Introduction to Machine Learning
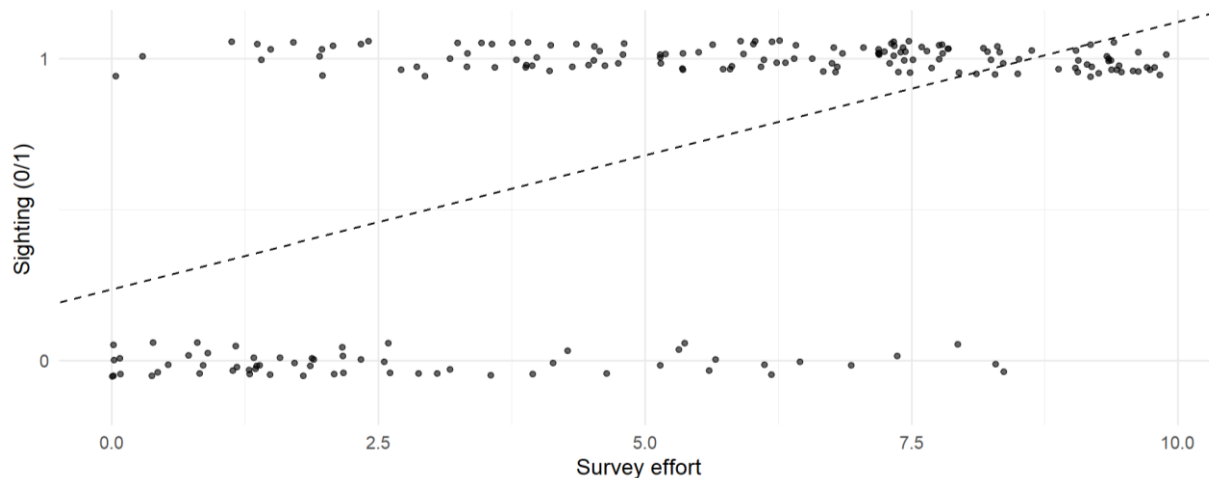
Dr Niamh Mimnagh

niamh@prstats.org

https://github.com/niamhmimnagh/IMLR04-Introduction-to-Machine-Learning

Modelling Binary Data

- If we try to fit a straight line to this data, we are asking the line to directly explain the observed successes and failures.

- This is saying: "the response itself (0 or 1) changes linearly with effort." But this does not make sense: the outcome is discrete, so the line will not pass through the points in a meaningful way.

- What we really care about is not the individual 0s and 1s, but the chance of seeing an animal given the level of effort.

Modelling Binary Data

- Rather than trying to predict the raw binary outcomes directly, we aim to model the probability of a sighting.

- This shifts the focus from "can we exactly predict each 0 or 1" to "can we explain the underlying chance of success."

- Probabilities are continuous between 0 and 1, which allows us to draw a smooth curve that describes how the chance of success changes with effort.

- A straight line still does not work on the probability scale (because it can go below 0 or above 1), but this gives us the right target: we want to model the probability, not the binary outcome itself.
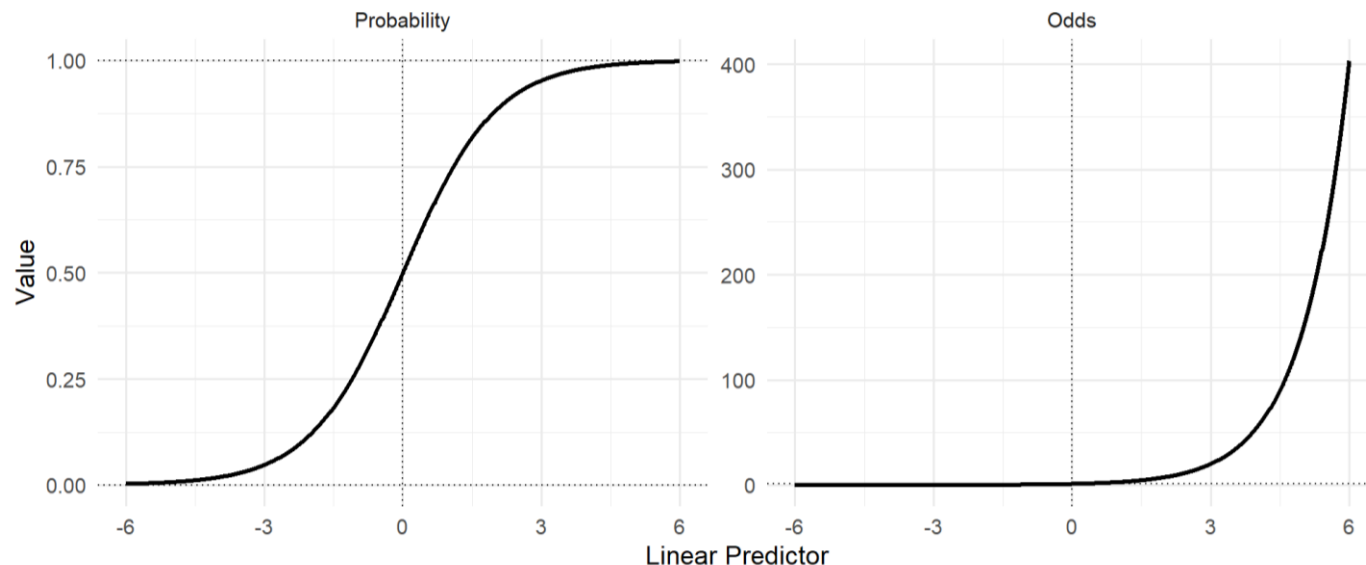
Logistic Regression

$$Y_i \sim Bernoulli(\pi_i)$$

- π_i is the probability, and it describe the chance of success as a number between 0 and 1.

