a penalty):

Equivalently, constrained form: $\arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{N}(y_i - \mathbf{x}_i\boldsymbol{\beta})^2 + \lambda\|\boldsymbol{\beta}\|_1$

λ is the Lagrangian equivalent of t (more intuitive to tune as a scalar penalty). Adding this term lowers the loss curve: the model can still minimise squared error, but every time it increases $|\beta_j|$ it incurs an extra cost (trade off fit VS complexity). Any value inside the constraint ($\|\boldsymbol{\beta}\|_1 \leq t$) is permissible, but moving further inside means moving further from $\hat{\boldsymbol{\beta}}_{\text{OLS}}$, increasing the squared error. The optimum is the point that touches the constraint boundary→ corner solution.

The Corner Solution: Why LASSO Yields Exactly Zero Coefficients

LASSO often yields coefficient estimates of **exactly 0**:

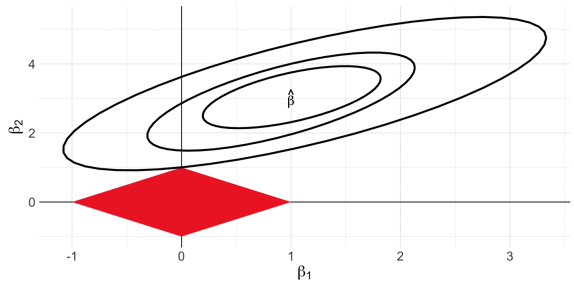


Figure 2: Example of LASSO corner solution

Think of it as a constrained search. We have 2 objects:

- The **diamond**: all (β_1, β_2) where $\|\boldsymbol{\beta}\|_1 \leq 1$, the feasible region (all values that are $\beta < t$). We must stay inside it.
- The **isocurves** (of loss): each ellipse is a ring of observations with equal squared-error loss (inner ellipses = lower loss). The centre is $\hat{\boldsymbol{\beta}}_{\text{OLS}}$ (what OLS would give unconstrained).

Goal: find the *smallest* ellipse (lowest loss) that still touches the diamond. Any point strictly inside the diamond is suboptimal — we could move closer to $\hat{\boldsymbol{\beta}}_{\text{OLS}}$ while still satisfying the constraint. So the solution is always on the **boundary** of the diamond (ex. 1,0 or 0,1, where at least 1 coefficient is always 0).

Relevance of LASSO:

1. **Prediction accuracy:** accepts some bias for a (hopefully) greater reduction in variance, improving out-of-sample prediction. Particularly useful with high-dimensional data (many predictors relative to observations).
2. **Variable selection:** corner solutions (yielding coefficient=0) exclude variables automatically. Variables set to zero are probably less important predictors. Performs variable selection for you: ex. useful in social science for selecting controls...
3. **Bonus:** aids fitting when predictors are highly correlated (helps convergence under multicollinearity).

Type of shrinkage2: Ridge Regression

L2 norm - sum of squared coefficients: $\|\boldsymbol{\beta}\|_2^2 = \sum_j \beta_j^2$

Plugged into the loss function, we get RIDGE REGRESSION ESTIMATOR:

$R(f) = \|\boldsymbol{\beta}\|_2^2$, giving: $\arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{N} (y_i - \mathbf{x}_i\boldsymbol{\beta})^2 + \lambda\|\boldsymbol{\beta}\|_2^2$

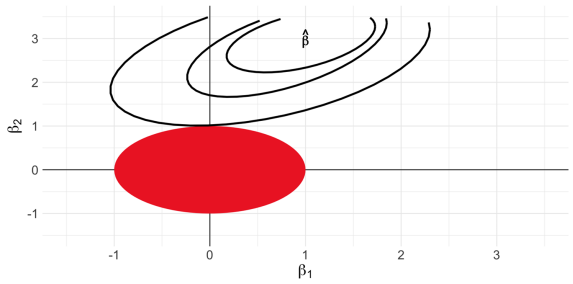


Figure 3: Ridge regression does not yield corner solutions

The L2 constraint is **circular** (smooth, no corners) ⇒ the isocurve touches the constraint (circle) at a smooth point, not a corner. Ridge shrinks all coefficients *towards* zero but **not to exactly 0** — it does **not** perform variable selection.

Ridge is **less aggressive** than LASSO: it provides constant proportional scaling of all coefficients. Useful when you want to shrink without eliminating predictors.

Gradient of ridge penalty — always differentiable, so vanilla SGD works:

$$\frac{\partial \lambda \sum_j \beta_j^2}{\partial \beta_k} = 2\lambda\beta_k$$

Full SGD update for ridge regression: (adds penalty gradient to standard gradient):

$$\text{grad} = \underbrace{(\hat{y}_i - y_i) \cdot \mathbf{x}_i^{(k)}}_{\text{standard logistic gradient}} + \underbrace{2\lambda\beta_k}_{\text{ridge penalty gradient}} \qquad \Rightarrow \qquad \beta_k \leftarrow \beta_k - \lambda_{\text{lr}} \cdot \text{grad}$$

```
def ridgeLoss(ytrue, yhat, coefficients, lambda_):
    nll = NLL(ytrue, yhat)                          # squared-error / NLL term
    l2_penalty = lambda_ * np.sum(coefficients**2)  # lambda * ||beta||^2_2
    return nll + l2_penalty
# Inside the SGD loop (one observation at a time):
grad = (yhat_i - y[i]) * row_vec   # standard logistic gradient
l2_penalty_grad = 2 * lambda_ * coefs          # 2*lambda*beta_k
coefs = coefs - l_rate * (grad + l2_penalty_grad)
```

Elastic Net (LASSO + Ridge)

Elastic Net combines L1 and L2 regularisation using a **blending parameter** $\alpha \in [0, 1]$:

$$\arg\min_{\boldsymbol{\beta}} \underbrace{\sum_{i=1}^{N}(y_i - \mathbf{x}_i\boldsymbol{\beta})^2}_{\text{sum of squared errors}} + \underbrace{\lambda}_{\text{penalty scale}} \left(\underbrace{\alpha\|\boldsymbol{\beta}\|_1}_{L_1} + \underbrace{(1-\alpha)\|\boldsymbol{\beta}\|_2^2}_{L_2}\right) \qquad \alpha \in [0, 1]$$

- $\alpha = 1$: pure LASSO $\quad \alpha = 0$: pure Ridge
- $0 < \alpha < 1$: blend of both (constraint that is a blend of diamond-circle): some 0s → Less sharp corner solutions than LASSO, less pure variable selection
- 2 hyperparameters to tune: λ (penalty scale) and α (blend)
- Better with correlated predictors (see below)

	Penalty	Constraint shape	Exact zeros?	Variable selection?
LASSO	L1	Diamond (corners)	Yes	Yes
Ridge	L2	Circle (smooth)	Rarely	No
Elastic Net	L1 + L2	Blend	Sometimes	Partial

Sample Question — True/False: LASSO coefficients are always smaller than Ridge (same λ). False.

Intuition: LASSO uses a threshold mechanism — coefficients below the threshold are set to exactly 0, but coefficients *above* the threshold are shrunk by a constant amount (parallel shift) and then grow at the same rate as the OLS coefficient. Ridge, by contrast, applies proportional scaling all the way along — every coefficient is a constant fraction of the OLS value, so large coefficients are shrunk more in absolute terms.

Consequence: for a sufficiently large OLS coefficient, the LASSO value (shrunk by a fixed amount but then growing linearly) can *exceed* the Ridge value (which is always a fixed proportion of OLS). So LASSO coefficients are **not** always smaller.

Choosing a Generalisable Model

To choose the best regularised model, we need to set 3 things:

1. **Hyperparameters to tune**: Values that change the behaviour of the model or the learning algorithm. Set by the researcher *before training* begins and held constant throughout (vs **parameters** ($\boldsymbol{\beta}$), determined *during* training via gradient descent).

2. **Method for validating models** → cross-validation

Problem: **Why can't we just optimise λ with gradient descent?** Because the best λ to minimise the *training* loss is always 0 (no penalty, perfect fit). λ must be fixed externally and compared using held-out validation data.

Solution: **K-Fold Cross-Validation**: Divide *training* data into K (mutually-exclusive) folds. Each fold acts as the validation set once; all other folds serve as training data in that iteration. Assumes observations are i.i.d.

El λ que dóna menys error de validació és el guanyador. Llavors reentrenes el model final amb totes les dades d'entrenament i aquell λ, i avalues una sola vegada al test set.

Algorithm:

1. Randomly partition training data $\mathcal{D}$ into K parts of equal length: $d_1, \ldots, d_K$
2. Initialise empty error vector $\mathbf{E} \leftarrow (e_1, \ldots, e_K)$
3. For $k = 1, \ldots, K$:
   - $\text{Tr}_k \leftarrow \mathcal{D} - d_k$ (training: all folds except k)
   - $\text{Te}_k \leftarrow d_k$ (validation: fold k)
   - Train model $g(\text{Tr}_k)$; predict $\hat{y}_{\text{Te}_k}$; compute e_k; insert into $\mathbf{E}$
4. Return $\sum_{k=1}^{K} \frac{n_k}{n} e_k$ (weighted mean; n_k/n accounts for unequal fold sizes)

Choosing K: popular values are 5, 10, n. → Recommended: $K = 10$ — good bias-variance balance, computationally feasible for most models. $\uparrow K \Rightarrow \downarrow$ bias, $\uparrow$ variance (yet another bias-variance trade-off!).

K-fold cross-validation (definition). The correct answer: within the training split, the data is subdivided into k folds; for each model specification, k models are trained, each leaving out one fold for Validation; fit metrics are averaged across all k models; the specification with the best average cross-validated fit is chosen.

Common wrong answers:

- **(a)** "Train a different model on each of the k splits and choose the best" — wrong: you train *every* candidate specification k times, not one model per fold.
- **(b)** "Use a fixed validation split" — wrong: that is a simple train/val/test split, not cross-validation. CV rotates the validation fold across all k partitions.

