Introduction to Machine Learning

Dr Niamh Mimnagh

niamh@prstats.org

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

PR STATS

What is Machine Learning?

- Machine learning is the study of algorithms that improve their performance at some task through experience.
- Unlike rule-based programming, where every instruction must be explicitly coded, machine learning systems adapt based on data. Instead of being told exactly how to solve a task, they discover rules and patterns for themselves.
- The more data and feedback an algorithm receives, the better it can refine its predictions or decisions. This makes machine learning especially powerful for complex problems where writing explicit rules would be impractical or impossible.

A Definition

Machine learning is a computer program said to learn from experience ‘E’ with respect to some class of tasks ‘T’ and performance measure ‘P’, if its performance at tasks in ‘T’, as measured by ‘P’, improves with experience ‘E’ - Tom Mitchell (1997)

Machine Learning vs. Traditional Statistics

- Statistics has a focus on understanding
  - Focuses on inference, quantifying uncertainty, and identifying relationships between variables.
- Machine Learning has a focus on prediction
  - Focuses on predictive accuracy and the ability to generalise to new, unseen data.
- Many models appear in both fields (e.g., regression, classification).
- The distinction is often in emphasis: statistics seeks explanation, machine learning seeks performance.
- In practice, both perspectives complement each other—understanding relationships can improve predictions, and predictive models can reveal new relationships worth studying.

Inference vs. Prediction

- Statistical Models (Inference-focused): How does X affect Y?
  - Useful for explanation, understanding causality, and testing scientific hypotheses.
- Machine Learning Models (Prediction-focused): What is Y given X?
  - More concerned with a prediction working well than interpreting why a relationship exists.
- Trade-off
  - Machine learning often sacrifices interpretability to achieve higher predictive performance.
  - For example: a simple linear regression is easy to interpret but may underperform compared to a complex ensemble model like a random forest.

When to Use Machine Learning

Large or complex data

- ML excels when datasets are too big, high-dimensional, or messy for traditional models to handle easily.

Non-linear or hidden relationships

- Many real-world problems involve complex patterns that aren't well captured by simple linear models.
- ML methods (e.g., trees, neural networks) can detect subtle, non-linear interactions between variables.

Prediction is the priority

- Use ML when the goal is to predict future outcomes accurately rather than explain underlying mechanisms.

Categories of Machine Learning

Supervised Learning (predict outcomes for new, unseen data)

- Examples:
  - Predicting house prices from features (size, location).
  - Classifying emails as spam or not spam.

Unsupervised Learning (uncover hidden patterns)

- Examples:
  - Grouping customers by purchasing behavior.
  - Reducing dimensions of images with PCA.

Reinforcement Learning (discover strategies that maximise long-term reward)

- Examples:
  - Training robots to walk.
  - Game-playing agents (e.g., AlphaGo).

Supervised Learning

- Model is trained on labelled data: both inputs (*X*) and correct outputs (*Y*) are provided. The algorithm learns the mapping from $X \rightarrow Y$ and then applies it to new, unseen data.

- The goal is to predict the outcome (*Y*) from the input (*X*) as accurately as possible, and evaluate performance using metrics such as accuracy, precision, recall, or mean squared error, depending on the task.

Regression and Classification

Classification

- The outcome is categorical (discrete labels) and the goal is to assign each observation to one of a set of classes.
- Examples:
  - Predicting whether an email is spam or not spam.
  - Species classification (cat, dog, bird).
  - Medical diagnosis (disease present / not present).

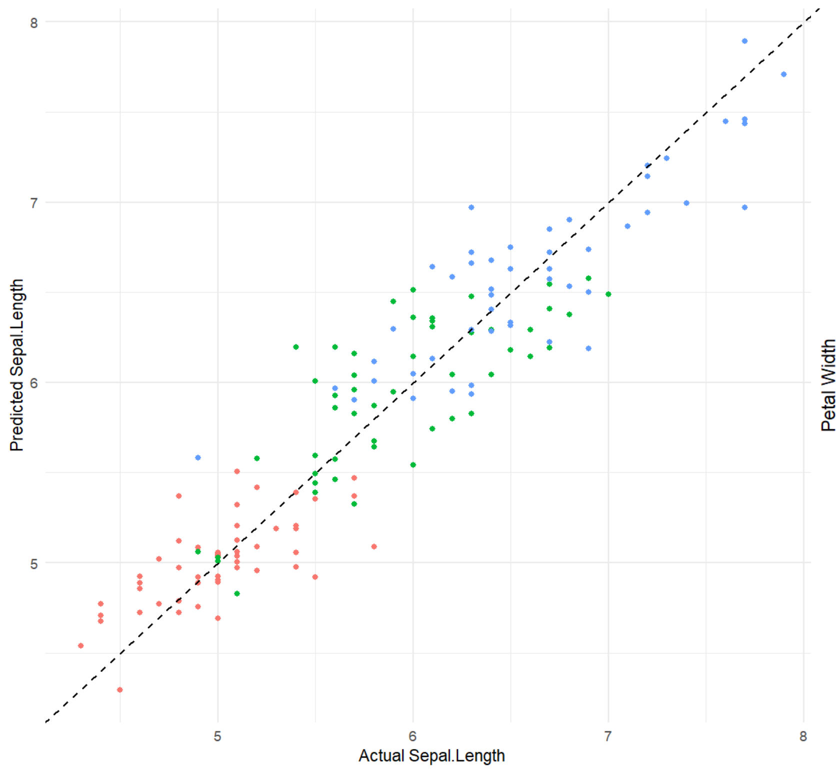
Regression

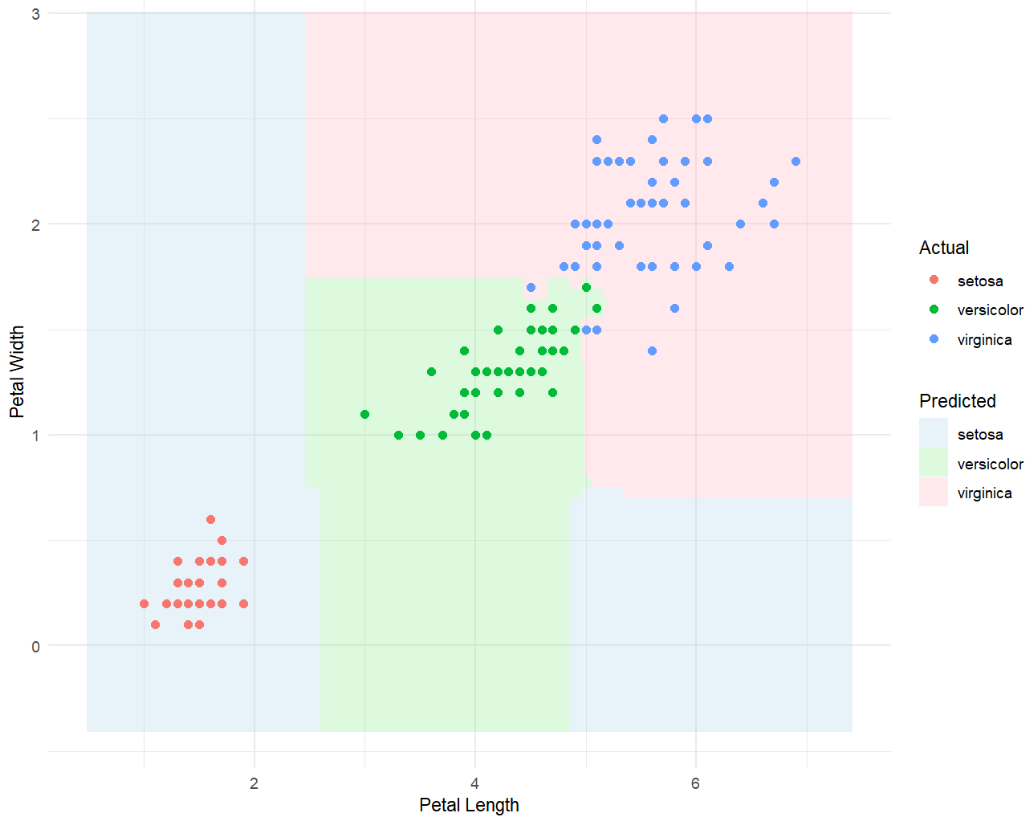
- The outcome is continuous, and the goal is to predict a numerical value given input features.
- Examples:
  - Predicting house prices.
  - Forecasting tomorrow's temperature.