- Now we want to use our predictor variables to model the probability of success, but the linear predictor $\beta_0 + \beta_1 x$ can take any real value.

- If we directly equate probability with a linear predictor ($\pi_i = \beta_0 + \beta_1 x$), the model could easily predict probabilities less than 0 or greater than 1, which makes no sense.

- We need to transform the probability scale (0 to 1) onto the whole real line, so that the linear predictor can work properly.
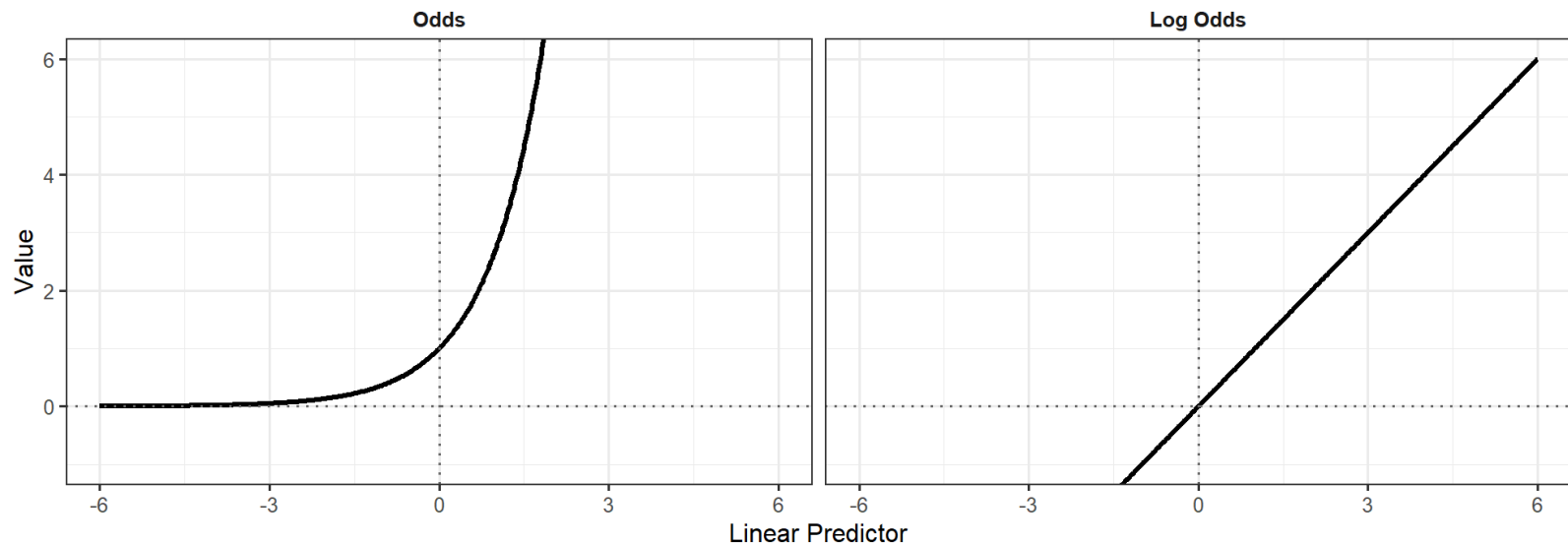
Logistic Regression

How do we do this?

If instead of modelling probability, we model the odds ($\frac{\pi}{1-\pi}$), this gets us partway there:

Logistic Regression

However, the odds still don't allow the linear predictor to be negative so we take the log(odds), so we model using $\log(\frac{\pi_i}{1-\pi_1}) = \beta_0 + \beta_1 x$

Interpreting Coefficients

- Raw coefficients are interpreted on the log-odds scale – a one unit increase in the predictor adds β_1 to the log-odds of the response.

- Taking the exponential of the raw coefficients gives the odds – a one unit increase in the predictor multiplies the odds of the response by e^{β_1}.

Logistic vs. Linear Regression

- Linear regression
  - Predicts a continuous outcome (e.g., height, income).
  - Model: $y = \beta_0 + \beta_1 x + \cdots + \epsilon$
  - Predictions are unbounded $(-\infty, \infty)$
- Logistic regression
  - Predicts a probability for a binary outcome (e.g., spam vs not).
  - Model the log-odds: $logit(p) = \beta_0 + \beta_1 x + \cdots \rightarrow$ always in [0,1].
  - Choose a threshold (often not 0.5) to convert p to a class.