3. **Method for choosing which combinations to test**

With K-fold CV we can validate a given combination efficiently. But we need to decide *which values of* λ to test.

A. Grid Search

(tries tots els valors d'una graella predefinida. Exhaustiu però lentíssim, l'espai creix exponencialment amb el nombre d'hiperparàmetres)

1. For each value of λ, compute cross-validation error
2. Select λ with smallest CV error
3. Refit model on entire training data with selected λ

Limitations: SLOW (search space grows exponentially with: number of hyperparameters, number of discrete cuts per continuous dimension) $\Rightarrow$ **curse of dimensionality**.

B. Random Hyperparameter Search

(mostres valors aleatòriament d'una distribució, amb un pressupost computacional fix. Millor que grid search perquè si un hiperparàmetre importa poc, no malbarateixes trials provant molts valors seus)

Randomly sample a subset of possible models. Treat each hyperparameter as a **distribution** and sample values from it. Set a **computational budget** (number of samples) to limit time.

Why better than grid search? Suppose λ is important but α is not. A grid search wastes trials testing many values of α that make no difference, while only testing a few values of λ. Random search, by contrast, samples independently in each dimension, so with the same budget, you end up with more distinct values of λ tested, and are more likely to land near the optimum.

However, (1) Still no memory of past trials, (2) can waste budget by sampling twice in similar regions by chance.

C. Modelled / Bayesian Optimisation / TPE

(aprèn dels trials anteriors per decidir on provar a continuació. Modela quins valors han funcionat bé fins ara i prioritza zones prometedores. Optuna és la llibreria estàndard a Python; usa l'estimador TPE i pot aturar trials dolents aviat (pruning) per estalviar còmput)

Grid and random search treat each trial as independent. **Bayesian optimisation** uses results so far to decide what to try next:

- Fit a probabilistic model of performance as a function of hyperparameters
- Choose the next configuration using an **acquisition rule** (e.g. expected improvement)

Tree-structured Parzen Estimator (TPE): Splits past trials into "good" vs "bad" sets and models $p(x \mid \text{good})$ and $p(x \mid \text{bad})$. The ratio $p(x \mid \text{good})/p(x \mid \text{bad})$ is a likelihood ratio used to rank candidate configurations. Advantages: works well with mixed discrete/continuous parameters.

Optuna: Python framework for hyperparameter optimisation using TPE + pruning.

- **Pruning/early stopping**: strategy to avoid wasting compute: monitor intermediate results (e.g. fold-by-fold CV scores). If running scores are much worse than past trials, stop the trial early and mark it pruned.

`Study`	whole optimisation session (history of all trials + best result)
`Trial`	One go (a set of hyperparameter values) + its outcome
`Sampler`	How Optuna proposes new trials (e.g. `TPESampler`)
`Pruner`	When Optuna stops a trial early (early stopping)

```
import optuna
from optuna.samplers import TPESampler
from optuna.pruners import MedianPruner

def objective(trial):
#calls 'objective' function once per trial, creating new 'trial' object for each:

    lambda_ = trial.suggest_float("lambda_", 1e-4, 1e1, log=True)
    # suggests a  value (between 1e-4 and 10, in log scale)

    # k-fold
    cv = KFold(n_splits=10, shuffle=True, random_state=42)
    fold_mse = []

    # Splits training data in train/validate for this fold:
    for fold, (tr_idx, va_idx) in enumerate(cv.split(Xtrain)):

        mod = Lasso(alpha=lambda_).fit(Xtrain.iloc[tr_idx], ytrain.iloc[tr_idx])
        mse = np.mean((ytrain.iloc[va_idx] - mod.predict(Xtrain.iloc[va_idx]))**2)
        fold_mse.append(mse)
        trial.report(np.mean(fold_mse), step=fold) # report accumulated error

        # pruner checks if this trial is running too slow
        if trial.should_prune():
            raise optuna.TrialPruned()
        # if Yes, stop trial now (don't complete the 10 folds)
    return np.mean(fold_mse) # if it gets here, return final error

study = optuna.create_study(
    direction="minimize", #aim: minimize MSE
    sampler=TPESampler(seed=42), # TPE
    pruner=MedianPruner(n_startup_trials=5, n_warmup_steps=2),)
    #indicate if you want first 5 trials to keep going (to store sufficient memory)
    # inside each trial, first 2 folds are not cut either

study.optimize(objective, n_trials=100) #calls 'objective' 100 times
lambda_min = study.best_params["lambda_"] #contains lambda with lower validation error
```

Seminar 2: Building Ridge from Scratch & Optuna on Applied Data

Ridge Loss and SGD Update

```
def ridgeLoss(ytrue, yhat, coefficients, lambda_):
    nll = NLL(ytrue, yhat)                          # negative log-likelihood
    l2_penalty = lambda_ * np.sum(coefficients**2) # L2 penalty term
    return nll + l2_penalty

# Inside training loop (SGD update):
grad = (yhat_i - y[i]) * row_vec          # standard logistic gradient
l2_penalty_grad = 2 * lambda_ * coefs     # ridge penalty gradient
coefs = coefs - l_rate * (grad + l2_penalty_grad)
```

Applied Pipeline: Communities & Crime Dataset

Predict violent crime per capita (`ViolentCrimesPerPop`) on unseen communities.

```
# 1. Split (do once, forget test until the end)
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2, random_state=42)

# 2. Tune lambda with Optuna (TPE + k-fold CV + pruning)  -> see objective() above
study.optimize(objective, n_trials=100)
lambda_min = study.best_params["lambda_"]   # sklearn calls this 'alpha'

# 3. Fit final model on ALL training data
final_mod = Lasso(alpha=lambda_min).fit(Xtrain, ytrain)

# 4. Inspect zero coefficients (variable selection)
print(sum(final_mod.coef_ == 0))

# 5. One final test evaluation
test_mse = np.mean((ytest - final_mod.predict(Xtest))**2)

# Extra: compare to OLS (no regularisation)
from sklearn.linear_model import LinearRegression
ols_mse = np.mean((ytest - LinearRegression().fit(Xtrain, ytrain).predict(Xtest))**2)
```

Note: `sklearn`'s `Lasso` and `Ridge` use the argument `alpha` for what the lecture calls λ.

Exam — key distinctions:

Parameter vs. hyperparameter: parameters ($\boldsymbol{\beta}$) are learned during training via gradient descent. Hyperparameters (λ, α, learning rate) are fixed before training by the researcher. The best λ cannot be found by gradient descent — it would always converge to 0.

LASSO vs. Ridge: LASSO (L1, diamond constraint) yields exact zeros and performs variable selection. Ridge (L2, circular constraint) shrinks all coefficients proportionally but rarely to zero. Elastic Net blends both.

Why cross-validation for hyperparameter tuning: a single validation split is noisy and wastes data. K-fold recycles every observation as validation data once, giving K independent estimates of generalisability without permanently holding out data.

The full pipeline in order:

1. Hold out test set immediately — touch it only once at the very end
2. Tune hyperparameters using K-fold CV on training data only

3. Refit final model on all training data with best hyperparameters
4. Evaluate once on test set and report

Grid vs. random vs. TPE: grid is exhaustive but faces the curse of dimensionality; random sets a budget and samples continuously; TPE learns from past trials to guide search and prunes bad trials early.

Regularisation adds bias on purpose: it trades some bias for reduced variance, lowering total out-of-sample MSE. "ML is biased" — yes, by design; that is how we go from high-variance statistical inference to generalisable machine learning.

L2 norm squared. For $\boldsymbol{\beta} = [2, -1, 3]$:

$$\|\boldsymbol{\beta}\|_2^2 = \sum_j \beta_j^2 = 2^2 + (-1)^2 + 3^2 = 4 + 1 + 9 = \mathbf{14}$$

Common errors:

- Computing $\|\boldsymbol{\beta}\|_2 = \sqrt{14} \approx 3.742$ (the norm, not the norm *squared*)
- Computing the L1 norm: $|2| + |-1| + |3| = 6$
- Computing $(|2| + |-1| + |3|)^2 = 36$ (squaring the L1 norm)

Week 5: Proximity and Partitioning

Until now we visualized data in the classical way: Y on the vertical axis, X on the horizontal axis. Now we set aside the Y axis and imagine each observation as a **point in the feature space** — a p-dimensional space where each axis is a predictor. The dataset is a finite sample of points in this space, and Y becomes simply the *color* of each point, not another axis.

Think of a city map: each person in the dataset is a building placed according to their characteristics (age, income, religiosity…). The goal is to find the boundary that separates the red buildings (Trump) from the blue ones (Clinton). From this, two strategies emerge:

Proximity - KNN: when a new unlabeled point arrives, you look at the k nearest neighbors and say *"if everyone living around you is blue, you probably are too"*. The logic is: things that are similar in feature space tend to have similar outcomes.

Partition - CART: instead of looking at neighbors one by one, you draw splits that cut the space into regions—and you try to ensure that within each region all points share the same Y. You are not interested in who each point's neighbor is; you want to find the cuts that make each zone as homogeneous as possible.

Both ideas share the same intuition: **similar things should have similar outcomes**. The difference lies in how they implement it (one looks around, the other draws boundaries).

Feature spaces

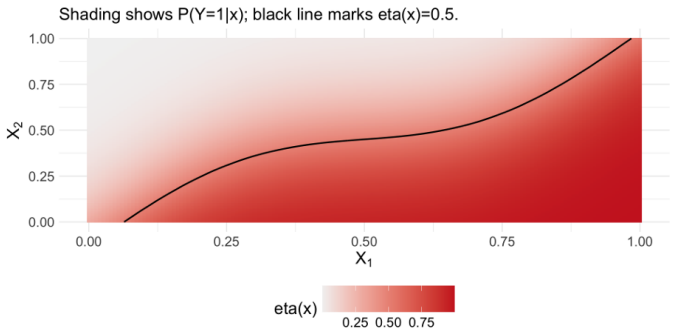
Each observation i is a point $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ip}) \in \mathcal{X}$.
Each row in your dataset is just a list of numbers — age, income, religiosity... That list defines a unique position in space. If you have 2 predictors, it's a point on a 2D graph. If you have 100 predictors, it's a point in 100-dimensional space (impossible to draw, but the maths works the same).

