Introduction to Machine Learning

Dr Niamh Mimnagh

niamh@prstats.org

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

PR STATS

Training vs. Testing Performance

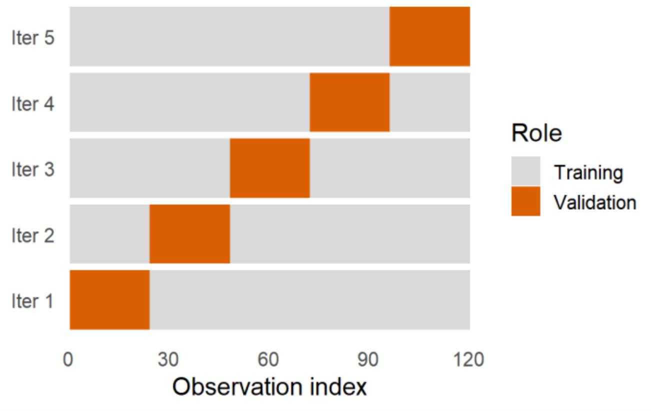
- Training error is computed on the data the model has already seen, so it often underestimates the true error and can hide overfitting.
- Test error is computed on unseen data, so it provides a better estimate of generalisation and should be reserved for a single, final evaluation.
- Always separate data into train/validation/test (or use cross-validation); use validation (or CV folds) for tuning, and touch the test set only once at the end.
- Guard against leakage by applying all preprocessing (scaling, PCA, feature selection) inside the resampling loop so it is learned from training folds only.
- Monitor the gap between train and test: a small gap with high error suggests underfitting; a large gap with low train error suggests overfitting.

Holdout Validation

- Holdout validation splits the data once into training and testing sets (e.g., 70/30), ideally with a fixed random seed and stratification to preserve class balance; for time series use a temporal split, not a random one.
- Train the model only on the training set, and fit all preprocessing steps on the train set then apply them to the test set to avoid leakage.
- Use the training set (or a further validation split) for hyperparameter tuning, and keep the test set untouched until a single, final evaluation.
- A single split yields a high-variance estimate that depends on how the split landed.
- Holdout is fast and simple and works well for large, homogeneous datasets, but it can be unreliable for small data.

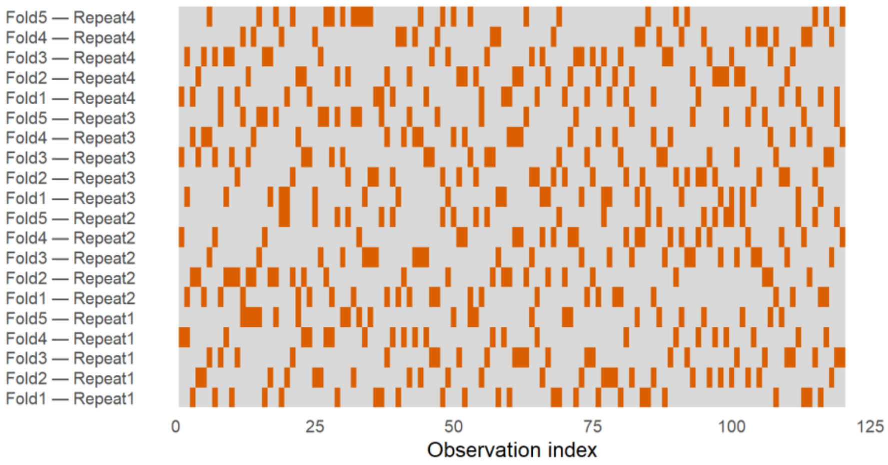
K-Fold Cross-Validation

- In k-fold cross-validation, we split the data into k roughly equal, non-overlapping folds, typically with stratification to preserve class balance.
- We iterate k times, each time training on k−1 folds and evaluating on the held-out fold, so every observation serves once as validation and k−1 times as training.
- We average the metric across folds to get a more stable estimate of generalisation than a single train/test split.
- Sensible defaults are k = 5 or 10 for tabular data; larger k reduces bias but increases variance and compute.



Repeated Cross-Validation

- Repeated cross-validation runs k-fold CV multiple times with different random fold assignments, then pools the results across all folds × repeats.
- Be aware of the higher computational cost; fix seeds and save resample indices so results are reproducible.
- Treat repeated CV as a model-selection tool, and still reserve a final test set (or a later time window) for an unbiased last check.



Linear Regression

- Linear regression is a supervised learning method for predicting a continuous outcome.
- Models the relationship between a dependent variable (*y*) and one or more predictors (*x*).

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \varepsilon$$

- β_0 = Intercept (expected value of *y* when predictors are 0)
- β_j =Coefficient (effect of predictor $x_j$on *y*)
- ε= Error term (variation not explained by the model)
- Coefficients tell us how much *y* changes when a predictor increases by one unit, holding other predictors constant.

Predictive vs. Inferential Use

- Inference – Understanding relationships between predictors and the outcome.
  - *Example:* Is smoking associated with higher blood pressure?
  - Focus on statistical significance, effect sizes, confidence intervals.

- Prediction – Accurately forecasting new values of the outcome.
  - *Example:* Given a patient's age, weight, and habits, can we predict their blood pressure?
  - Focus on generalisation to unseen data, predictive accuracy.