The Iris Dataset

- This is one of the most famous datasets in data science and statistics.
- Originally introduced by Ronald A. Fisher in 1936 as an example of linear discriminant analysis.
- It contains measurements of iris flowers from three species: setosa, versicolor and virginica
- 4 numerical variables for each flower: sepal length, sepal width, petal length, petal width
- 1 categorical variable: species

Regression and Classification



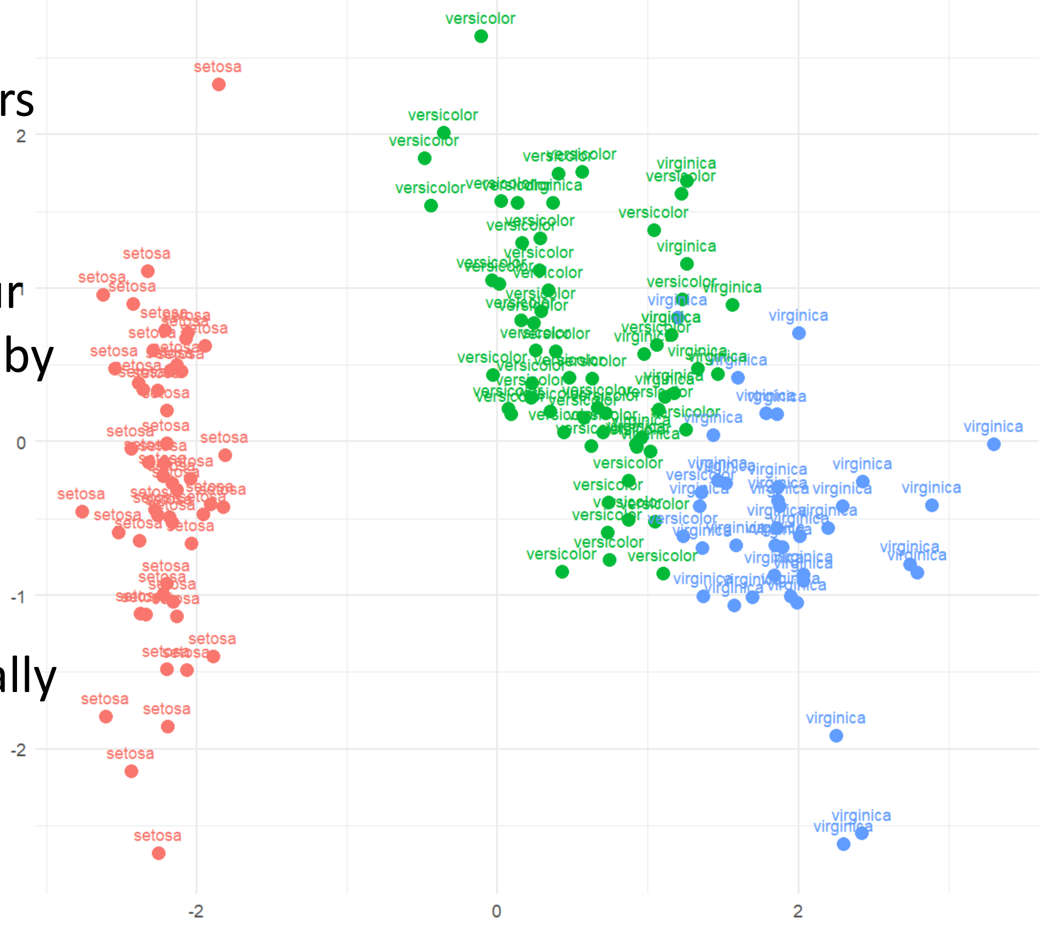


Unsupervised Learning

- Data has no target variable (no labelled outcomes), and the goal is to uncover hidden structure, patterns, or groupings in the data.
- Clustering: grouping similar observations together.
  - Example: customer segmentation in marketing, grouping species by ecological traits
- Dimensionality Reduction: simplifying data by reducing variables while preserving structure.
  - Example: Principal Component Analysis (PCA) for visualising high-dimensional data.
- Unsupervised learning is more about exploration and discovery than prediction. It's useful when patterns are unknown and labels are unavailable or expensive to obtain.

Clustering

- Groups observations into clusters based on similarity.
- Ecology: Group GPS tracks of animals by movement behaviour
- Marketing: Segment customers by purchasing habits.
- Clustering reveals behavioural states or hidden subgroups that aren't obvious at first glance.
- Example: The iris dataset naturally clusters into different flower species, even without labels.



Dimensionality Reduction

- Simplifies complex datasets by reducing the number of variables while retaining most of the important information.
- Example: reduce 50 habitat features into 2-3 key components using Principal Component Analysis (PCA).
- Visualisation: makes high-dimensional data easier to plot and interpret (e.g., 2D scatterplots of complex datasets).
- Efficiency: reduces computational cost and noise.
- Model simplification: helps avoid overfitting by focusing on the strongest signals in the data.

Reinforcement Learning

- An agent learns by interacting with an environment.
- The agent takes actions and receives feedback in the form of rewards or penalties, then adjusts behaviour to maximise long-term reward.
- Learning is based on trial and error rather than labeled data.
- Applications include:
  - Robotics: teaching robots to walk, grasp objects, or adapt to changing conditions.
  - Self-driving cars: vehicles that learn to navigate traffic, avoid obstacles, and make split-second decisions.

Common Machine Learning Algorithms

- Supervised:
  - Decision trees
  - Random forests
  - Gradient boosting
  - K Nearest Neighbours
  - Neural networks

- Unsupervised:
  - K-means clustering
  - Hierarchical clustering
  - Principal Components Analysis

Trade-Offs in ML

- Bias vs. Variance
  - Bias: error from overly simple models (underfitting).
  - Variance: error from overly complex models (overfitting).
  - Goal: find the goldilocks spot that generalises well to new data.

- Interpretability vs. Accuracy
  - Simple models (e.g., linear regression, decision trees) are easy to interpret but may have lower accuracy.
  - Complex models (e.g., ensembles, neural networks) often perform better but act as "black boxes."
  - Choice depends on whether explanation or prediction matters more.

Trade-Offs in ML

- Speed vs. Performance
  - Lightweight models train and predict quickly, useful for real-time applications.
  - High-performance models may require more computation, memory, and time.
  - Consider practical constraints: dataset size, computing power, application needs.

- There's no single "best" model. Balancing these tradeoffs depends on the problem, the data, and the goals.

Limitations and Risks

- Bias in Training Data
  - Models can only learn from the data they are given.
  - If the data is biased, incomplete, or unrepresentative, the model will reproduce and even amplify those biases.
  - Example: facial recognition systems performing poorly on underrepresented groups.
- Overfitting
  - A model that is too complex may fit noise in the training data rather than the true underlying patterns.
  - This leads to poor performance on new, unseen data.
  - Regularisation, cross-validation, and simpler models can help prevent overfitting.

Limitations and Risks

- Black-box Models
  - Many high-performing models (e.g., deep neural networks, ensembles) are difficult to interpret.
  - Lack of transparency can be a problem in high-stakes areas such as healthcare, finance, or law.

- Models should be assessed critically - not just on predictive performance, but also on fairness, interpretability, and robustness.

Machine Learning in R

tidymodels

- A modern, consistent framework for machine learning in R.
- Built on the tidyverse philosophy: consistent syntax, tidy data principles, and modular packages (e.g., parsnip, recipes, yardstick).
- Designed for reproducibility and ease of use.

caret

- The legacy standard for machine learning workflows in R.
- Provides a unified interface for many models.
- Still widely used, but gradually being replaced by tidymodels.

Machine Learning Needs Structure

- Machine learning models are sensitive to how we handle data.
- Small mistakes can make results look strong but fail in real-world use.
- Common mistakes include:
  - Data leakage: when information from outside the training set “leaks” into the model, leading to overly optimistic performance.
  - Evaluation shortcuts: testing on training data rather than unseen data.
- A good workflow:
  - Enforces a strict separation between training, validation, and test data.
  - Encapsulates preprocessing steps so they’re applied consistently across datasets.
  - Encourages transparency: every decision (feature selection, parameter choice, evaluation metric) should be reproducible and documented.