- $\mathcal{X}$: **feature space**: all possible combinations of inputs. Our data is a finite sample from it. Universe of all possible people that could exist in your dataset — every possible combination of age, income, religiosity, etc. Your actual dataset is just a handful of dots sampled from that universe. You never observe the whole space, only a slice of it.
- In *low* dimensions, we can visualise it directly; in high dimensions, we cannot.
- Y is a **label/category/colour** on those points, not an axis.

We want to learn how outcomes Y vary across $\mathcal{X}$ → The **PROBABILITY SURFACE** (function that, for each point x/ each zone of the map, the probability of being = 1.

- **Response surface** (continuous Y): $m(\mathbf{x}) = \mathbb{E}[Y \mid \mathbf{X} = \mathbf{x}]$
- **Class-probability surface** (binary Y): $\eta(\mathbf{x}) = \Pr(Y = 1 \mid \mathbf{X} = \mathbf{x})$ → probability of being class 1 if situated at point $\mathbf{x}$ of the map

The red shading is $\eta(\mathbf{x})$ across the feature space. Black line = decision boundary $\eta(\mathbf{x}) = 0.5$ (where the model is undecided): right → class 1 (deep red); left → class 0 (near white). Curved because the relationship is non-linear.



2 STRATEGIES (once you have points in space):

The 1st - KNN is looking around. A new unlabeled point arrives, and you say: "I look at the k nearest points that already have a color, and assign it the majority color". It is the logic of the neighborhood: if all your neighbors voted for Clinton, you probably did too.

The 2nd - CART is drawing boundaries. Instead of looking neighbor by neighbor, you draw lines that divide the space into zones, and you try to ensure that within each zone all points share the same color. You do not care who anyone's neighbor is; you want each zone to be as pure as possible.

Both share the same bet: similar things should have similar outcomes. The difference is the method.

1. K-Nearest Neighbours (KNN)

Non-parametric: not defining regression coefficients in advance. Instead, the distribution of points in space *is* the model structure (we expect closer points to have more similar outcomes).

Lazy learner: no fitting procedure needed (no gradient descent, no optimisation). Unlike Ridge or LASSO where we search for optimal $\boldsymbol{\beta}$, here the **training data itself *is* the model**. KNN just stores data and waits. No theory needed, just proximity.

Still supervised: training data must be labelled (not test data, we just look for its labelled neighbours). This distinguishes KNN (outcomes y_i are needed) from K-means, which is unsupervised.

Algorithm: knn predict() does the work

1. **Classification** — for a new point $\mathbf{x}^*$:
    1. Compute distance $d(\mathbf{x}^*, \mathbf{x}_i)$ for every training observation $(\mathbf{x}_i)$.
    2. Find $N_k(\mathbf{x}^*)$: the k closest training points.
    3. Return the **majority class** (encode classes as -1 and $+1$ so the sign does the counting):

$$g(\mathbf{x}^*) = \text{sign}\left(\sum_{x_k \in N_k(\mathbf{x}^*)} y_k \right), \qquad y_k \in \{-1, +1\}$$

Returns $+1$ if more neighbours are positive, -1 otherwise.

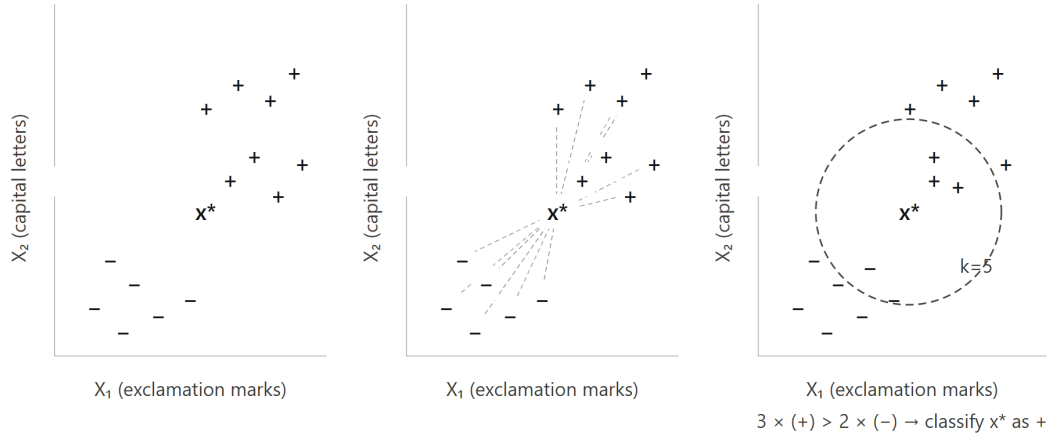


Figure 4: Steps 1, 2, 3

2. **Regression** — same N_k, different aggregation: instead of majority vote, take the average: (called "prediction" in the slides — continuous Y) — same N_k, different aggregation)

$$g(\mathbf{x}^*) = \frac{1}{k} \sum_{x_k \in N_k(\mathbf{x}^*)} y_k, \qquad y_k \in \mathbb{R}$$

Distance metrics

Metric	Formula	When to use
Euclidean (most common)*	$\sqrt{\sum_{j=1}^{J}(x_a^{(j)} - x_b^{(j)})^2}$	Default; continuous features
Manhattan**	$\sum_{j=1}^{J} \lvert x_a^{(j)} - x_b^{(j)} \rvert$	Better when J is large
Hamming	count of differing bits	All-categorical data

*To get the length of a diagonal: difference between 2 points in a dimension, sum across all dimensions, and then square root
**Taking the observed differences and summing them

```
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier

# Scaling is mandatory for KNN: Euclidean distance is the sum of squared differences.
# If income ranges 0-100k and age 18-90, income dimension will dominate in the
# distance calculations over age, unless both are standardised (mean 0, sd 1).
scaler = StandardScaler()

# Always fit the scaler on training data only, then transform (apply same scaler)
# both train and test. Never fit on test data (it would be data leakage).
X_train_s = scaler.fit_transform(X_train)   # fit on train only
X_test_s  = scaler.transform(X_test)        # same scaler, no refit

# When a new point needs classifying, it will look at its 10 nearest neighbours
knn = KNeighborsClassifier(n_neighbors=10)

# Stores the scaled training data X_train_s and the labels y_train in memory
knn.fit(X_train_s, y_train)

# For each test point in X_test_s, sklearn:
```

```
# (1) Computes Euclidean distance to every single point in X_train_s
# (2) Finds the 10 closest ones,
# (3) Takes a majority vote of their labels in y_train
# (4) Returns that as the prediction
y_pred = knn.predict(X_test_s)
```

Bias–variance via k

- $k = 1$: only 1 single nearest point decides ⇒ very low bias, very high variance (overfitting). 100% train accuracy trivially.
- $k = N$: predicts the global mean for everyone ⇒ very high bias, 0 variance (underfitting).
- Choose k via **cross-validated hyperparameter search** (e.g. Optuna).

Advantages	**Disadvantages**
Minimal assumptions (closer points → + similar)	**Curse of dimensionality**: with many features/dimensions, "closeness" breaks down
No parametric form required	**Computationally expensive at prediction**: must compute n distances per query
Trivially extends to multi-class	**High memory**: the entire training set must be stored
Very intuitive algorithm	Does not scale to large n or large p

2. Classification and Regression Trees (CART)

Trees sit between regression (rigid, fully specified) and KNN (no structure at all).
They **partition feature space** *inductively* by splitting on predictors, building squares/rectangles in that space. All tree-based methods (random forests, XGBoost, BART) rest on this same logic.

A single-tree model fitted **once** on $\mathbf{X}$ to predict unseen $\mathbf{X}'$. Two types: **classification trees** (predict which *class* an observation belongs to) and **regression trees** (predict the outcome of an observation).

Across both, training determines: (1) the structure of the tree, (2) the variable to split on at each node, (3) the splitting criterion c at each node.

Key terminology

- **Decision node**: a junction where the data is split into 2. Requires two things: (1) a *variable* to split on and (2) a *value/threshold* c as splitting rule (e.g. age ≥ 42?).
- **Branch**: yes/no answer coming out of a decision node (path leading to next node).
- **Root node**: the initial node — the full training data before any split.
- **Terminal node/leaf**: the final node along a branch (no further splits). Returns a prediction/label for y (outcome).
- **Depth**: maximum number of decision nodes (questions) between the root node (full data) and any leaf node (final prediction).

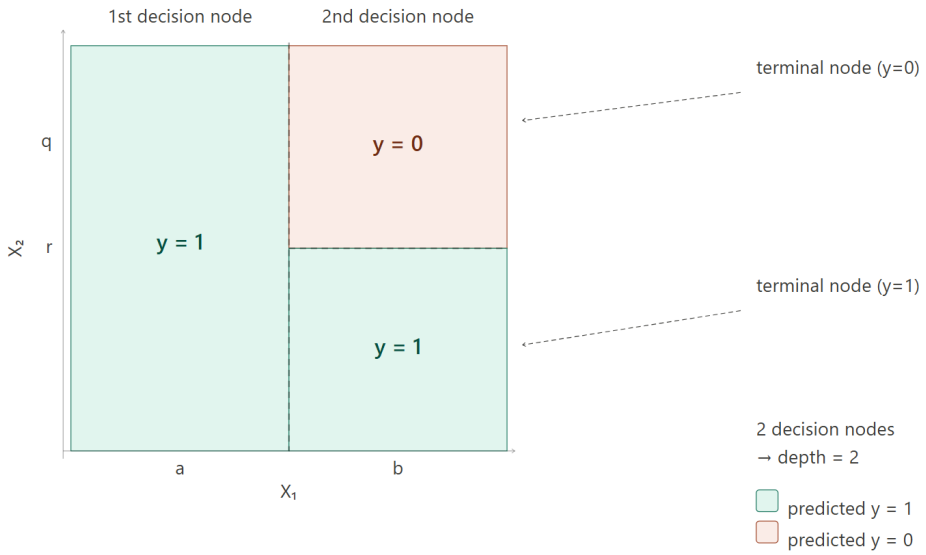


Figure 5: Partitioning the feature space with 2 decision nodes (split on X_1 at value a, then on X_2 at value r for the $X_1 = b$ branch). 3 terminal nodes return predictions. Maximum depth = 2.