Fitting a Logistic Model in R

- Outcome must be a 2-level factor; set the positive class explicitly (last level by default in yardstick).

- Preprocess with a recipe; normalisation isn't required for glm but is tidy and future-proof.

- Use a workflow to bind preprocessing and model specification steps.

- Predict probabilities and then classes

```
log_reg_spec <- logistic_reg() %>%
set_engine("glm") %>%          # base R glm
set_mode("classification")    # classification task
```

Evaluating Logistic Models

- Confusion matrix: counts of True positive predictions (TP), false positive predictions (FP), true negative predictions (TN), and false negative predictions (FN) (at a chosen threshold).

- Accuracy $= \dfrac{TP+TN}{N}$

- Precision $= \dfrac{TP}{TP+FP}$

- Recall $= \dfrac{TP}{TP+FN}$

- F1 $= \dfrac{2(Precision)(Recall)}{Precision+Recall}$

Predicting On A Test Set

- Step 1 – Extract predicted probabilities for each observation

- Step 2 – Pick a threshold t:

- Default $t = 0.5$ only makes sense when classes are balanced and FP/FN costs are similar. A better choice is to choose t to maximise a metric.

- Step 3 – Decision rule: predict positive if probability is greater than the threshold, and negative otherwise.

```
prob <- predict(model_fit, test_data, type = "prob")
pred_class <- ifelse(prob > 0.5, "spam", "not_spam")
```

Email Spam Classification

- The DAAG::spam7 dataset is a compact spam-filter dataset

- Our goal is to predict whether an email is spam.

  - yesno — target label (yes = spam, no = not spam).

  - crl.tot — total capital run length (sum of lengths of consecutive capital-letter runs).

  - dollar — frequency of the "$" symbol in the email.

  - bang — frequency of "!".

  - money — frequency of the word "money".

  - n000 — frequency of the pattern "000".

  - make — frequency of the word "make".

MAKE $$$ MONEY NOW!!!

$$ 10,000 GUARANTEED $$

MAKE YOU RICH FAST!!!

PR STATS

Introduction to Machine Learning
Dr Niamh Mimnagh, PR Statistics

Limitations in Logistic Regression

- Linearity in the log-odds
  - Assumes each predictor has an additive, linear effect on the logit. Real signals can be curved or interact

- Sensitive to multicollinearity
  - Highly correlated predictors inflate SEs, make signs unstable, and muddle interpretation (though predictions may still be ok).

- Separation issues
  - If a combo of predictors perfectly splits classes, MLE coefficients blow up ($|\beta| \rightarrow \infty$), warnings like "probabilities 0 or 1".

- Class imbalance & thresholding
  - Accuracy can look great while minority recall is awful; the default 0.5 threshold is rarely optimal.

Multiclass Logistic Regression

- In many applications, the outcome variable can take on more than two categories, and these categories are not ordered (e.g., animal = cat/dog/horse).

- Binomial logistic regression cannot be used when the outcome has more than two unordered categories. In such cases, we use multinomial logistic regression.

- Multinomial logistic regression models the probability of each possible outcome as a function of predictor variables.

- Each outcome is treated as a distinct category, and we model the log-odds of each category relative to a chosen baseline category.

Multiclass Logistic Regression

- In a scenario with K classes ($K \geq 3$), the goal is to model the probability that the response is in class K.

- In the binomial case, we need $\log\left(\frac{\pi}{1-\pi}\right)$ to map (0,1) onto the real line. In the multinomial case, we have a similar need to map probabilities to the real line, but now we have multiple probabilities, and they must sum to 1.

- In the binomial case, π is the probability of success and $(1-\pi)$ is the probability of failure. In the multinomial however, we no longer have probabilities for success and failure, we now have π_A (the probability of being in class A), π_B (the probability of being in class B), π_C (the probability of being in class C) etc.

Multiclass Logistic Regression

- What if we compute $\log\left(\frac{\pi}{1-\pi}\right)$ for each class separately? i.e., modelling the log of the probability of 'successfully' being in class A divided by the probability of failure (not being in class A).

- This would map (0,1) to the real line successfully for each class

- However, this is essentially fitting K entirely separate models to obtain the probability of success and failure for each class.

- There is no restriction here that would ensure $\pi_A + \pi_B + \pi_C = 1$

- We could estimate that $\pi_A = 0.7, \pi_B = 0.4$ and $\pi_C = 0.2$

- If the only options for classes are A, B and C, then the sum of their probabilities must be 1, so this is not the solution.

Multiclass Logistic Regression

- Instead, we must pick a baseline (reference) class, and for all other classes k, we compute the log-odds of being in class k rather than the baseline class.

- If we choose class A as the baseline, then we model using:
$$Y_i \sim Categorical(\pi_i)$$
$$\log\left(\frac{\pi_B}{\pi_A}\right) = \beta_{0B} + \beta_{1B}x$$
$$\log\left(\frac{\pi_C}{\pi_A}\right) = \beta_{0C} + \beta_{1C}x$$

- We can then convert back from log odds to probability using the softmax function, which ensures the probabilities are between 0 and 1 and sum to 1. Each coefficient for class k tells you how a predictor shifts the log-odds of k vs. A.