The Machine Learning Workflow

- A machine learning workflow is a formal sequence of steps guiding the process of building, validating, and applying predictive models.
- It covers the full cycle: from defining the problem to deploying the model and monitoring its performance in real-world use.
- It provides structure to ensure models are reliable, transparent, and reproducible, protects against common issues such as bias, overfitting, and data leakage, and encourages good scientific practice: every step is documented, repeatable, and justifiable.

1. Define the Problem Clearly

- Every machine learning workflow begins with a clearly defined problem.
- We need to specify:
  - Target variable (Y): what we want to predict.
  - Predictors (X): the variables we'll use to make predictions.
- Key Questions to Ask
  - Is this a classification problem (categorical outcome) or a regression problem (continuous outcome)?
  - Are we prioritising predictive accuracy or interpretability?
  - What does "success" look like - high accuracy, balanced error rates, or actionable insights?

2. Gather the Data

- Data can come from databases, surveys, sensors, field observations, or simulations.
  - Is the dataset representative of the real-world problem we're trying to solve?
  - Is it timely and reliable, or outdated and noisy?
  - Does it contain hidden biases that could mislead the model or reinforce inequality?
  - Are there legal or ethical concerns about collecting, storing, or using this data?
- Machine learning is only as good as the data it learns from.
- Poor quality or biased data leads to misleading models, unreliable predictions, and potential harm.
- "Rubbish in, rubbish out" is especially true in ML.

2.5. Exploratory Data Analysis

- Exploratory Data Analysis (EDA) helps us understand the data before modelling. It's essential for identifying patterns, outliers, skew, and relationships, allowing us to make informed preprocessing and modelling choices.
- EDA is the process of summarising, visualising, and understanding data before fitting machine learning models.
- It provides insights into variable behaviour, missingness, and structure, guiding preprocessing and feature engineering.

Why EDA is Important in Machine Learning

- Exploratory Data Analysis (EDA) is a critical step before modelling because ML algorithms are highly sensitive to data issues.
  - Feature scale – some models assume standardised predictors.
  - Data quality – missing values, errors, or inconsistent formats distort results.
  - Multicollinearity – highly correlated predictors can destabilise models.
  - Outliers – extreme values can dominate training and mislead algorithms.
- EDA reveals these problems early, helping you choose appropriate preprocessing, transformations, and models.

EDA vs. Inferential Statistics

- Exploratory Data Analysis (EDA) focuses on discovery.
  - Detects structure, patterns, and anomalies in data.
  - Prepares data for modelling and highlights preprocessing needs.
  - Relies heavily on visualisation and descriptive summaries.
- Inferential Statistics focuses on confirmation.
  - Tests hypotheses and draws conclusions about populations.
  - Uses tools like p-values, confidence intervals, and formal models.
  - Aims to generalise beyond the observed data.
- EDA is open-ended and creative, while inferential statistics is confirmatory and rule-based.

Goals of EDA

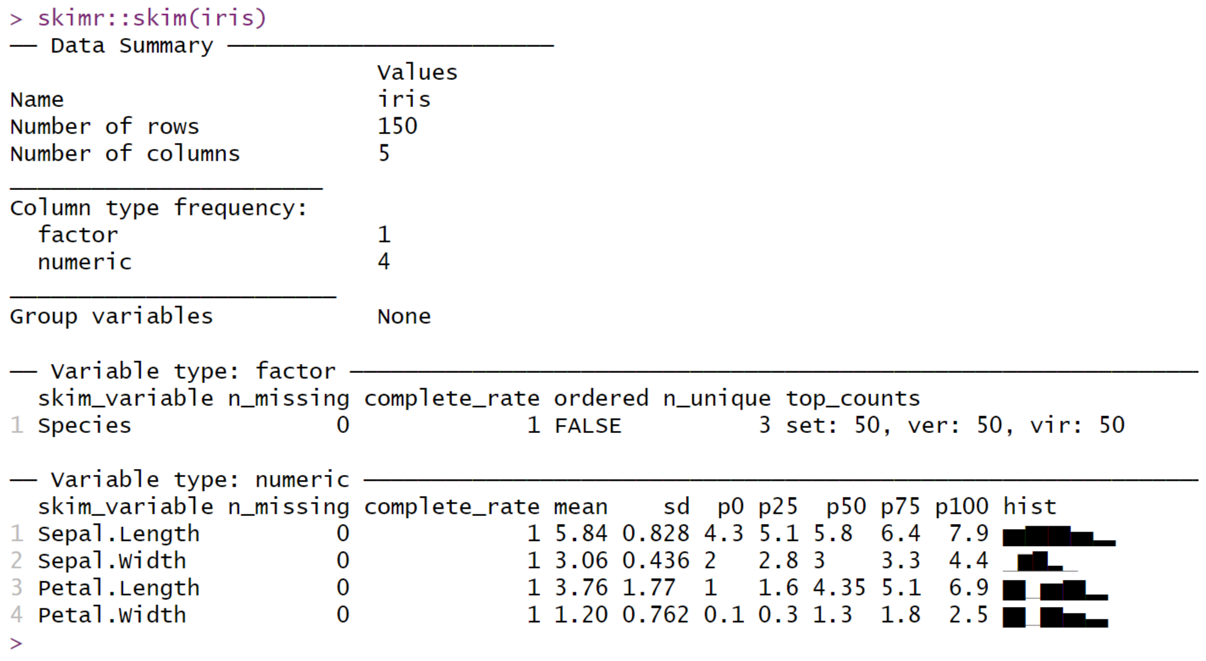
- EDA helps you build intuition about your dataset before modelling. Key goals include:
  - Understand variable types: Identify categorical, numerical, ordinal, or text data.
  - Examine distributions: Check ranges, skewness, and central tendency.
  - Detect missing values: Spot gaps that may bias models.
  - Identify outliers: Unusual points that may distort results.
  - Explore feature-target relationships: Assess correlations and predictive potential.
- Good EDA guides preprocessing choices and model selection.

Understanding Variable Types

- Different data types require different handling for both visualisation and preprocessing:
  - Continuous (numeric): Use histograms, density plots, scatterplots; may need scaling or transformations.
  - Categorical (nominal): Use bar plots or proportions; convert with one-hot/dummy encoding.
  - Ordinal (ordered categories): Use ordered bar plots, boxplots by group, or median comparisons; preserve ordering in encoding.
- Correctly identifying variable types ensures appropriate preprocessing and avoids introducing bias.

Basic Dataset Overview

- A quick structural scan helps you spot problems before modelling.



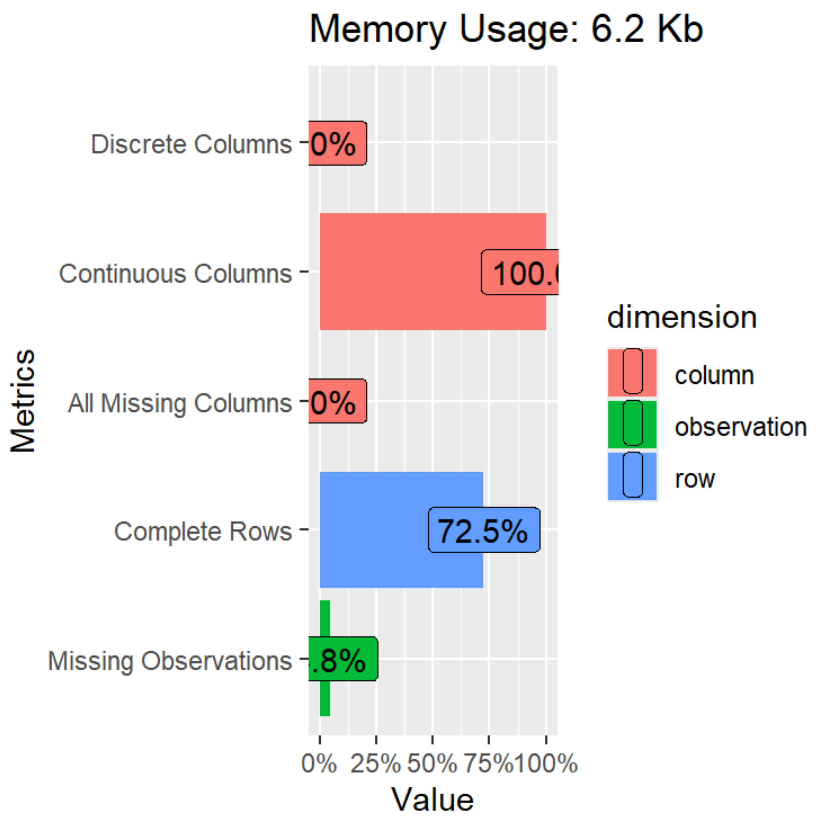
```
> skimr::skim(iris)
── Data Summary ────────────────────────
                           Values
Name                       iris
Number of rows             150
Number of columns          5
_______________________
Column type frequency:
  factor                   1
  numeric                  4
________________________
Group variables            None

── Variable type: factor ──────────────────────────────────────────────────────────────
  skim_variable n_missing complete_rate ordered n_unique top_counts
1 Species               0             1 FALSE          3 set: 50, ver: 50, vir: 50

── Variable type: numeric ─────────────────────────────────────────────────────────────
  skim_variable n_missing complete_rate mean    sd  p0 p25  p50 p75 p100 hist
1 Sepal.Length          0             1 5.84 0.828 4.3 5.1 5.8  6.4  7.9 ▆▇▇▅▂
2 Sepal.Width           0             1 3.06 0.436 2   2.8 3    3.3  4.4 ▁▆▇▂▁
3 Petal.Length          0             1 3.76 1.77  1   1.6 4.35 5.1  6.9 ▇▁▆▇▂
4 Petal.Width           0             1 1.20 0.762 0.1 0.3 1.3  1.8  2.5 ▇▁▇▅▃
>
```