General algorithm (both types)

1. Start with full training data $\mathbf{X}$ at the root node.
2. For *every* variable and *every* splitting point c along it, split the data and evaluate the quality of that split.
3. Choose the variable + c that yields the **best** split → Fixate that split.
4. Either stop (terminal node), or repeat from step 2 on the subset.

The algorithm is **greedy**: each split is chosen to be locally optimal given the current partition (it does not look ahead or re-evaluate past splits). A split that looks good **now** may be suboptimal once you see what comes later. This is a key disadvantage and a major source of high variance.

How is the best split chosen and when does it stop?
Objective for both: minimise classification error in $\mathbf{y}$ *for training data* $\mathbf{X}$.

A. Classification trees

To know whether a split is good, you need to measure **uncertainty** (how mixed the colors are within each zone). For *classification*, the tool is *information gain*: the difference between the uncertainty before the split (parent node) and the uncertainty after (child nodes, weighted by the number of observations that go into each).

How much uncertainty do we ↓ by splitting the data at a certain point? We want features of $\mathbf{X}$ such that dividing on them yields more *certain* predictions about y_k for each k.

Information Gain: increase in certainty from a split:

$$IG = I(\mathbf{y}^{\text{Parent}}) - \left(\frac{n_{\text{Left}}}{n_{y^{\text{Parent}}}} I(\mathbf{y}^{\text{Left}}) + \frac{n_{\text{Right}}}{n_{y^{\text{Parent}}}} I(\mathbf{y}^{\text{Right}}) \right)$$

```
def information_gain(y_parent, l_idx, r_idx, criterion=gini_index):
    y_left = y_parent[l_idx]
    y_right = y_parent[r_idx]
```

```
    n = len(y_parent)
    n_l = len(y_left)
    n_r = len(y_right)

    I_p = criterion(y_parent)
    I_l = criterion(y_left)
    I_r = criterion(y_right)

# calcula incertesa del pare, resta la mitjana ponderada de la incertesa dels 2 fills
    ig = I_p - ((n_l / n) * I_l + (n_r / n) * I_r)
    return float(ig)
```

- **Highest** when there is lots of uncertainty in the parent (molts vermells i blaus barrejats) and little in the children (un quasi tot vermell, l'altre quasi tot blau).
- Split need **not** be symmetric: the fractions $\frac{n_{\text{Left}}}{n_{y^{\text{Parent}}}}$ weight each child's uncertainty.
- IG depends on: (1) the **information function** $I(\cdot)$ (a measure of uncertainty, e.g. Gini); (2) the **threshold** c determining the Left/Right split.

Cross-entropy can also be used; it penalises impurity more smoothly, but both behave similarly (Gini is parabolic, entropy is more stretched out).

```
def gini_index(y):
    gini_c = [np.mean(y == a) * (1 - np.mean(y == a)) for a in np.unique(y)]
    return float(np.sum(gini_c))
```

B. Regression trees

No information function $I(\cdot)$ here — we look directly at the reduction in prediction error.

$$\textit{Prediction error} = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \quad \text{where} \quad \hat{y}_i = \frac{1}{\sum_i \mathbb{I}(\mathbf{x}_i \in \mathcal{R}_k)} \times \sum_i \mathbb{I}(\mathbf{x}_i \in \mathcal{R}_k)\, y_i$$

Similar idea to linear regression but not identical: we recursively partition the data so that the split at node k only uses the data from its parent node, not the full dataset.

When **Y is continuous**, it makes no sense to talk about "mixed classes". Instead of Gini, you look directly at the reduction in prediction error: for each possible split, you compute the mean of each side (β_{Left}, β_{Right}) and sum the squared deviations. You choose the split that minimizes this sum. The final prediction for a new point is simply the mean of the observations that fell into the same terminal node.

Regression algorithm:

1. For every variable and every splitting point c, split the data and calculate the conditional mean for the two branches: β_{Left}, β_{Right}.
2. Calculate the *weighted* total squared loss:

$$\sum_{i \in \mathbf{X}^{\text{Left}}} (y_i - \beta_{\text{Left}})^2 + \sum_{i \in \mathbf{X}^{\text{Right}}} (y_i - \beta_{\text{Right}})^2$$

3. Choose the variable + c that yields the **smallest** total loss.
4. Repeat recursively for each new partition.

Do it for each variable–splitting value combination and keep the one with smallest loss. Prediction for a new observation = mean of y_i in the terminal node it falls into.

Stopping criteria (regularisation for trees)

Without stopping, the tree keeps splitting until each terminal node has 1 observation (equivalent to $k = 1$ KNN, completely overfit).

Method	How	Explanation
A) Natural stop	Stop when $I(\mathbf{y}^{\text{Parent}}) = 0$	Stop when uncertainty = 0 (automatic)
B) Max depth	Stop after λ decision nodes	Fixate max depth
B) Min leaf size	Stop when split yields $< \lambda$ obs. in a child	
C) Pruning	Grow full tree, hold out val. data, successively remove splits, keep tree with lowest val. error	Deixar créixer l'arbre sencer i després podar-lo eliminant les branques que no milloren el rendiment en dades de validació.

Trees vs. linear models

- **Linear decision boundary** ⇒ classification model wins (can draw the diagonal directly; a tree needs many tiny squares).
- **Non-linear/complex boundary** ⇒ tree wins.
- In practice: unknown which applies ⇒ cross-validate across both model families.

Advantages & Disadvantages of Trees

Advantages	**Disadvantages**
Intuitive; mirrors human decision-making	Rarely the best model in terms of prediction
Visualisable (especially small trees)	**High variance**: greedy algorithm amplifies this
Handles categorical variables naturally	Greedy splits cannot be re-evaluated
	Subsetting is not smooth (struggles with truly linear boundaries)

Exam:

In KNN, k controls bias–variance: small k ⇒ low bias, high variance (overfitting); large k ⇒ high bias, low variance (underfitting). Tune with cross-validation.

Gini Index calculation: $I_{\text{Gini}}(\mathbf{y}) = \sum_a P(\mathbf{y} = a)(1 - P(\mathbf{y} = a))$ With two classes: $= 0$ if pure, $= 0.5$ if 50/50. Common error: confusing Gini with misclassification error or cross-entropy.

Information Gain: the right split is the one that maximises IG (biggest drop from parent to weighted children). Highest when parent is uncertain and children are pure.

CART is greedy: splits are chosen one at a time and fixed. Cannot look ahead or revise. Consequence: high variance.

Pruning requires held-out validation data, not cross-validation: because different CV folds may produce structurally incompatible trees that cannot be reconciled.

Trees vs. linear models: trees win on non-linear boundaries; linear models win when the true boundary is linear.

Code errors to watch for (KNN/CART):

- `scaler.fit_transform(X_test)` ⇒ should be `scaler.transform(X_test)`
- `knn.fit(X_test_s, y_test)` ⇒ should use training data
- `knn.predict(X_train_s)` ⇒ evaluates in-sample, not out-of-sample
- `accuracy_score(y_train, y_pred)` ⇒ labels must match the data that was predicted

Week 7: Data in ML Settings

This week there are no new models. The point is that **preparing data well matters more than the choice of model**. Scaling variables correctly has more impact on R^2 or AUC than deciding between Random Forest or neural networks. The lesson is structured in two parts: (1) a brief ethical warning about data, and (2) the practical techniques of *feature engineering* (coding categorical variables, scaling continuous variables, missing values, class imbalance).

0. Data Ethics

3 stages where ethical concerns arise:

Stage	Key concern
Data collection	Accessibility ≠ permission. Surveillance without consent is intrusive even when legal. Scraping may violate terms of service. The NYT vs. OpenAI lawsuit is the canonical example of using data without permission at scale.
Data bias	Even legally-collected data can encode social biases. Twitter's image-cropping AI (2021): trained on human eye-tracking, systematically favoured white faces. The data reflects biassed social processes.
Data usage	GDPR (EU) / Data Protection Act (UK) dictate 7 principles. Most important for ML: **purpose limitation** (can't repurpose data beyond stated use), **data minimisation** (collect only what you need).

Transparency tension in ML: ML research should be transparent on: (1) intended purpose, (2) storage and security, (3) who is the data controller.

Reproducibility: OLS on fixed data always gives the same result. But once stochastic processes enter (random forests, gradient descent, GPU parallelism):

- Results may differ across runs due to random seeds, numerical precision, or even hardware-level randomness.
- **Always set random seeds.** use `random_state=` everywhere.

1. Why Feature Engineering Matters

Why does the same data give different outcomes across models?

- **Distance-based objectives** (KNN): a variable on a 0–1000 scale dominates Euclidean/Manhattan distance over a 0–1 variable, regardless of predictive importance.
- **Gradient-based fitting** (log regression, neural networks): large-scale variables produce large gradients ⇒ oscillating, slow convergence. Scaling makes the loss surface ↑ spherical.
- *Decision trees are unaffected by scale* — splits look at one variable at a time.