Fitting a Multinomial Model

- Outcome must be a factor with 3+ levels (first level is the baseline; it only affects coefficient labels, not probabilities).

- Preprocess with a recipe (encode/scale predictors).

- ```
  multi_spec <- multinom_reg() %>%
  ```
- ```
    set_engine("nnet") %>%      # or: set_engine("glmnet")
  ```
- ```
    set_mode("classification")
  ```

Example: Iris

- The Iris dataset has data on 150 flowers (3 species)
- It has four numeric predictors:
  - Sepal.length
  - Sepal.width
  - Petal.length
  - Petal.width
- The outcome is species type, and it has three unordered classes: setosa, versicolor and virginica
- We have more than two classes here, and no natural ordering, so we will use a multiclass logistic model.

Ordinal Classification

- What if we have categories that have a natural order? (e.g., Low < Medium < High)

- We want to predict the probability that an observation lies within each category, while respecting the order.

- If we use a standard multiclass model, it ignores the grouping. This means throwing away information, which can be less data-efficient.

- Now, the challenge is that the probabilities must be in [0,1], and sum to 1, and the order must be encoded.

Ordinal Classification

- We can think of the categories as steps on a ladder. Between each pair of steps there is a boundary.

- We can then ask the model a series of cumulative yes/no questions:
  - Is the outcome at most level 1?
  - Is it at most level 2?
  - …

- These questions are nested. If its less than level 1 then it is also less than level 2, so they naturally encode each other.

Ordinal Classification

- Ordinal regression is built on the idea that there is a continuous unobserved variable that gets 'cut' into categories.

- The model estimates:
    - How predictions shift the latent variable (slopes)
    - Where the category boundaries (thresholds/cutpoints) lie

- The model learns where those boundaries sit for a population.

- A higher cutpoint at a boundary means it is harder to cross into the higher category.

Ordinal Classification

- For ordinal Y with K categories: $Y \in \{1, 2, \ldots, K), 1 < 2 < \cdots < K$

- We want to model cumulative probabilities:
$$P(Y \leq k), \qquad k = 1, \ldots, K - 1$$

- A binary logistic regression models 'success' vs 'failure.'

- Here we create a series of binary splits: 'At or below category k' vs. 'Above category k,' and we model:
$$\log(\frac{P(Y \leq k)}{P(Y > k)} = \alpha_k - \beta_1 x$$

- This preserves the ordering, rather than fitting $K - 1$ unrelated models.

Ordinal Classification

$$\log(\frac{P(Y \leq k)}{P(Y > k)} = \alpha_k - \beta_1 x$$

- α_k : threshold/cutpoint for category k. This is the point where we cross from category k into category $k + 1$

- Latent severity scale   ----|---------|-------------|------------------

$$\alpha_1 \qquad \alpha_2 \qquad\qquad \alpha_3$$

none   mild   moderate   severe

- β_1: predictor effects are assumed to be the same across all thresholds. A positive effect shifts the latent severity scale to the right, increasing the probability of higher categories.

- For example, if $\beta_1 = 0.7$, for every 1 unit increase in x, the odds of being in a higher category (vs. being in category k or below) is multiplied by $e^{0.7}$.

Modelling Ordinal Data

Make the outcome an ordered factor with levels in the correct order (lowest → highest).

- Engines you can use
  - set_engine("polr") → MASS::polr (basic, fast).
  - set_engine("ordinal") → ordinal::clm (often better diagnostics/options).

```
ord_spec <- proportional_odds(mode = "classification") %>%
set_engine("MASS")
```

Example: Arthritis Treatment

- The vcd::Arthritis dataset contains patient-level data from a clinical study of rheumatoid arthritis treatment.

- Our goal is to model patient improvement after treatment as an ordinal outcome.

  - ID - patient identifier
  - Treatment - Treated vs Placebo.
  - Sex - Male / Female.
  - Age - age in years (numeric).
  - Improved - clinician-rated improvement with an order: None < Some < Marked.

Classification Trees

- A classification tree splits the predictor space into distinct regions by asking a series of binary questions.

- Each terminal node is assigned the most common class among the training observations that fall into it.

- Trees learn split rules based on minimising impurity.

- At any node you've got training cases. Some are class A, some B, maybe C.

- A good split makes two children that are purer (more one-class) than the parent.

- Think: jars of marbles. A jar that's all blue (one class) is pure. A jar that's 50/50 blue/red is impure. Impurity measures are just different ways to score "how mixed is this jar?".

Splitting Criteria

- Gini index (mixing score)
  - Pick two cases at random without looking. The Gini index is the chance they're from different classes.
  - The more mixed the class labels are inside the child node, the higher the Gini index.
- Entropy (uncertainty score)
  - Pick one case from a node and try to guess its class. You are allowed to ask 'yes/no' questions like: 'is it class A?' 'is it A or B?'
  - If the classes in the node are evenly mixed you will need more questions to figure it out. Entropy is the number of questions on average needed to identify the class. If classes are more mixed, it's a harder guessing game, which means higher entropy.