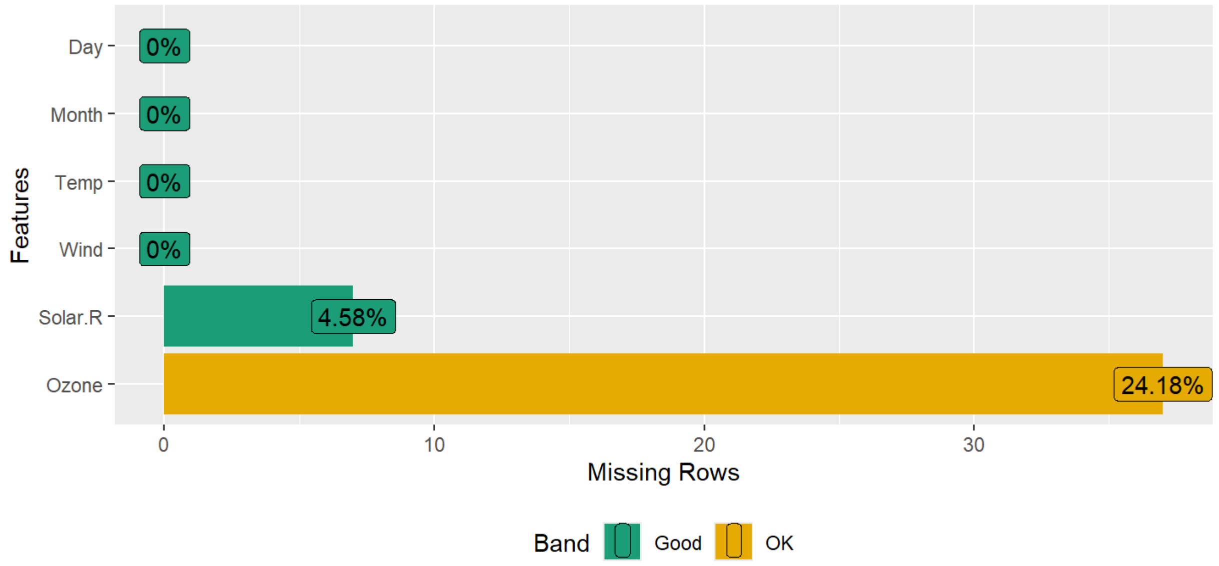
Summary Visualisation

- `DataExplorer::plot_intro (my_data)`
  - Overview of rows, columns, data types, and missingness.
  - Useful for confirming structure after import/joins.



Summary Visualisation

- `DataExplorer::plot_missing(my_data)`
  - Highlights variables with heavy missingness.



Visualisation: Variable Roles and Counts

- Class frequency (target / categorical features)
  - Use bar plots to check balance vs. imbalance.
  - Imbalanced classes affect metric choice and resampling later.

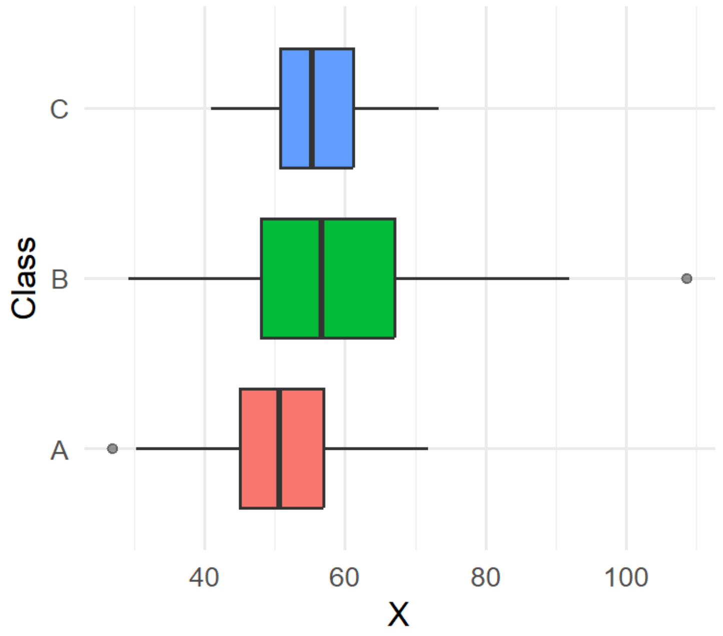
- Numeric feature distribution
  - Use histograms (and/or density plots) to see range, skew, and multimodality.
  - Guides transformations (e.g., log), binning, or outlier handling.

Checking Distributions

- Reveal shape of data
  - Normal, skewed, or multimodal distributions guide choice of model and transformations.
- Identify outliers
  - Extreme values may signal data entry errors, rare events, or influential points.
- Guide preprocessing
  - For skewed predictors, consider log/square-root transforms.
  - For non-normal residuals, consider robust models.
- Model assumptions
  - Some algorithms (e.g., linear regression) assume normality and homoscedasticity, while others (trees) do not.

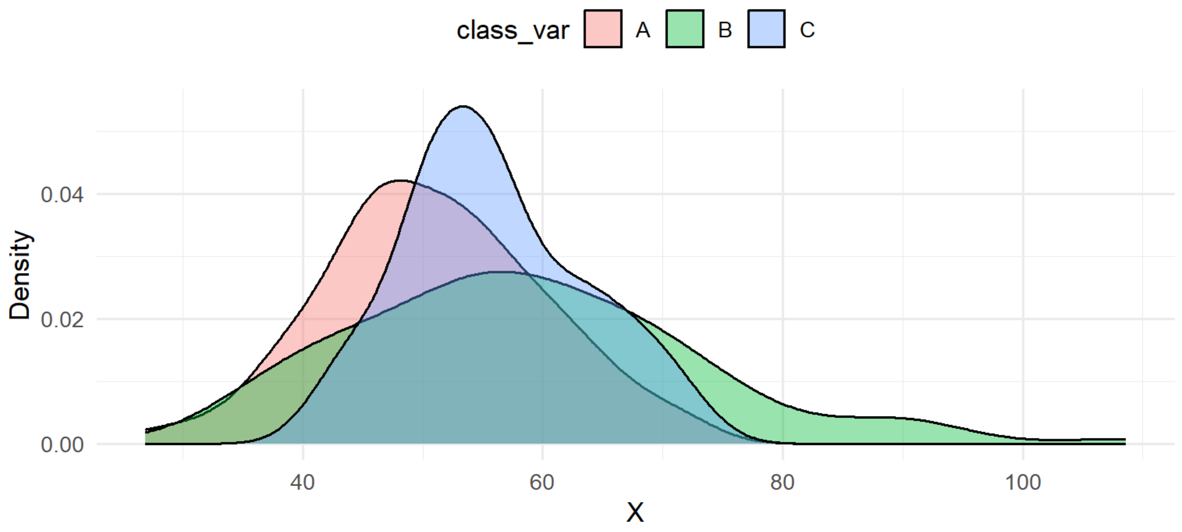
Boxplots

- Summarise distributions with median, interquartile range (IQR), and outliers (points beyond 1.5×IQR).
- Excellent for comparing a numeric variable across categories.
- Quickly reveals skew, spread, and group differences.