Feature engineering = any transformation of **X** before it enters the model:

Technique	Purpose
A. One-hot encoding	Convert categories into binary (sets of) variables
B. Interaction & polynomial features	Capture non-linearities/interactions
C. Re-scaling & centring	On continuous variables, to have similar ranges
D. Missing data handling / imputation	Recover information from incomplete rows
E. Resampling	Fix class imbalance in the outcome

A. One-Hot Encoding (Categorical Data)

ML models need numbers. Categorical variables {cat, dog, fish} have no inherent numeric distance. Solution: convert from a **dense** single-column representation to a **sparse** multi-column representation of binary indicators. This is just dummy variable creation — called **one-hot encoding**.

Example: variable $X_3 \in \{a, b, c\}$

$$a \mapsto [1, 0, 0], \quad b \mapsto [0, 1, 0], \quad c \mapsto [0, 0, 1]$$

A dataset with 3 columns becomes 6 after one-hot encoding 2 categorical variables.

Key difference from conventional regression:

- **Regression (OLS/GLM)**: Reference category ⇒ Must omit one (multicollinearity)
- **ML models**: Not needed ⇒ ML models can handle all classes.

Practical cost: encoding a variable with many levels (e.g. country of origin: ∼192 levels) adds 192 columns. This can grind some algorithms to a halt (especially iterative ones that loop over every variable–split combination e.g. decision trees, some imputation algorithms).

- **However:** Decision trees and tree-based models handle categorical variables *natively* — they search over subsets of levels directly ("is country in {UK, France}?"), so one-hot encoding is actually not needed for trees. This is one reason tree-based methods are so convenient for social-science data.

```
# Option 1 | pandas (simple, in-place)
pd.get_dummies(df, columns=["X3"])

# Option 2 | sklearn (recommended for pipelines: saves encoder for future data)
from sklearn.preprocessing import OneHotEncoder
enc = OneHotEncoder(sparse_output=False)
enc.fit_transform(df[["X3"]])   # fit on train; later use enc.transform(X_test)
```

B. Feature Interactions & Polynomial Features

Beyond transforming existing features, we can **create new ones**:

- **Interaction terms:** $X_1 \times X_2$ captures how the effect of one variable depends on another.
- **Polynomial features:** $X^2, X^3, \ldots$ capture non-linear relationships (e.g. age may have a non-linear effect on vote choice).

Especially useful for **linear models** (LASSO, Ridge) that cannot learn interactions/non-linearities on their own. Whether to include them must be chosen via cross-validation.

Tree-based models learn interactions automatically via conjunctions of splits ("under 18 AND male" = interaction).

C. Rescaling (Continuous Data)

Problem of scale: if $X^{(1)} \in [0,1]$ and $X^{(2)} \in [0,1000]$, KNN will be dominated by $X^{(2)}$ in Euclidean distance, even if $X^{(1)}$ is more predictive.

3 forms of scaling:

Method	Formula	Notes
1.Constant scaling	$\mathbf{x}^{\text{Scaled}} = \mathbf{x}/a$ (dividing by a constant n$^{\text{o}}$)	Intuitive interpretation, but requires a different a for each variable; hard to make ranges identical
2.Min-max scaling	$\mathbf{x}^{\text{Scaled}} = \dfrac{\mathbf{x} - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})}$ (x - its minimum value / range)	All $[0,1]$. Preserves order. Drawback: no negative values, which can constrain optimisation
3.Standardisation (recommended)	$\mathbf{x}^{\text{Scaled}} = \dfrac{\mathbf{x} - \bar{\mathbf{x}}}{\sigma_{\mathbf{x}}}$	Mean = 0, SD = 1. Resembles standard normal. Allows negatives. Industry standard.

Scaler memory — critical for test data:

Test data must be scaled relative to *training* data, not its own:

$$\mathbf{x}^{\text{Test-Scaled}} = \frac{\mathbf{x}^{\text{Test}} - \bar{\mathbf{x}}^{\text{Train}}}{\sigma_{\mathbf{x}}^{\text{Train}}}$$

Scaling test data with its own mean/SD = **data leakage**. The test set can only ever be `transform`ed, never `fit`-ted.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()                          # define scaler object

X_train_scaled = scaler.fit_transform(X_train)
# fit on training data, learn mean/SD and transform

X_test_scaled  = scaler.transform(X_test)
# apply training parameters to new data (SAME mean/SD to test)
# Equivalents: MinMaxScaler, RobustScaler (robust to outliers)
```

Putting It Together: Pipelines

Encoding + scaling + imputation steps must be applied *consistently* to train and test data. Doing this manually is error-prone (e.g. accidentally fitting the scaler on test data). **Pipelines** chain preprocessing steps together:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.neighbors import KNeighborsClassifier

preprocessor = ColumnTransformer([
    ("num", StandardScaler(),  ["age", "income"]),   # scale numeric cols
    ("cat", OneHotEncoder(),   ["party"])            # encode categorical col
])
```

```
pipe = Pipeline([
    ("preprocess", preprocessor),
    ("model", KNeighborsClassifier(n_neighbors=100))
])

pipe.fit(X_train, y_train)    # all steps applied consistently
pipe.predict(X_test)          # same transformations applied to test
```

Key advantage: calling `pipe.predict(X_test)` automatically applies the same train-fitted scaler/ encoder to test data — scaler memory is handled automatically.

D. Missing Data

Missingness in the social sciences is the norm, not the exception. Unlike in causal inference (where missingness biases coefficients), in ML the primary concern is **losing training examples** — which degrades model performance.

Not all missing data is the same:

Type	Example and implication
Truly missing (but exists)	Incomplete survey response, corrupted measurement. Reasonable to impute.
Purposive missingness	Subject refused to answer ("I don't know", abstention). May be *informative* — consider keeping as its own category.
Not actually missing	Skipped due to question logic ("What's your cat's name?" not asked to non-cat-owners). Should remain NA. Imputing would be misleading.

Three strategies:

1. **Listwise deletion:** remove any row with a missing value. Simple. Can bias findings if missingness is non-random. Reduces training set size — bad for data-hungry models.
2. **Recode to a missing category:** for categorical columns, create a new level "missing". Straightforward; preserves the fact that the person refused to answer. Less obvious for continuous columns.
3. **Imputation:** predict the missing values using a model trained on the observed data.

Imputation methods:

Method	Description
Mean imputation	Replace NA with column mean. Does not de-bias the data.
Predictive single imputation	Use observed columns to predict missing values → ML algorithms (missForest, sklearn). More principled; still a single draw (noisy).
Multiple imputation	Generate several imputed datasets. Incorporates prediction uncertainty. Less developed in ML contexts (still rudimentary).

Decision trees handle missing values by default via randomisation (flip a coin at the split node). This is technically valid but not ideal — it can send an observation down a very wrong branch. Always better to impute rather than rely on this default behaviour.

E. Resampling: Class Imbalance

Problem: when one class is rare (e.g. COVID cases at peak $\approx 1/7$ of population, missile strikes), models tend to collapse to the majority class:

- In evaluation: trivially high accuracy ("everyone is negative")
- In training: model never learns to distinguish the minority class ("mode collapse")

Naive approaches:

Method	How	Drawback
Oversampling	Randomly duplicate minority class observations	Exact duplicates — no new information added
Undersampling	Randomly remove majority class observations	Loses training data; model degrades

When to prefer each: *oversample* when the dataset is small (can't afford to lose data); *undersample* when the dataset is huge (reducing it to balance is fine).

SMOTE: Synthetic Minority Oversampling TEchnique *(Chawla et al., 2002)*

Instead of duplicating, SMOTE *synthesises* new minority observations:

1. For each minority class observation, find its K nearest *minority class* neighbours.
2. For each neighbour, draw a vector from the original point to that neighbour.
3. Sample a random number between 0-1. Multiply this number to the vector to get a synthetic point.
4. Place the new synthetic point at λ along that vector.
5. Repeat until class balance is achieved.

Return the j closest synthetic points per minority observation (j as % of oversampling needed divided by 100)

Example: 300% oversampling $\Rightarrow$ we need $j = 3$ synthetic points (3 closest neighbours per minority observation).

Limitations of SMOTE:

Issue	Explanation
Overlapping classes	Synthetic points between two nearby minority observations may land in majority-class territory $\Rightarrow$ introduces noise
High dimensionality	Nearest-neighbour search becomes unreliable in very high dimensions ("curse of dimensionality")
Categorical features	Standard SMOTE only works with continuous features; use SMOTE-NC (Nominal-Continuous) for mixed data

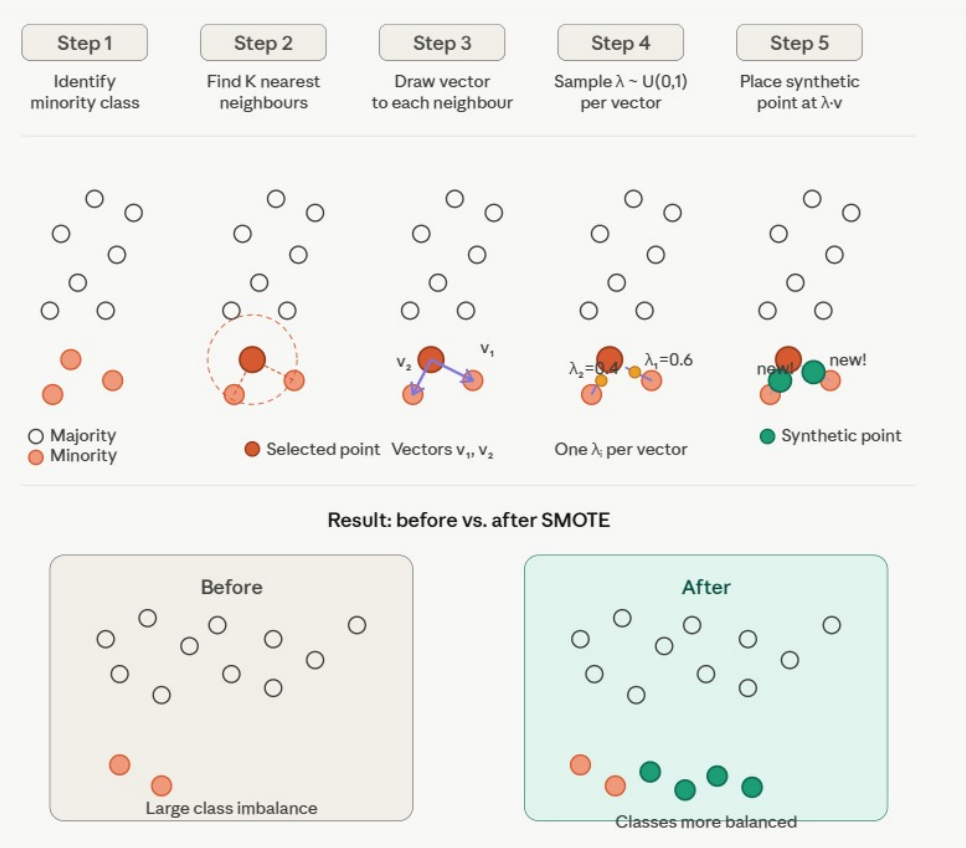


Figure 6: SMOTE > duplicate: the green points do not coincide exactly with any original point, but are instead interpolated between them — this is new information within the same distribution of the minority class.