Fitting a Classification Tree

```
tree_spec <- decision_tree(
  cost_complexity = 0.001,   # pruning strength (cp)
  tree_depth      = 10,      # max depth
  min_n           = 10       # min samples to split
) %>%
set_engine("rpart", parms = list(split = "gini"/"information")) %>%
set_mode("classification")
```

Pruning the Tree

- Trees can overfit. If you keep splitting, a tree can perfectly memorise quirks of the training data (tiny, very pure leaves).

- This means that the training accuracy will look great, but test accuracy will drop.

- Pruning deliberately makes the tree smaller/simpler to improve generalisation on the test set.

- In tidymodels this is done via the cost_complexity option. While growing the tree, if a split doesn't improve fit by at least this value, it is not made.

  - cp ≈ 0 to 1e-4: essentially no penalty → very large trees; high variance / risk of overfit.
  - cp ≈ 1e-3 to 1e-2: often a good range on many tabular problems (but data-dependent).
  - cp ≈ 0.05–0.2: aggressive pruning → very shallow tree; can underfit.
  - cp ≥ 0.3: commonly yields a stump (one split) or even the root only.

Advantages of Classification Trees

- Highly interpretable
    - White-box rules: "If bang > 0.3 and dollar <= 0.1 → spam."
    - Easy to visualise and explain to non-technical audiences.
- Work with many data types
    - Handle numeric and categorical predictors natively.
    - Units and scaling don't matter (no standardisation needed).
- Naturally capture interactions and nonlinearity
    - Splits create piecewise regions; interactions emerge automatically (e.g., bang interacts with dollar if they appear on different branches).

Limitations of Trees

- Prone to overfitting

- High variance / instability

- Small data changes can flip early splits which can lead to very different trees.

- Greedy search – Chooses the best split locally and can miss better global structures.

- Impurity criteria can ignore rare classes; leaves predict the majority.

- Missing data handling can be brittle

Random Forests

- A random forest is an ensemble of decision trees built on bootstrapped samples of the data.

- Two sources of randomness reduce variance:
  - Bootstrapping rows: each tree sees a different sample
  - Subsampling columns: at each split, consider only a random subset of predictors. This means trees are decorrelated.

- In a classification setting, the prediction is the majority vote. Averaging many diverse but competent trees stabilises predictions and combats overfitting.

Fitting Random Forests for Classification

```
rf_spec <- rand_forest(
  mtry  = 5,     # number of predictors sampled at each split
  trees = 1000,  # number of trees
  min_n = 5) %>% # minimum n to split a node
set_engine("ranger", importance = "impurity") %>%
  set_mode("classification")
```

Comparing Tree and Forest Accuracy

- Bias–variance trade-off
  - Tree: low bias, high variance → small data edits can change splits a lot.
  - Forest: averages many de-correlated trees → much lower variance, usually higher accuracy.
- Interpretability
  - Tree: human-readable if/else rules; easy to explain.
  - Forest: less transparent; use permutation importance, SHAP, or a surrogate tree to explain.

K-Nearest Neighbours

- k-NN is a non-parametric algorithm that assigns a label to a new point based on the majority class of its k nearest neighbours in the training set.
    1. Compute distances from the new point to all training points.
    2. Take the k smallest distances (its neighbours).
    3. Vote: majority label (or weight by 1/distance).

- Choosing k
    - Small k → flexible, can overfit/noisy.
    - Large k → smoother, can underfit/blur class edges.
    - Pick k via cross-validation; often an odd k avoids ties.

Distance Metrics in kNN

- Euclidean distance: $\sqrt{((x_1 - y_1)^2 + ... + (x_n - y_n)^2)}$
  - Smooth, round neighbours.
  - Default for many kNNs
  - Sensitive to scaling - requires standardising features.

- Manhattan distance: $|x_1 - y_1| + ... + |x_n - y_n|$
  - Diamond-shaped neighbours
  - A bit more robust to outliers

Introduction to Machine Learning
 Dr Niamh Mimnagh, PR Statistics

Feature Scaling for kNN

- k-NN uses distances. A feature measured in large units can dominate Euclidean/Manhattan distance and drown out smaller-scale features.

- We must put all numeric predictors on a comparable scale so each can contribute fairly.

- What to use:
  - Z-score standardisation (mean 0, sd 1): default and most common.
  - Min–max scaling (0–1): good for bounded ranges.
  - Robust transforms (Yeo-Johnson/Box-Cox): help when skewed/outliers.