Density Plots

- Smooth estimate of a variable's distribution shape.
- Useful for seeing skewness, multimodality, and overlaps between groups.

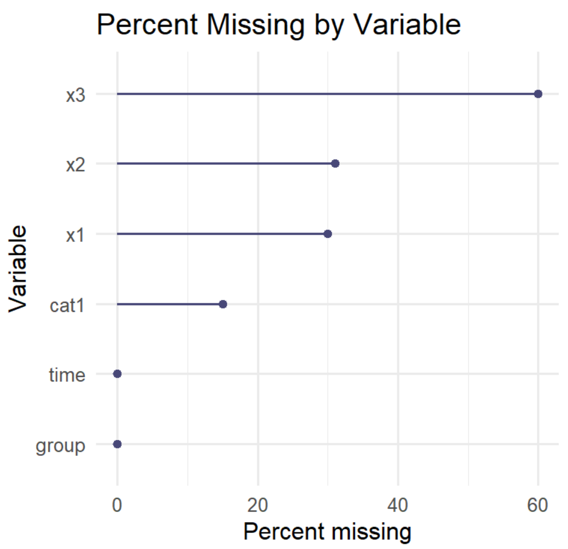


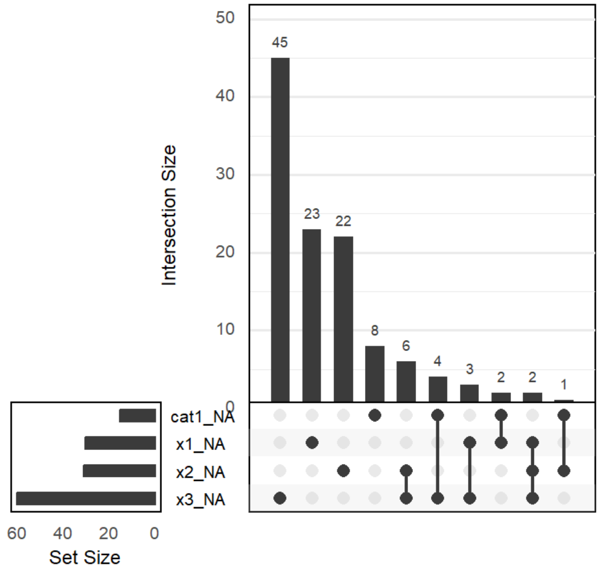
Detecting Missing Data

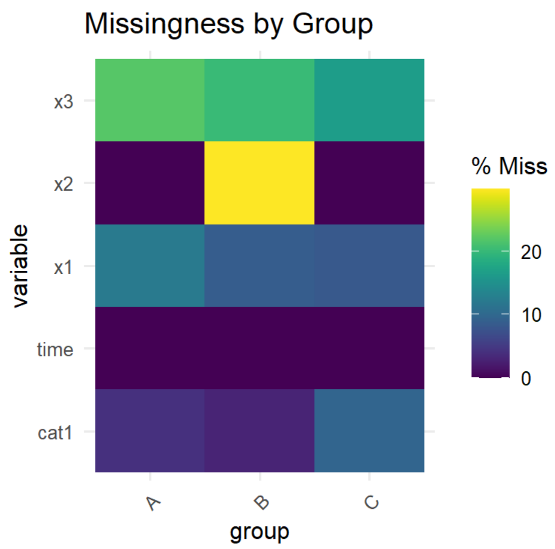
- Missingness can reduce sample size, bias results, and break models; visuals show where, how much, and which variables co-miss.
- We are looking for:
  - Variables or rows with the highest % missing
  - Blocks/patterns of missingness (by group, time, source)
  - Co-missingness (variables that tend to be missing together)

- Useful tools (R):
- Heatmap of missingness: naniar::vis_miss()
- Missing % per variable: naniar::gg_miss_var()
- Combinations of missing vars: naniar::gg_miss_upset()
- Missingness by group: naniar::gg_miss_fct()

Detecting Missing Data







Detecting Outliers

- An outlier is an observation that deviates markedly from the overall pattern of the data. It may be a data error, a rare/novel case, or evidence of a hidden subgroup.
- We detect outliers because they:
  - Can distort summaries (mean, standard deviation) and mislead models.
  - May exert high influence on fitted parameters and predictions.
  - Often reveal data-quality issues or important phenomena
- After detection we:
  - Verify and correct obvious errors; document changes.
  - Consider robust methods (robust regression, median-based summaries).
  - Transform (e.g., log), or model with heavy-tailed distributions when appropriate.
  - Remove only with clear justification and record the decision.

Visualising Relationships Between Features

- Relationships between numeric variables:
  - Scatterplots reveal linear/non-linear trends, clusters, and heteroscedasticity. Add a smooth to see trend.
- Relationships between categorical and numeric variables:
  - Boxplots/violins compare distributions across groups and highlight spread and outliers.
- Redundancy among predictors
  - Correlation heatmaps (or pair plots) flag highly correlated features that may cause multicollinearity or add little new information.

Correlation and Redundancy

- Highly correlated predictors carry overlapping information. This redundancy can inflate variance, destabilise linear models, and make effects hard to interpret.
- What we look for:
  - Blocks of strong correlation ($|r| \geq 0.8$ is a common flag).
  - Near-duplicate variables (engineering artifacts, multiple encodings).
  - Perfect/near-perfect pairs that can break model fitting.
- What to do:
  - Remove or consolidate one of the pair (keep the more interpretable/stable).
  - Combine with feature extraction (PCA) when many features are correlated.
  - Use regularisation (Ridge/Lasso/Elastic Net) to mitigate collinearity in linear models.