Exam-style questions (Week 7)

Q1. Suppose a colleague recommends you use KNN in a project. They argue that KNN is a good model choice because "it involves zero training". Which of the following explanations is most compatible with this line of reasoning?

Correct: Unlike many other model forms, when we fit a KNN model, we hold the entire training dataset in memory. Test observations are labelled simply by taking the average/vote of the nearest training datapoints, and so no parameters need to be learned in advance.

- (b) "KNN does not use SGD so it doesn't need training" — The reason is not about SGD; it's about the fact that the data *is* the model. Many non-SGD models still learn parameters.
- (c) "KNN does not make predictions" — KNN absolutely makes predictions.
- (d) "KNN models are pre-trained"—KNN has no pre-training; it just memorises training data.

Q2. To improve the performance of your prediction model, you standardised both your labels and predictors prior to training. The scaling parameters are: $\bar{y} = 13.2$, $\sigma_y = 6.7$, $\bar{X}_1 = 2.5$, $\sigma_{X_1} = 1.8$, $\bar{X}_2 = 423.3$, $\sigma_{X_2} = 189.6$. Your model returns a standardised prediction of 0.57. What is the final prediction on the original y scale?

Correct: 17.019. Reverse the standardisation using only y's parameters:

$$y^{\text{original}} = y^{\text{scaled}} \times \sigma_y + \bar{y} = (0.57 \times 6.7) + 13.2 = 3.819 + 13.2 = \mathbf{17.019}$$

The X_1 and X_2 scaling parameters are a **red herring** — the prediction is already on the y scale; you only need to invert y's own standardisation.

Q3. A colleague suggests the following revision to a decision tree fitting algorithm: after fitting a deep tree, for every decision node, check whether adjusting the splitting criteria (including the splitting variable), holding everything else constant, improves the model fit. If it does, make that change. What feature of decision trees does this revision best attempt to address?

Correct: Greedy splitting. The CART algorithm fixes each split permanently as it is made and never re-evaluates it. This revision explicitly addresses that by going back to reconsider past splits.

- (a) Binary splitting — **Not addressed.** The revision still uses binary splits; it only reconsiders *which* split to make.
- (b) Tree depth — **Not addressed.** The revision does not change how deep the tree grows.
- (d) Recursive splitting — **Not addressed.** The recursive structure (splitting subsets) is unchanged; what changes is whether past splits can be revised.

Q4. [GOT WRONG] Which of the following completes the SMOTE sequence?
1) Identify all minority class observations
2) For each observation, find the K closest neighbours (among the minority class)
3) Calculate a vector that connects each neighbour to the observation
4) Scale each vector using a *separate* draw from a random Uniform(0,1) distribution.
5) Generate new training datapoints at this scaled vector location and add to the training data

- (a) "a single draw from Uniform(0,1)" — We do use Uniform(0,1), but one *independent* draw *per vector*, not a single shared draw for all vectors. Each synthetic point gets its own $\lambda_i \sim U(0,1)$.

Q5. [PARTIALLY CORRECT] Connect each ethical concept to the relevant example:

Scenario	Correct principle
Scraping a forum whose ToS allows it, but users are unaware of researchers' intentions	**Fairness**
Saving training data on Dropbox "just in case it's useful in the future", beyond stated purpose	**Storage limitation**
Neural network cannot explain to a regulator why it denied a life insurance application	**Transparency**
Recording mouse movements in an AI-trust survey not included in the pre-analysis plan	**Purpose limitation**
Same person, same details, same time: rejected one day, accepted the next	**Reproducibility**

- **Fairness** vs. **Purpose limitation**: Fairness = using data people didn't know would be used (legal but ethically questionable). Purpose limitation = collecting *extra* data beyond your stated research purpose. The forum scenario is about users being *unaware* ⇒ Fairness. The mouse scenario is about collecting data *outside the pre-analysis plan* ⇒ Purpose limitation.
- **Storage limitation** vs. **Purpose limitation**: Storage = keeping data longer/more than needed. Purpose = using/collecting data for a different purpose than declared.
- **Reproducibility** = same inputs should give same outputs. Getting different decisions on different days is a reproducibility failure, not a fairness one.

Week 8: Active Learning

We have treated training data as fixed, one-off, and fully labelled. Active learning relaxes all 3 assumptions. Core idea: **instead of passively accepting whatever data we have, we let the current model *choose* what to label next** — selecting only the observations that would improve it the most. The result is the same (or better) model performance with a fraction of the labels.

Passive Learning can be suboptimal

With a fixed training set, the model is **passive**: parameters are learned from $\mathbf{X}$ once, and the model has no say in what new data is collected. 2 problems arise:

1. **Budget constraints**: Random sampling wastes budget labelling easy/redundant cases. Active learning asks: *how do we use a fixed budget to generate the best possible model?*
2. **Concept drift**: [problem in ML] $f(X)$ may not be constant over time. E.g. what counts as 'hate speech' shifts. A passively trained model cannot adapt because it doesn't see updated labels.

How to solve this? Allow our current model to interact with our choice of new training data (select data that will best improve the model). → Highly predictive & flexible model.

Key features:

1. Sequential: model is trained many times
2. Involve (re-) labelling new training examples
3. Model is refit successively

Active learning has 3 components:

1. **Target model** $g(\cdot)$: prediction/classification model you are trying to improve.
2. **Pool** $\mathbf{X}^P$: set of unlabelled observations you *could* label. Features $\mathbf{X}$ are observed; labels $\mathbf{y}$ not (yet). May be depletable (finite set of people) or not (same ad can be shown again).
3. **Querying strategy** $I(\cdot)$: a function that returns an *informativeness score* for each pool observation. We realise labels for the most informative examples.

The Basic Algorithm

Input: initial labelled data $\mathbf{D}^0 = \{\mathbf{X}^0, \mathbf{y}^0\}$ (initial model at time 0), pool $\mathbf{X}^P$

1. Fit $\hat{g}^t$ on $\mathbf{D}^t$
2. Calculate informativeness $I(\mathbf{X}^P, \hat{g}^t)$
3. Select the n observations with highest I scores; request their labels
4. Update $\mathbf{D}^{t+1} \leftarrow \mathbf{D}^t \cup \{\mathbf{X}^t, \mathbf{y}^t\}$; go to step 1

Output: trained model $\hat{g}$

This loops forever without a **stopping criterion**. ⇒ A slightly more refined version adds a check: only continue step 3 if $e = \mathcal{L}(\hat{g}^t(\mathbf{X}^{\text{Test}}), \mathbf{y}^{\text{Test}}) > \epsilon$.
Check whether the reduction in the error has led to a better model, using threshold ϵ.

Common stopping criteria:

Criterion	Description
Budget exhaustion	Stop after a fixed number of labelling rounds (most common)
Performance plateau	Stop when held-out performance has not improved for k consecutive rounds
Informativeness threshold	Stop when the most informative pool example falls below a threshold (model is "confident enough")

Batch Active Learning

In its simplest form, one observation is labelled per step — optimal but very slow. **Batch active learning** chooses $n > 1$ observations at once (e.g. $n = 50$).

Caveat: batch sampling is greedy. The second most informative observation may be redundant once the first is labelled (they may be very close in feature space, conveying the same information). A fully optimal batch strategy would compute *conditional* informativeness, but this is computationally expensive.

Streaming vs. Active Learning

Streaming (labelled data arriving sequentially) is *not* active learning — the model does not control what gets included. This is **continual learning** (beyond scope).

Streaming-based AL: unlabelled data arrives one point at a time; the model must decide *on the spot* whether to request its label (usually at some cost). Harder because only one point is considered at a time, not the full pool.

The Oracle Assumption

Basic AL assumes a perfect **oracle**: any requested label is returned correctly. In practice this fails:

- Human annotators may be uncertain, guess, or abstain on hard cases
- Certain points may be structurally unlabellable (e.g. users behind a VPN)

LLMs as labellers: fast and cheap, but noisy (not ground truth). A practical approach: use the LLM as a weak labeller for most examples, reserve expensive human labels for cases where the LLM is uncertain.

Active Learning is Biased

Farquhar, Gal & Rainforth (2021): AL training data does not follow the population distribution (because we are deliberately *not* sampling randomly). This means:

- We want to predict over $\mathcal{P}$ but we train on $\mathcal{P}'$ — "optimising on the wrong objective"
- Can be corrected by applying inverse-probability weights to sampled datapoints
- But the bias can also *help*: it acts as a form of regularisation, especially useful for overparameterised models (deep neural networks)

Querying Strategies

There are many ways to operationalise "informativeness". They differ along the **exploration vs. exploitation** axis (borrowed from reinforcement learning):

- **Exploitation**: sample in regions model already knows well—confirm existing knowledge
- **Exploration**: sample in uncertain/unfamiliar regions—garner new knowledge

1. Random Sampling: $I()$ returns a random number. No model information is used. *Pros:* avoids statistical bias (still randomly representative); useful when the problem is purely about training volume. *Cons:* highly inefficient; no guarantee of informative samples; exploration/exploitation balance is 50/50 by chance.