- Use preprocessing steps like:
- ```
  recipe(..., data = ...) %>%
  step_normalize(all_numeric_predictors())
  ```

Fitting kNN with tidymodels

```
# Recipe — scale numerics
rec <- recipe(Class ~ ., data = train) %>%
  step_zv(all_predictors()) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_normalize(all_numeric_predictors())


# Model spec
knn_spec <- nearest_neighbor(neighbors = 5) %>%
  set_engine("kknn") %>%
  set_mode("classification")
```
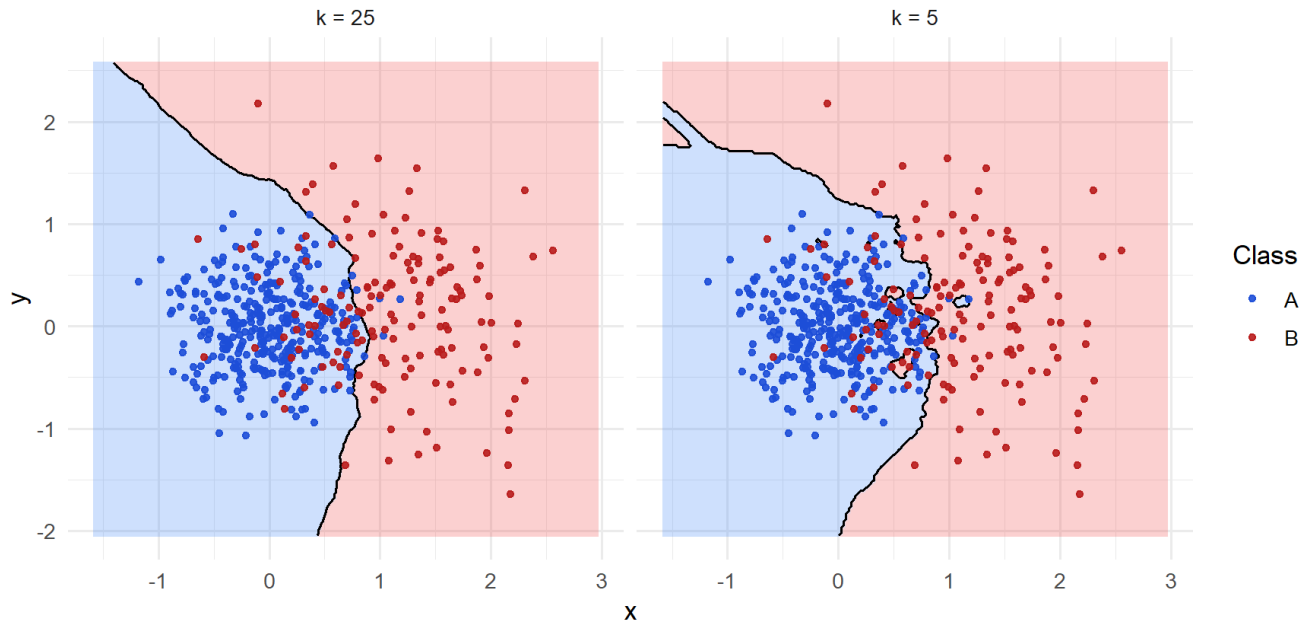
Evaluating kNN

- When it performs well
  - Local, smooth boundaries with nearby same-class points.
  - Low–moderate dimension with mostly informative features.
  - Plenty of training data near each test point.
- Common failure modes
  - Irrelevant/noisy predictors: distances get polluted ("curse of dimensionality").
    Fix: feature selection, PCA, or stronger regularisation via larger k.
  - Class imbalance: majority class dominates neighbourhoods.
    Fix: resample (SMOTE/upsample), class weights (if available), balanced accuracy/PR AUC, tune decision threshold.

Visualising Decision Boundaries

Naïve Bayes

- Naïve Bayes is a classifier that treats each feature as a separate piece of evidence about the class.

- It starts with a prior belief for each class (how common each class is in the training data).

- For every feature, it asks: "If this case were class k, how typical is its feature value?" and turns that answer into a small evidence score.

- It adds up the evidence from all features plus the prior for each class, and picks the class with the largest total.

- For each class k, Naïve Bayes scores an observation by:
  $(prior\ belief\ in\ class\ k)(how\ likely\ the\ features\ are\ if\ class\ k\ were\ true)$

Types of Naïve Bayes Models

- How do we calculate how likely the features would be if class k were true?

- It depends on what type of features we have.
  - For continuous features we use Gaussian Naïve Bayes
  - For binary features we use Bernoulli Naïve Bayes
  - For count features we use Multinomial Naïve Bayes

- The method is called "naïve" because it assumes the features contribute independently once you know the class. Because of the naïve independence assumption, we can mix types: treat numeric variables with Gaussian, counts with Multinomial and binaries with Bernoulli, and multiply the (per-feature) likelihoods for a class.

Strengths of Naïve Bayes

- Fast to train
  - "Training" is just counting/averaging per class (priors + per-feature stats) so the algorithms is linear in data size; scales to millions of rows and huge vocabularies.

- Tolerant of many irrelevant features
  - Under the independence assumption, extra weak features mostly contribute tiny, near-zero log-likelihoods; they don't swamp the signal the way distance-based models (e.g., k-NN) can.

- Works well with small datasets
  - Few parameters per feature (just rates/means/variances) which leads to low variance and good performance when data are limited.