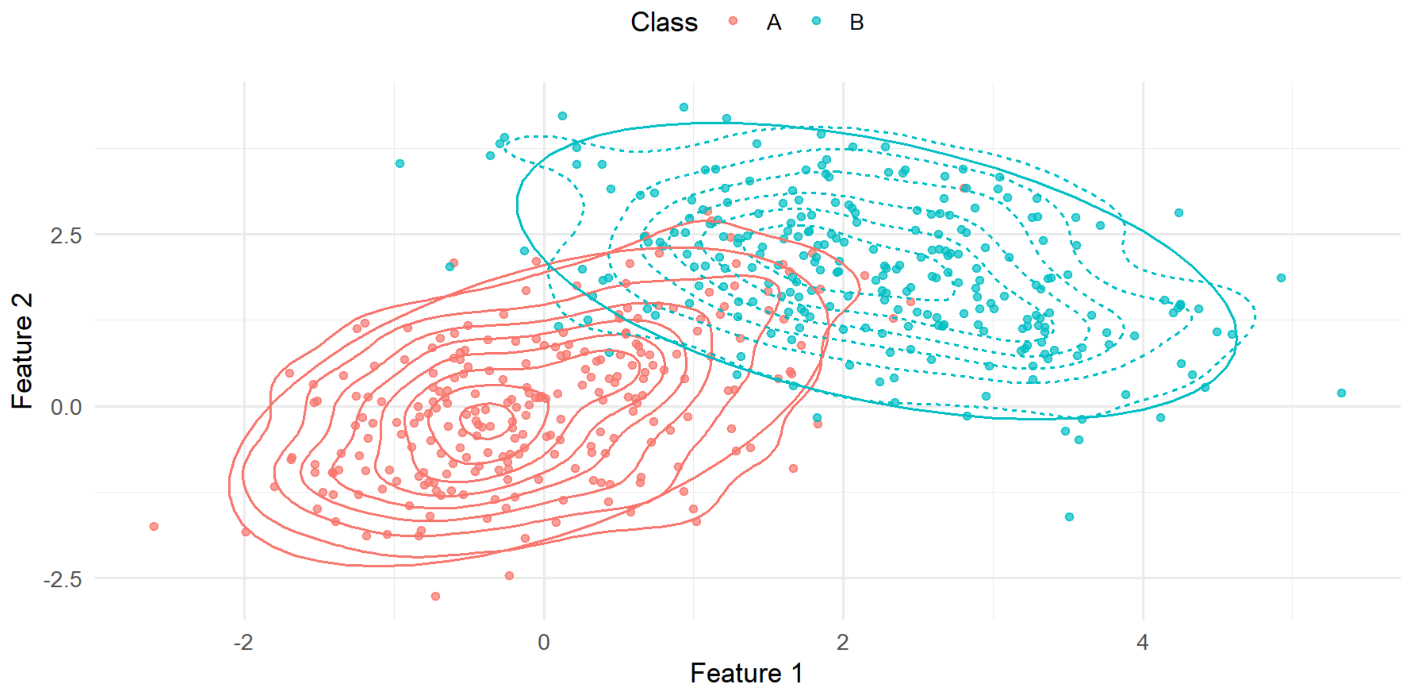
Class Balance

- Imbalanced classes can bias models toward the majority class.
- Accuracy becomes misleading (a model that predicts the majority class can look “good”).
- It affects how you split data, choose metrics, and tune models.
- Inspect class counts and proportions (table + bar plot).
- If imbalance is severe, plan for: stratified splits, appropriate metrics (ROC-AUC, PR-AUC, F1, balanced accuracy), and possibly rebalancing (over-sampling/under-sampling).

Class Separation

- Class separation plots show how different groups or classes can be distinguished in the feature space.
- When classes are clearly separated, simple models such as logistic regression may perform well.
- If there is substantial overlap between classes, model accuracy is likely to be lower.
- In such cases, more flexible and non-linear models may be required to capture complex boundaries.
- Visualising class separation also helps identify whether preprocessing or feature engineering could improve separation.

Class Separation



Using EDA in Your Analysis

- Exploratory data analysis should be performed on the training set only, to avoid leaking information from the test set into the modelling process.
- The insights you gain from EDA should be used to guide recipe construction, such as deciding which transformations, imputations, or feature engineering steps are necessary.
- After fitting models, it is important to revisit EDA findings and compare them with results such as feature importance or residual analysis to refine your approach.

3. Preprocess the Data

- Raw data almost never comes in a form ready for machine learning.
- Models are sensitive to scaling, missingness, and hidden signals.
- Careless preprocessing can lead to biased, unstable, or even invalid results.
  - Handle missing values - decide whether to drop, impute, or flag them.
  - Outliers - investigate whether they are errors, rare but real cases, or influential points.
  - Scaling - standardise or normalise (range 0-1) for models sensitive to scale.
  - Remove leakage variables - exclude predictors that indirectly give away the target (e.g., using “hospital discharge date” to predict survival).

Data Preprocessing in R

- Preprocessing tasks include:
  - Cleaning and handling missing values
  - Transforming features (scaling, normalisation, log-transforms)
  - Encoding categorical variables (dummy coding, one-hot)
  - Reshaping data into model-ready formats
- It matters because:
  - Machine learning models are highly sensitive to scale, structure, and completeness.
  - Poor preprocessing introduces bias, inflates performance, or prevents deployment.
  - Strong preprocessing builds a solid foundation for trustworthy models.
- In R, the recipes package (part of tidymodels) makes preprocessing structured and reproducible.

Goals of Preprocessing

- Preprocessing aims to:
  - Make raw data usable
  - Ensure algorithm compatibility
  - Normalise scales
  - Encode categories
  - Impute missing entries
  - Remove outliers
- The key is to transform training data in a way that can be replicated exactly on new data.

Recipes in tidymodels

- A recipe is a modular pipeline for data preprocessing.
  It defines transformations step by step, using step_* functions.
- Steps - individual preprocessing actions (e.g., normalisation, imputation, encoding).
- Prep - “train” the recipe on the training data (learn scaling factors, levels of factors, etc.).
- Bake - “apply” the trained recipe to new data (test, validation, deployment).
- This guarantees that no test data leaks into preprocessing.

```
recipe(Species ~ ., data = iris) %>%
step_normalize(all_numeric_predictors()) %>%
step_dummy(all_nominal_predictors())
```

Handling Missing Data

- Machine learning models cannot handle NA values directly.
  We must deal with missingness carefully to avoid bias or data loss.
- Common strategies include:
  - Remove rows with NA – simple, but risky if too much data is lost.
  - Impute with summary statistics – mean, median, or mode (step_impute_mean, step_impute_median, etc.).
  - Model-based imputation - more sophisticated methods such as KNN or bagged trees.
- The best approach depends on:
  - How much data is missing
  - Whether missingness is random or systematic
  - The importance of the variable in the model
  - Bad imputation leads to bad predictions.

Handling Missing Data

- The reason for the missing data matters.

MCAR (Missing Completely at Random):

- Probability of missingness is unrelated to observed or unobserved data (e.g., a sensor randomly fails). Analysis on complete cases is unbiased, but less efficient.

MAR (Missing at Random):

- Missingness depends on observed data, not the missing value itself. (e.g., income missing more often for younger people). Imputation methods are valid if predictors of missingness are included. Cannot be tested for directly, but check patterns against observed variables.

MNAR (Missing Not at Random):

- Missingness depends on the unobserved value itself (e.g., people with very high income less likely to report it). Requires explicit modelling. Cannot be tested for, must be argued from context.

Imputing Missing Values

- Imputation replaces missing entries with plausible values so we can retain as much data as possible.
  - step_impute_mean() - mean for numeric predictors
  - step_impute_median() - median for numeric predictors
  - step_impute_mode() - most common category
  - step_impute_knn() - K-nearest neighbours
  - step_impute_bag() - bagged trees
  - step_impute_linear() - regression-based

```
recipe(y ~ ., data = dat) %>%
step_impute_median(all_numeric_predictors()) %>%
step_impute_mode(all_nominal_predictors())
```

Scaling and Centering Predictors

- Many machine learning models assume predictors are on the same scale.
- Centering: subtracts the mean so variables have mean = 0.
- Scaling: divides by the standard deviation so variables have SD = 1.
- Scaling and centering:
  - Is essential for distance-based models (KNN, SVM, neural nets).
  - Prevents large-scale variables from dominating.
  - Not necessary for tree-based models (random forest, boosting), but often still applied for consistency.

```
recipe(y ~ ., data = dat) %>%
step_normalize(all_numeric_predictors())
```

Encoding Categorical Predictors

- Most machine learning algorithms require numeric inputs, so categorical predictors must be encoded.
- One-Hot Encoding: creates binary indicator variables for each category. step_dummy()
- Group Rare Levels: combines infrequent categories into "other" to avoid instability: step_other()
- Handle Missing Categories: explicitly codes missing levels as "unknown." step_unknown()
- This ensures categorical variables are represented consistently and prevents models from breaking when new or rare levels appear.

```
recipe(y ~ ., data = dat) %>%
step_dummy(all_nominal_predictors()) %>%
step_other(all_nominal_predictors()) %>%
step_unknown(all_nominal_predictors())
```

Feature Engineering within Recipes

- Feature engineering creates new predictors from existing ones to better capture patterns. It can reveal non-linear relationships or interactions that raw variables miss.
- Interactions: combine predictors (e.g., Age × Income) using step_interact()
- Polynomials: capture curves with squared or cubic terms using step_poly()
- Careful use of feature engineering can improve model accuracy, but it also increases model complexity and the risk of overfitting.

```
recipe(y ~ ., data = dat) %>%
step_interact(~ var1:var2) %>%
step_poly(var3, degree = 2) %>%
step_ns(var4, deg_free = 4)
```

Preprocessing

- Preprocessing is not optional - it is part of the model that shapes every downstream decision.
- The recipes package provides structure, reproducibility, and clarity.
- Do it early, do it well, and your models will thank you.

Checking and Debugging Recipes

- You can use `summary(recipe)` to check which variables are included, their assigned roles, and any preprocessing steps applied to them.
- The `tidy()` function allows you to inspect the learned parameters of each step (e.g., means and standard deviations used in normalisation, thresholds used in imputation).
- To examine how preprocessing is applied, you can use `juice()` to preview the transformed training data, and `bake()` to apply the recipe to the test data or new data.