2. Uncertainty Sampling: *Exploration strategy*: choose the observations the current model is most uncertain about.

1. **Binary classification** — uses binary entropy:

$$I() = -\hat{p}\ln\hat{p} - (1-\hat{p})\ln(1-\hat{p})$$

   Maximised when $\hat{p} = 0.5$ (model is a coin flip or the points near the decision boundary).

2. **Multi-category generalisation**: (sum of all classes)

$$I() = -\sum_k \hat{p}_k \ln\hat{p}_k \quad \text{where } \hat{p}_k \text{ is the predicted probability of class } k$$

Disadvantages:

- Focuses on "hard" cases near the decision boundary—may overfit on inherently noisy data
- Ignores the broader data distribution (highly myopic)
- Relies on the model's uncertainty estimates being well-calibrated — complex models (e.g. neural networks) can be confidently wrong

3. Query-by-Committee (QBC): Maintains a *committee* of models; queries the examples where committee members disagree most.

Algorithm:

1. Train C models on the same labelled data (each seeing a slightly different version)
2. Each model VOTES on the label for each pool observation. Sum up votes in favour/against.
3. Select observations with the largest **disagreement**

Composing a committee — how to get different hypotheses from the same data:

- **Bootstrap the data**: resample rows with replacement ⇒ each model sees a slightly different training set
- **Random feature subsets**: each model only sees a random subset of columns ⇒ forces different learned representations

Vote entropy (the disagreement measure): *sum over the classes*

$$I() = -\sum_k \left[\underbrace{\frac{\sum_c \mathbb{I}(g_c(\mathbf{x}) = k)}{C}}_{\text{proportion of models voting class } k} \log \underbrace{\frac{\sum_c \mathbb{I}(g_c(\mathbf{x}) = k)}{C}}_{\text{same proportion (inside log)}} \right]$$

where $\mathbb{I}(g_c(\mathbf{x}) = k) = 1$ if committee member c predicts class k for observation $\mathbf{x}$, and 0 otherwise. The numerator $\sum_c \mathbb{I}(g_c(\mathbf{x}) = k)$ counts how many of the C models voted for class k; dividing by C gives the vote share. This is maximised when votes are evenly split across classes (maximum disagreement) and equals 0 when all models agree. Note: each model performs a *hard* vote ($\hat{p} = 0.51$ and $\hat{p} = 0.99$ both count as 'voted class 1'), so predicted probabilities must be rounded before computing vote entropy.

Evaluating Active Learning

Need an evaluation to check if AL is actually working! $\Rightarrow$ **Learning curves**: plot model performance (accuracy, F1, AUC) against the number of labelled examples. (e.g. compare the AL strategy against a random sampling baseline).

- A good AL strategy should reach the same performance with *fewer* labels. The gap between AL–random sampling is your measure of **label efficiency**.
- Always evaluate on a **held-out test set** not part of the pool
- Repeat over **multiple random seeds** (AL performance can be sensitive to the initial labelled set)

Applied Example: Hate Speech Detection

Setup: 500,000 social media comments; classify each as hateful / not hateful.

Manual labelling is expensive, slow... $\Rightarrow$ Build a ML classifier:

1. **Initial labelled set D^0:** 200 comments labelled by an expert annotator (stratified to include hateful examples, rare class!)
2. **Pool $\mathbf{X}^P$:** remaining 499,800 unlabelled comments
3. **Target model:** logistic regression (fast at train and predict time)
4. **Querying strategy:** uncertainty sampling (binary entropy)
5. **Batch size:** $n = 50$ comments per round

Round 1: fit logistic regression on 200 comments; compute $I() = -\hat{p}\ln\hat{p} - (1-\hat{p})\ln(1-\hat{p})$ for all pool comments; select top 50 by entropy (comments near $\hat{p} = 0.5$, e.g. "These people should be locked up") and send to annotator.

Keep repeating the process. After 10 rounds: 700 labelled comments, 0.14% of corpus:

- Uncertainty sampling: F1 ≈ 0.88
- Random sampling (same budget): F1 ≈ 0.78
- To reach F1 $= 0.88$ with random sampling, we would need $\approx$ 2,000+ labels
- **Label efficiency: AL achieved same perfomance with $\approx 3\times$ fewer labels**

What is happening under the hood: each AL round shifts the estimated decision boundary closer to the true (nonlinear) boundary. The next round then queries near the *updated* boundary — so the model progressively corrects its own deficiencies.

Challenges in this example:

1. **Class imbalance:** Hate speech is $\approx$ 2–5% of comments. Uncertainty sampling helps: the model is most uncertain near the boundary, which is where the rare class lives.
2. **Oracle reliability:** Hate speech is subjective; annotators may disagree; sarcasm/irony make labelling hard. Multiple annotators + adjudication increases cost
3. **Concept drift:** What constitutes HS speech may shift. Model may need periodic retraining.

Week 9: Ensemble Methods

So far we have seen that individual decision trees are highly interpretable but poor predictors (high variance and do not generalize well). The solution is to **combine many trees into an ensemble.**

There are 3 ways to do this, from least to most sophisticated:
(1) bagging (bootstrap aggregation), (2) random forests (bagging + randomness in features) and (3) boosting (sequential adjustment on the residual).

(1)–(2) reduce variance in parallel; (3) builds the predictor iteratively. All use the idea of ensembles (a collection of models) and produce some of the best predictors in ML.

Bootstrap

Resampling method for estimating the sampling distribution of a statistic $\hat{\beta}_k \Rightarrow$ Treat the observed sample as a mini-population, and draw many samples *with replacement* from it.

How it works (3 steps):

1. From original sample n, draw a bootstrap sample of same size with replacement (some obs will appear > 1, others 0 times); re-estimate model with the new sample and record $\hat{\beta}^*_{k,b}$.
2. Repeat step 1 B times to get $\hat{\beta}^*_{k,1}, \ldots, \hat{\beta}^*_{k,B}$. $\Rightarrow$ You'll have B different estimates (slightly different from one another).
3. Compute the variance of those B estimates to get *Bootstrap SE* (measures how much $\hat{\beta}_k$ varies from a sample to the next). More conservative/realistic than those in small samples.

$$\widehat{\text{se}}_{\text{bootstrap}}(\hat{\beta}_k) = \sqrt{\frac{1}{B-1}\sum_{b=1}^{B}\left(\hat{\beta}^*_{k,b} - \bar{\hat{\beta}}^*_k\right)^2}, \qquad where \quad \bar{\hat{\beta}}^*_k = \frac{1}{B}\sum_{b=1}^{B}\hat{\beta}^*_{k,b}$$

Why with replacement? Sampling without replacement creates dependence across samples, violating the i.i.d. assumption. Also, the same size n is needed so standard error formulas (which depend on n) remain comparable.

Ensembles

Bootstrap samples provide multiple training sets from the same data $\Rightarrow$ **train a separate model on each bootstrap sample, then combine predictions** and get a better predictor. An **ensemble** is simply a collection of models (same or different types).

- **For a Regression task**: average predictions across models.
- **For a Classification task**: majority vote across models.

E.g. combine many trees $\Rightarrow$ through **BAGGING, RANDOM FORESTS, BOOSTING**. All options use the idea of 'ensembles' and produce the best ML predictors.

Bagging

General-purpose procedure for reducing the variance of a statistical learning method.

Un sol arbre de decisió té alta variança: si l'entrenes en dades lleugerament diferents, el resultat canvia. La idea del bagging és que si fas promig molts de diversos predictors, la variança és menor que la de cadascun per separat.

Why does averaging reduce variance? For n i.i.d. random variables $Z_1, \ldots, Z_n$ each with variance σ^2: $(\bar{Z}) = \frac{\sigma^2}{n} \rightarrow$ Averaging $\downarrow$ variance by a factor of n ($\uparrow$ models: final predictor varies less).

Procedure:

1. Generate B bootstrap training sets from the original data.
2. Train a separate model (e.g. a full, unpruned tree) on each bootstrap sample.
3. Combine predictions: average (regression) or majority vote (classification).

Out-of-Bag (OOB) Error Estimation

On average, each bootstrap sample uses approximately **2/3** of observations. → The remaining **∼1/3** are the **out-of-bag (OOB)** observations for that tree.

OOB error: for each observation i, $\approx B/3$ trees did *not* train on it. You can use these trees to predict and measure the error (free out-of-sample prediction for every training observation, equivalent to **leave-one-out cross-validation**). ⇒ No need for a separate validation set (OOB error is estimated automatically during training).

Random Forests

Improvements over bagging that further reduces variance by **decorrelating the trees** (+ some randomisation on features)

The problem with bagging: if one feature is overwhelmingly predictive (e.g. square metres in a house price model), most trees split on it first. Despite different bootstrap samples, the trees remain highly correlated with each other. Correlated trees ⇒ variance doesn't ↓ as fast.

Fix: random feature subsets: at each split, only consider a **random subset of** $m < p$ **features**. Given that subset, pick the best split as usual. This prevents the dominant predictor from always driving the first split.

- The subset is **re-drawn at every split** (not once per tree).
- Forces trees to be different across samples ⇒ **decorrelation** ⇒ additional variance ↓.

Choosing m:

1. Common baseline value for classification: $m \approx \sqrt{p}$ (p = no. features)
2. Regression: $m \approx p/3$
3. Alternatively Cross-validation

Boosting

Most competitive approach. Unlike bagging/RF, boosting is **sequential** and does **not rely on bootstrapping**. → Each tree is grown sequentially (using info for past trees). Each tree fits on the **residual/error** left without explanation by the previous ones.