Limitations of Naïve Bayes

- Independence assumption (often false)
  - Correlated features "double-count" evidence which leads to overconfident posteriors and unstable feature effects.
- Distributional mismatch for numeric features
  - Gaussian NB assumes class-conditional normality
- Can't model interactions or context
  - "FREE" and "$" together might be more than the sum. NB can't learn that.
- Imbalanced classes
  - Priors reflect class frequencies; minority can be swamped

Using kNN vs Naïve Bayes

- Use k-NN when:
  - You expect non-linear, local decision boundaries (clusters).
  - You can standardise features and keep irrelevant features to a minimum.
  - Feature space is low–moderate dimensional (dozens, not thousands).
  - You can afford slower prediction (distance to all training points).
- Use Naïve Bayes when:
  - You need speed (training and prediction) and tiny memory footprint.
  - Data are high-dimensional and sparse
  - Simple per-feature evidence is sensible (tokens/flags/frequencies).

Evaluating Classification Models

$$\text{Accuracy} = \frac{TP + TN}{N}$$

- Accuracy can be useful but also misleading.

  - If classes are imbalanced, a model that always predicts the majority class will have high accuracy but will have no ability to discern one class from another.

  - Accuracy is threshold-dependent. Changing the cutoff (e.g., 0.5 to 0.3) can cause accuracy to swing wildly.

  - Accuracy cannot deal with unequal costs: missing a case of cancer/fraud is worse than a false alarm. However, accuracy treats them the same.

Precision, Recall and F1

$$\text{Precision} = \frac{TP}{TP+FP}$$

- Precision answers the question: 'of the positive predictions, how many were actually positive?'

$$\text{Recall} = \frac{TP}{TP+FN}$$

- Recall answers the question: 'of the real positives, how many did I catch?'

- High precision means few false positives. High recall means few false negatives.

$$\text{F1} = \frac{2(Precision)(Recall)}{Precision+Recall}$$

- F1 is the harmonic mean of the precision and the recall. It is only high if both precision and recall are high. Its useful for imbalanced data and when the cost of false positives and false negatives are similar.

ROC Curve and AUC

- ROC curve:
  - Plots True Positive Rate (TPR / Recall) vs False Positive Rate (FPR) as you sweep the decision threshold from 0 to 1.

- AUC (Area Under the ROC Curve):
  - Probability a random positive is scored higher than a random negative.
  - AUC = 0.5 → random guessing;
  - AUC = 1.0 → perfect ranking.

- Reading the curve: upper-left is best (high TPR, low FPR). A model dominates another if its ROC is everywhere above.
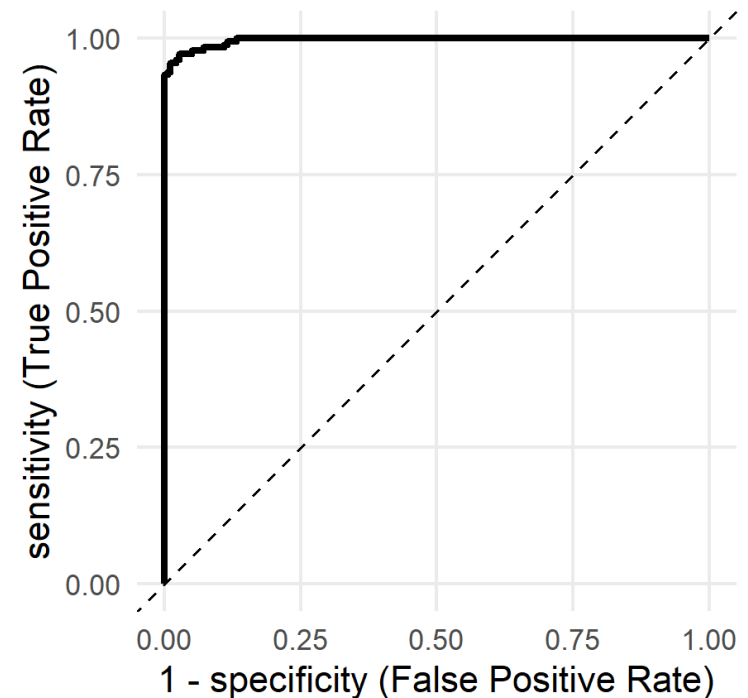
Precision-Recall Curve

- A precision-recall curve plots precision vs recall as the probability threshold varies.

- These are more informative than ROC when class imbalance is present.

- AUPRC: area under the PR curve

- Curves nearer the top-right are better; pick a threshold that gives the precision-recall trade-off you can live with.
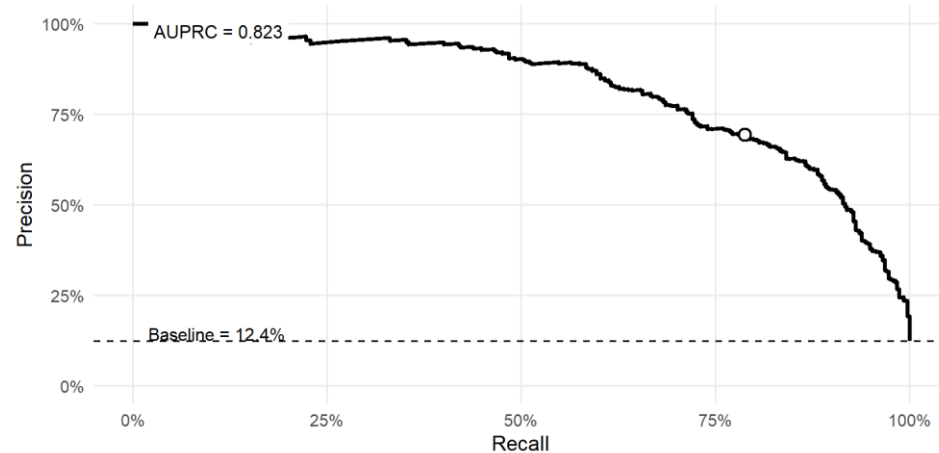
Imbalanced Data