4. Split the Data

- Models must be evaluated on data they haven't seen during training. Without a test set, we risk overfitting - a model that looks great on past data but fails on new cases.
- Training set is used to fit the model and tune parameters.
- Test set is held back until the very end for unbiased evaluation.

Advanced approaches

- Validation set is a third set is carved out for hyperparameter tuning.
- The test set should never influence model training.
- If you "peek" at the test set during development, it's no longer a valid test.

A GOOD EXAMPLE OF
OVERFITTING

Splitting Data Comes First

- You should always split your dataset before any modelling or preprocessing to ensure an honest evaluation of model performance.
- The training set is used for model fitting and hyperparameter tuning, where the algorithm learns patterns from the data.
- The test set is reserved for the final unbiased evaluation of the model's performance on unseen data.
- The test set must never be used during training or tuning, as this can lead to data leakage and artificially inflated performance estimates.

5. Choose the Model

Match the model to the problem

- Linear vs. Non-linear: linear regression/logistic vs. trees, random forests, boosting.
- Outcome: regression for continuous data, classification for categories.

Balance trade-offs

- Interpretability: linear/logistic regression, decision trees.
- Accuracy / flexibility: random forests, gradient boosting, neural networks.
- Computation: some models scale poorly with very large datasets.
- Model choice isn't just about "what's most accurate" - it's about what best fits your problem, data, and constraints.

Specifying Models with parsnip

- The parsnip package provides a unified and consistent syntax for specifying models, regardless of the underlying engine.
- You define the model structure (e.g., random forest, logistic regression, SVM) separately from the engine that fits it (e.g., ranger, glmnet, xgboost).
- This separation makes it easy to switch between engines without rewriting the model specification.
- You can also leave arguments as tune() when you want them optimised during the tuning process.

```
rf_spec <- rand_forest(mtry = tune(), trees = 500) %>%
set_mode("classification") %>%
set_engine("ranger")
```

Building a Workflow

- A workflow combines a model and a recipe into a single object.
- This ensures that preprocessing and modelling steps are applied consistently during training and testing.
- Workflows help to prevent data leakage, since all preprocessing is locked inside the workflow and only applied to the appropriate data.
- They also make experiments more reproducible and easier to manage.

```
wf <- workflow() %>%
add_model(rf_spec) %>%
add_recipe(my_recipe)
```

Fitting a Workflow

- Once a workflow is defined, it can be fitted directly to training data.
- This applies all recipe preprocessing steps and then trains the model in a single operation.
- The result is a fitted workflow object that contains both the trained model and the preprocessed training data.
- This approach ensures consistency and prevents mistakes from manually applying steps in the wrong order.

```
final_model <- fit(wf, data = train)
```

6. Tune Model Hyperparameters

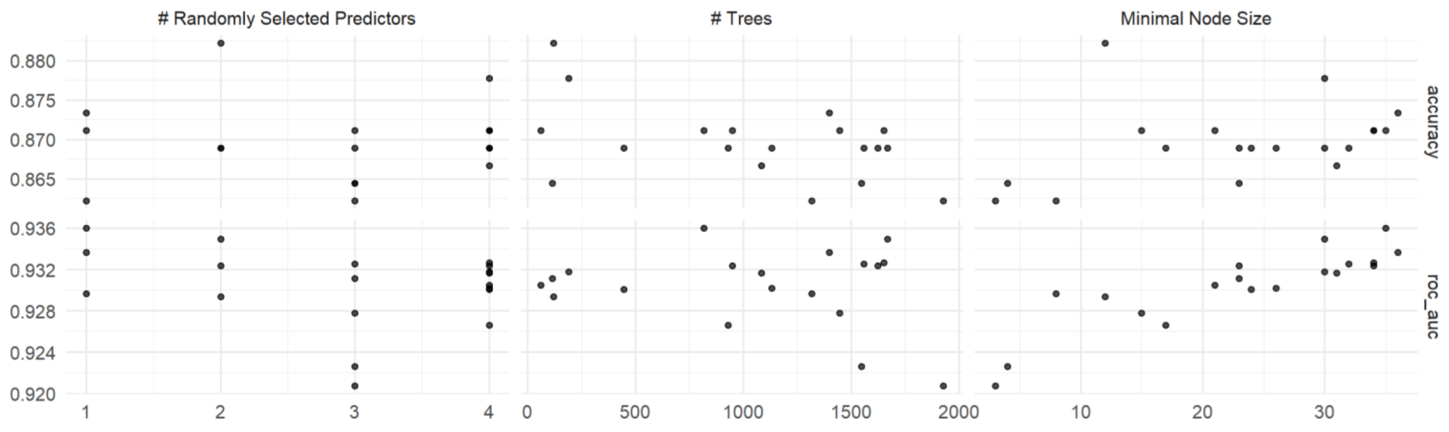
- Hyperparameters are settings chosen before training begins.
- They control model complexity, learning speed, or structure.
- Unlike model parameters (e.g., regression coefficients), they aren't learned from data.
- Tuning hyperparameters:
  - Prevents underfitting (too simple) or overfitting (too complex).
  - Ensures the model generalises to unseen data.
  - Protects the test set as a true final check.

Hyperparameter Tuning

- Hyperparameter tuning searches for settings that improve out-of-sample performance without touching the test set.
- tune_grid() evaluates many candidate settings by repeatedly fitting the workflow on training folds and validating on held-out folds from resamples.
- The grid argument controls how many combinations to try (larger grids explore more but take longer).
- Results are summarised with chosen metrics (e.g., accuracy, ROC-AUC, RMSE), averaged across folds to estimate generalisation.

Autoplot for performance

- After tuning, autoplot(tuned) provides quick visual diagnostics of how hyperparameters affect your chosen metric(s).
- Use it to spot the stability of the best settings (are top scores narrow or broad?) and to avoid overfitting to a tiny region of the grid.
- These visuals help justify your final hyperparameter choices and guide whether a wider search or different ranges are needed.



Selecting Best Model

- After tuning, you should select the hyperparameter set that optimises your chosen metric (e.g., ROC-AUC for imbalanced classification, RMSE for regression).
- Use that selection to finalise the workflow, so the winning settings are locked in for training on the full training set.
- Fit the finalised workflow to produce the final trained model that you'll evaluate on the test set (or via last_fit).

```
best <- select_best(tuned, "roc_auc");
final_model <- fit(finalize_workflow(wf, best), train)
```

7. Evaluate the Model

- The test set represents future, unseen data. Performance here is the true test of generalisation.

- For Classification:
  - Accuracy: proportion correctly predicted.
  - Recall: ability to detect positives.
  - Specificity: ability to detect negatives.

- For Regression:
  - RMSE (Root Mean Squared Error): penalises large errors.
  - MAE (Mean Absolute Error): average absolute difference.
  - R^2 (Coefficient of Determination): variance explained by the model.

- Never evaluate on training or validation data. Only the test set gives a fair estimate of real-world performance. The goal is to do well on data the model has never seen before.

Making Predictions

- Once the workflow has been fitted, you can use it to generate predictions on new data.
- The workflow automatically applies the same preprocessing steps from the recipe to the test data before predicting.
- This ensures that the model sees test data in the same format and scale as the training data.
- Predictions can be class labels or probabilities depending on how the model was specified.

```
predict(final_model, new_data = test)
```

Evaluating Performance

- After making predictions, we need to quantify model performance.
- Evaluation ensures that the model generalises well beyond the training data.
- Common metrics include accuracy, precision, recall, F1-score, and ROC-AUC, depending on the problem.
- The yardstick package in tidymodels provides a consistent framework for performance evaluation.

```
metrics(preds, truth = outcome, estimate = .pred_class)
```

8. Deploy and Monitor the Model

- Deployment involves saving the model for reuse and then using it. This can involve integrating into applications:
  - Web apps or APIs for real-time predictions
  - Dashboards for decision support
- Monitoring Matters because:
  - Data and environments change over time, which can lead to model degradation. We need to track accuracy, error rates, or key business metrics over time
  - Detect when predictions deviate from expectations, and set alerts for deteriorating performance
  - A model is not “done” when trained - it becomes part of a system that must be maintained.

Reproducibility

- To trust results, we must be able to recreate them exactly. This is essential for science, policy, and production use.