Intuition (iteratively):

1. Fit the first tree on y (all outcome is error before starting the prediction) ⇒ compute residual $e_i = y_i - \hat{f}(x_i)$.
2. Fit the second tree on the residual (small, max 1-2 depth levels). ⇒ subtract a scaled version from it.
3. Third tree: fit on the remaining residual.
4. Combine all trees (weighted by λ) to get the aggregate prediction.

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}_b(x)$$

λ acts as a **learning rate** (analogous to gradient descent): subtracting a scaled version of the prediction prevents overfitting too quickly.

Note: this algorithm is for regression only. Boosting for classification is +complex (out of scope).

Differences:

	Bagging	**Boosting**
Order	Parallel	Sequential
Bootstrap	Yes	No (basic version)
Each tree trained on	Bootstrap sample of y	Residual of previous trees
Main risk	Low reduction if trees are correlated	Overfit if λ too large or B too high

Exam — Key Distinctions

Bagging vs. Random Forests: both use bootstrap samples + averaging. RF additionally restricts each split to a random subset of m features, further decorrelating trees and reducing variance. Bagging is the special case $m = p$.

Why random feature subsets help: if one predictor is overwhelmingly dominant, all trees split on it first $\Rightarrow$ high correlation across trees $\Rightarrow$ variance does not decrease much. Restricting to $m < p$ forces different splits across trees.

OOB error: each bootstrap sample uses $\sim 2/3$ of observations. The remaining $\sim 1/3$ (OOB) provide free out-of-sample evaluation, equivalent to LOO-CV for large B. No need for a separate valid. set.

Boosting does not use bootstrap (in the basic version): the ensemble comes from sequentially fitting residuals on the *same* dataset, not from parallel training on different bootstrap samples.

Feature importance $\neq$ causality: importance measures tell you what the model relies on, not what you can intervene on. 'Last login time' may be the strongest predictor of churn but is not a causal lever.

High cardinality bias in tree importance: continuous features offer more potential split points than low-cardinality categoricals $\Rightarrow$ may appear spuriously + important even if randomly generated.

Boosting hyperparameters you cannot ignore: learning rate (too large = overfitting; too small = underfitting), tree depth, and early stopping (prevents fitting residual to 0).

Week 10: Neural Networks

Simplest architecture: Single Layer Perceptron (SLP)

The simplest architecture is the **SLP**: a feedforward neural network with no hidden layer and an output layer with a single neuron and activation.

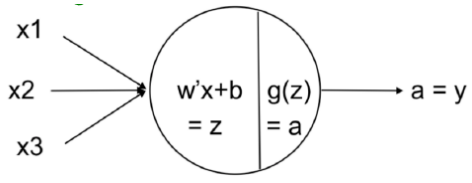


Figure 7: 3 features that we linearly combine

Given a p-dimensional input $\mathbf{x} = (x_1, x_2, x_3)$:

1. Compute the linear combination (the *pre-activation*):

$$z = \mathbf{w}'\mathbf{x} + b = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$$

   where $\mathbf{w} = (w_1, w_2, w_3)$ is the **weights** vector (slopes) and b is the **bias**.

2. Apply an **activation function** $g(\cdot)$: $a = g(z)$
   There's multiple activation functions: With sigmoid activation $g(z) = \dfrac{1}{1+e^{-z}}$, the SLP is equivalent to **logistic regression**.
3. Output: $a = y$.

Limitation. The SLP has a single neuron: it computes $z = \mathbf{w}'\mathbf{x} + b$, applies $g(z)$, and outputs a prediction. The problem is that with a single neuron, the only decision boundary it can learn is a straight line — it only handles linearly separable data.

Multilayer Perceptron (MLP)

The **MLP** solves SLP's limitations by adding **hidden layers**: rows of neurons between the input and the output. Each neuron does exactly the same thing as the SLP ($z = \mathbf{w}'\mathbf{x} + b$, then $g(z)$), but now the output of one layer becomes the input to the next. This chaining of non-linear transformations allows the network to learn decision boundaries of any shape.

Why does adding layers help? Each hidden layer learns a different representation of the inputs. The first layer may learn simple patterns, the second combines them into more complex ones, and so on.
Crucially, **activations are essential**: stacking only linear layers collapses back to a single linear transformation. Non-linear activations (e.g. ReLU) are what give the MLP its flexibility.

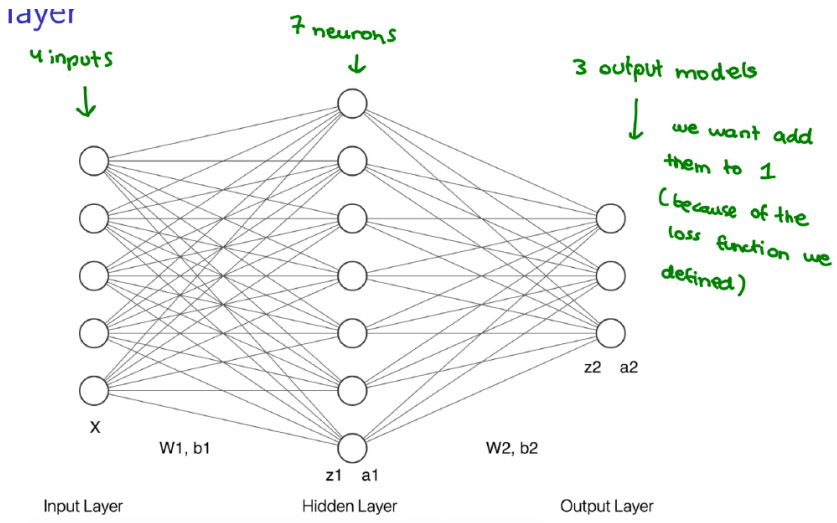


Figure 8: Example MLP: one hidden layer, 4 inputs, 7 neurons, 3 outputs

Forward pass. Information flows forward through the network layer by layer:

$$\mathbf{x} \xrightarrow{W^{(1)}, b^{(1)}} z^{(1)} \xrightarrow{g(\cdot)} a^{(1)} \xrightarrow{W^{(2)}, b^{(2)}} z^{(2)} \xrightarrow{s(\cdot)} a^{(2)} = \mathbf{p}$$

Each arrow is one step: a linear transformation (matrix multiplication + bias), followed by an activation. For a network with 5 inputs, 7 hidden neurons, and 3 outputs:

e.g. dimensions tell you the no. of parameters: $W^{(1)}$ has $7 \times 5 = 35$ weights.

Regression and Classification

Neural networks can be used for both regression and classification. The output activation differs:

Task	Output activation	Formula
Regression	Linear	$g(z) = z$
Binary classification	Sigmoid (comprises to 0-1)	$g(z) = \dfrac{1}{1+e^{-z}}$
Multi-class classification	Softmax (as Sigmoid, but for diff classes)	$s(z)_i = \dfrac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}$

Table 2: Output activation by task. Softmax transforms outputs to probabilities that sum to 1.

Loss Functions

We minimise a loss function via gradient descent. Common choices:

- **Mean Squared Error** (regression):

$$L(y_i, p_i) = (y_i - p_i)^2$$

- **Cross-entropy** (classification, K classes):

$$L(y_i, p_i) = -\sum_{k=1}^{K} y_{i,k} \log(p_{i,k})$$

The full cost over the training dataset with parameters θ (all weights and biases stacked into one 1-dimensional vector): *la corba que volem minimitzar amb gradient descent*

$$J(\theta;\, \{x_i, y_i\}_{i=1}^{n}) = \frac{1}{n} \sum_{i=1}^{n} L(y_i,\, p(x_i, \theta))$$

To perform gradient descent we need derivative $\nabla_\theta J(\theta)$. Applying the chain rule to the forward pass $x \to z^{(1)} \to a^{(1)} \to z^{(2)} \to a^{(2)} = p$:

$$\frac{\partial J}{\partial W^{(2)}} = \frac{\partial J}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}}$$

$$\frac{\partial J}{\partial W^{(1)}} = \frac{\partial J}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(1)}}$$

Mini-Batch Gradient Descent

Computing the gradient over the full dataset (*full batch*) is computationally too costly. Instead:

1. Randomly initialise weights (small random numbers; **0** would not work); select learning rate α.

2. Repeat for E epochs or until convergence:
   a. Shuffle all observations and divide into $\lceil n/B \rceil$ batches.
   b. For each batch of B observations, update:

$$\theta \leftarrow \theta - \alpha \cdot \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta L(y_i, p(x_i, \theta))$$

   (compute average derivative for that batch and update it. When done with all batches, continue with next epoch)

Regularisation

Neural networks have large numbers of parameters and can memorise training data easily. Common approaches to counter over-fitting:

- **L1 / L2 norms** added to the loss function ⇒ makes all weights and biases smaller.
- **Dropout**: randomly set a fraction of neurons in a layer to 0 *during training*, forcing varying sets of neurons to learn patterns. Named 'training an ensemble' (avoids overfitting).
- **Early stopping**: stop gradient descent once the loss on a validation set starts increasing.

Improving Training

- **Hyperparameter tuning:** hidden layers, activations, learning rate, etc.
- **Batch/layer normalisation:** batch norm normalises over observations in a batch; layer norm normalises over features within an observation.

Common Variants

Convolutional Neural Networks (CNNs)

Typically used in **computer vision**, but can also be applied to other tasks. Instead of fully connected layers, CNNs learn **filters** and combine operations such as:

- **Convolution**: slide a small filter (e.g. 3×3) across the image, computing a dot product at each position ⇒ produces a *feature map*.
- **Max pooling**: divide the feature map into regions (e.g. 2×2) and keep the maximum value in each ⇒ reduces spatial dimensions.

Filters are *learned* via gradient descent (e.g. one filter may learn to detect edges, another to detect blurs). A typical CNN architecture alternates convolution and pooling layers, ending with dense (fully-connected) layers that compress the feature maps into a single prediction vector.

Large Language Models (LLMs)

LLMs are neural networks (transformers) that predict the probability of the next token given a sequence of input tokens:

$$\Pr(x_{T+1} \mid x_T, x_{T-1}, \ldots, x_1)$$