- Imbalanced data occurs when one class is much rarer than the other(s), so the prevalence (positive rate) is low (e.g., 0.1-10%)

- Examples:
  - Fraud detection
  - Medical diagnosis
  - Rare species identification

- A model can get high accuracy by predicting the majority class.

- This risks biased decisions: missed positives (false negatives) where they matter the most.

Metrics for Imbalanced Data

- Precision asks: 'of the items we flagged as positive, how many were truly positive?'
  - It ignores true negatives, so a huge majority class can't inflate it. If you raise lots of false alarms, precision drops.
  - Increasing the probability threshold usually raises precision (you keep only the most confident alerts) but lowers recall.

- Recall asks: 'of all true positives, how many did we find?'
  - Recall also ignores true negatives, so it measures coverage of the minority class directly.
  - Lowering the threshold usually raises recall (you catch more positives) but can hurt precision

Oversampling

Oversampling is a technique for handling imbalanced datasets by increasing the size of the minority class so it's closer to the size of the majority class.

How it works:

- Identify the majority class (e.g., 990 negatives) and the minority class (e.g., 10 positives).
- Duplicate minority-class samples until the dataset is more balanced.
- Train the model on this larger, more balanced dataset, where the minority class has a stronger presence.

Advantages	Limitations
Keeps all original data	Risk of overfitting – duplicating minority examples can make the model memorise them
Helps the model learn the patterns in the minority class	Increases training time

Undersampling

- Undersampling handles imbalanced data by reducing the size of the majority class so it's closer to the size of the minority class.

- How it works:
  - Identify the majority class (e.g., 990 negatives) and the minority class (e.g., 10 positives).
  - Randomly remove a subset of majority class samples so that the two classes become more balanced.
  - Train the model on this smaller, more balanced dataset so it pays more attention to the minority class.

| Advantages	Limitations
Fast and simple to implement	Potential information loss when discarding majority-class samples
Reduces training time	Can lead to underfitting
	May not work well if the dataset is already small

SMOTE

- SMOTE creates synthetic minority-class samples instead of simply duplicating existing ones.

How it works:

- For each minority-class example, find its k-nearest neighbours in the minority class.
- Randomly pick one neighbour and interpolate a new synthetic sample between the two points.
- Repeat this process until the minority class reaches the desired size.

Advantages	Limitations
Reduces overfitting compared to simple oversampling	Can create overlapping classes
Creates more diverse minority samples	May introduce unrealistic samples
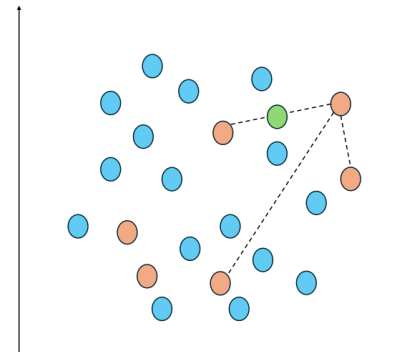
Class Weights and Cost-Sensitive Learning

- Using class weights makes mistakes on the minority class more important during training.

- Each observation's loss is multiplied by a weight which shifts the decision boundary toward the majority (to catch more minority cases)

- This often mimics what you'd get from lowering the threshold, but it's learned during fitting.

- In tidymodels, class weights can be incorporated in: random forests, decision trees, boosted trees, logistic regression, naïve Bayes.

- A common first choice for weights is the inverse frequency of the class occurrence.

When to Use Them

- Try class weights first when your model supports them (logistic/GLM, SVM, trees/forests/GBMs). They penalise mistakes on the minority class without duplicating data and are easy to tune with CV.

- Random undersampling when the dataset is very large (training is slow/memory-heavy).

- Random oversampling when the minority class is small and you need more positive examples for the learner to "see."

- SMOTE when features are mostly numeric and the minority class is sparse but has structure.

- Extreme imbalance (<0.5% positives): consider anomaly/one-class methods or cost-sensitive learning combined with undersampling

Evaluating After Resampling

- Do not over/undersample the whole dataset before splitting the data. If you do, duplicated synthetic rows can leak from train into validation/test and inflate metrics.

- Put rebalancing in the recipe() so it's applied to the analysis (training) slice of each fold only. The assessment slice remains untouched, and metrics will reflect real class balance.

- Use the right metrics for imbalance and pick a probability threshold on CV predictions—then lock it before testing.

- Perform the final check on untouched test set only once. Fit the finalised model on all training data (with rebalancing inside) and evaluate on the original test split.