1. Set a random seed (set.seed())
   - Ensures identical splits, resampling, and model fitting
2. Record package versions
   - Use sessionInfo() or renv for project-specific environments
3. Capture all steps in workflow objects
   - Recipes + models + tuning decisions stored together
4. Save trained models with parameters
   - Guarantees future predictions are identical to current ones

Saving and Reusing Models

- You should save the fitted workflow (recipe + trained model) so it can be reused consistently in apps, scripts, or reports.
- Saving preserves all learned preprocessing steps and hyperparameters, ensuring identical predictions later.
- Reload the object and call predict() on new data; the recipe will be applied automatically before the model predicts.

```
saveRDS(final_model, "model.rds");
predict(readRDS("model.rds"), new_data)
```

Mistakes in Machine Learning Workflows

- Beginners often fall into common traps that break models:
- Applying transformations after splitting: Causes data leakage (the model sees information it shouldn't).
- Evaluating on training data: Leads to overfitting and false confidence.
- Tuning with test data: Invalidates test performance-test data should only be touched once.
- Forgetting to save the final model: Makes results impossible to reproduce or deploy.
- A structured workflow (preprocessing + resampling + test set) prevents these errors.

Framework Limitations

- tidymodels and related frameworks can be slower with very large datasets, especially when tuning or resampling, compared to direct model calls.
- Some new or niche models may not yet be supported, limiting flexibility for cutting-edge applications.
- Debugging errors can be more complex because the workflow chains together many steps.
- Despite these limitations, the benefits of consistency, reproducibility, and reduced risk of data leakage usually outweigh the drawbacks.

Baseline Models

- A baseline model is a deliberately simple model that establishes a minimum performance benchmark for comparison.
- In regression tasks, a common baseline is to always predict the mean of the target variable.
- In classification tasks, a common baseline is to always predict the majority class.
- If your trained model performs worse than the baseline, this is a serious warning sign that the model may be overfitting, underfitting, or simply learning noise.
- Baselines ensure that any added model complexity is actually providing value beyond trivial rules.

Why Baseline Models Matter

- Baseline models help you determine whether a more complex model actually adds value compared to a trivial approach.
- They allow you to interpret performance metrics in context, by showing what accuracy, RMSE, or AUC looks like under a naive prediction rule.
- They can reveal issues such as data leakage or improper tuning if a complex model performs worse than the baseline.
- They are especially useful in the early stages of modelling, providing a quick reference point before investing significant time in tuning and optimisation.

Example: Regression Baseline

- A naïve regression baseline always predicts the mean of the training data, regardless of input features.
- This provides a minimum performance threshold for RMSE (Root Mean Squared Error).
- If your trained model does not improve upon this baseline, it suggests that the model is not capturing meaningful signal and may be overfitting or learning noise.

```
mean_model <- mean(train$outcome)
```

- Then compute RMSE on the test set:

```
sqrt(mean((test$outcome - mean_model)^2))
```

- Any real model should aim to improve on this RMSE.

Example: Classification Baseline

- A naïve classifier always predicts the most common class in the training set.
- This provides a minimum threshold for metrics such as accuracy, precision, and recall.
- If a more complex classifier cannot outperform this baseline, it suggests the model is not capturing useful patterns and may be overfitting or learning noise.

```
majority_class <- train %>% count(outcome) %>% top_n(1)
test_preds <- rep(majority_class, nrow(test))
```

Performance Metrics

- The choice of performance metric depends on the type of task being solved.
- For regression problems, commonly used metrics include Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R^2. Each highlights a different aspect of error or explanatory power.
- For classification problems, we often use Accuracy, Precision, Recall, F1 score, or AUC. These provide insights into correctness, sensitivity to minority classes, or overall discriminative ability.
- No single metric is “best.” The right choice depends on the goals of the analysis and the cost of different types of errors in the real-world context.

RMSE: Root Mean Squared Error

- Root Mean Squared Error is the square root of the average squared difference between predicted values and observed values.
- Because errors are squared before averaging, RMSE penalises large deviations much more than small ones.
- The formula is:

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

- RMSE is most useful when large errors carry a high cost, such as in energy load forecasting or weather prediction.
- A lower RMSE indicates better predictive performance, but it should always be interpreted in the context of the scale of the outcome variable.

MAE: Mean Absolute Error

- MAE is the mean of absolute errors between predicted and observed values.

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

- It is robust to outliers, since all errors contribute proportionally without being squared, and it provides an easily interpretable measure of the average prediction error in the same units as the outcome.
- A lower MAE indicates better predictive performance, though it does not highlight rare large mistakes as much as RMSE.
- MAE is useful in contexts where all errors matter equally, such as estimating delivery times or demand forecasts.
- Compared with RMSE, MAE provides a more intuitive sense of average deviation, while RMSE emphasises extreme errors.

Accuracy

- Accuracy is the proportion of correct predictions out of all predictions.
- Formula:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

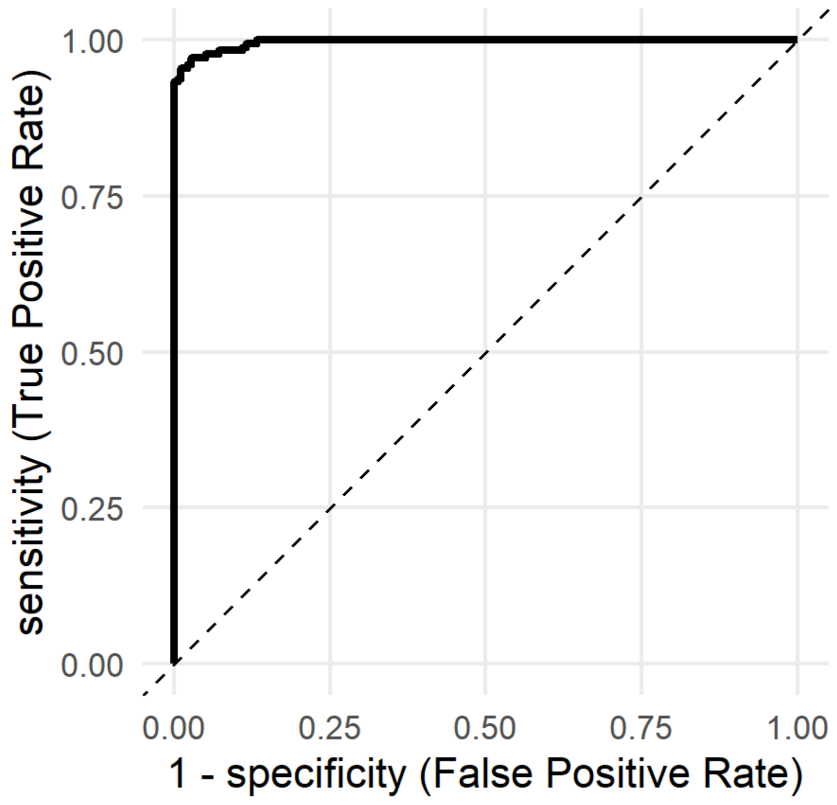
- It is simple, intuitive, and works well when classes are balanced.
- Accuracy alone can be misleading in imbalanced datasets.
- For example, if 95% of cases are negative, a model that always predicts “negative” achieves 95% accuracy but zero usefulness.
- Accuracy does not tell us about errors on each class - precision, recall, and F1 are often needed for a fuller picture.
- Best used as a baseline metric, but should be complemented with others depending on context.

Precision, Recall and F1

- Precision measures the proportion of predicted positives that are actually positive. It answers the question: “When the model predicts positive, how often is it correct?”
- Recall measures the proportion of actual positives that the model successfully identifies. It answers: “Of all the real positives, how many did we catch?”
- F1 Score is the harmonic mean of precision and recall, providing a single measure that balances the trade-off between them. It is useful when you need to balance both concerns rather than prioritising one.
- These metrics are especially important in imbalanced datasets, where accuracy alone can be misleading.

ROC Curve and AUC

- The ROC curve shows the trade-off between Recall and False Positive Rate across all classification thresholds.
- AUC (Area Under the ROC Curve) summarises discrimination ability across thresholds: higher is better.
- AUC = 0.5 indicates random guessing; AUC = 1.0 indicates perfect classification.
- ROC/AUC are threshold-independent and useful when class proportions are imbalanced or when you care about ranking by risk rather than a single cutoff.


sensitivity (True Positive Rate)
1 - specificity (False Positive Rate)