Why Use Linear Regression for Prediction?

Strengths

- Fast to train and computationally efficient.
- Interpretable - coefficients show direction and magnitude of effects.
- Works with both numerical predictors and categorical predictors

Limitations

- Linear and additive assumption - predictions are straight-line combinations of predictors.
- Poor with nonlinear patterns – struggles when relationships are curved or involve interactions unless manually added.
- Sensitive to outliers – extreme values can heavily influence coefficients.

Defining a Recipe

- Transform the outcome to a log scale to handle right-skew and stabilise variance.
- Impute numeric predictors with the median to handle missing values robustly.
- Impute categorical predictors with the mode (most frequent level).
- Collapse rare categorical levels (e.g., <1%) into an "Other" bucket to reduce sparsity/leakage.
- Remove zero-variance predictors that carry no information.
- Standardise numeric predictors (centre and scale) to put features on comparable scales and help regularised/gradient methods.

Defining the Model Specification

What it declares

- Algorithm: linear_reg, logistic_reg, rand_forest, boost_tree, etc.
- Mode: regression or classification (often inferred, or set via set_mode()).
- Engine: implementation backend (e.g., "lm", "glmnet", "ranger“)
- Hyperparameters: values or placeholders like tune() (e.g., penalty, mixture, trees).

What it doesn’t do

- No preprocessing, no formulas, no data storage.
- No fitting until you call fit() (usually via a workflow).

Creating a Workflow

- A single object that glues the preprocessor (recipe) to the model spec.
- Guarantees identical preprocessing is applied at fit and predict time.
- Makes pipelines reproducible and components swappable.
  - Start with your recipe and model spec.
  - Create the workflow
  - Fit on training data; predict on test; evaluate with yardstick.

```
lm_spec <- linear_reg() %>% set_engine("lm")
lm_wf <- workflow() %>% add_recipe(airbnb_recipe) %>%
add_model(lm_spec)
```

Model Training and Prediction

- Fit: the recipe is **prepped** on the training data, and the model learns coefficients.
- Predict: the prepped recipe is **baked** on the test data and the fitted model outputs

```
 lm_fit <- fit(lm_wf, data = train)

test_pred <- predict(lm_fit, new_data = test) %>%
bind_cols(test)
```

Performance Metrics

- RMSE: root mean square error
- MAE: mean absolute error
- R^2: proportion of variance explained

```
preds <- predict(lm_fit, new_data = test) %>% bind_cols(test)
metrics(preds, truth = log(price), estimate = .pred)
```

Ames Housing Prices

- Contains 80 descriptive features covering site, structure, quality, and sale info
- Mix of numeric (areas, counts, years) and categorical/ordinal (“quality” grades like Ex/Gd/TA/Fa/Po).
- Variables include lot/frontage/area; overall quality; total finished square footage; basement/garage size and type; baths/bedrooms; fireplaces; roof; neighbourhood; year built/remodelled; month/year sold; sale condition.
  - Many NAs mean “not present” (e.g., PoolQC, Alley, Fence, FireplaceQu).
  - Some numeric codes are actually categorical (MSSubClass).
  - SalePrice is right-skewed; log-transform is typical.
- Goal: predict house sale price.

Scaling to Many Predictors

Challenges

- Large p (hundreds-thousands): multicollinearity, unstable coefficients.
- Coefficients become hard to interpret and inference p-values break after selection.

Strategies:

- Regularisation: ridge (stability), lasso (sparsity), elastic net (balanced).
- Dimensionality reduction: PCA or feature hashing for very wide data.
- Feature screening: drop near-zero variance, prune high correlations.
- Categoricals at scale: collapse rare levels; avoid leakage with any target/embedding encoders.
- Model choice: if strong nonlinearity/interactions, consider tree/boosted models.

What is Overfitting?

- The model memorises noise instead of learning the signal. This gives a great fit on training data, but poor generalisation.
  - Train error ≪ test/CV error (large generalisation gap).
  - Tiny changes in data can lead to very different coefficients/predictions (high variance).
- Overfitting happens when:
  - Model is too flexible/complex (many predictors, high-degree interactions).
  - Data is small relative to features
- Fix / prevent
  - Regularisation (ridge, lasso, elastic net) — tune with cross-validation.
  - Simplify the model (fewer predictors, lower degree, prune interactions; or use PCA).

What is Regularisation?

- Add a penalty to the training loss to discourage overly flexible models and improve generalisation.

$$min_\beta \sum_{i=1}^{n} \left(y_i \; - \; x_i^\top \beta\right)^2 + \lambda P(\beta)$$

- This increases bias a bit, but reduces variance a lot which leads to lower test error.
- Ridge (L2): $P(\beta) = \sum_j \beta_j^2$
- Lasso (L1): $P(\beta) = \sum_j |\beta_j|$
- Elastic Net: $P(\beta) = \lambda[\alpha \sum_j |\beta_j| + (1 - \alpha) \sum_j \beta_j^2]$

Ridge Regression (L2 Penalty)

- Ridge regression adds the squared magnitude of coefficients as a penalty:

$$\text{Loss} = \text{RSS} + \lambda \sum_j \beta_j^2$$

- This shrinks all coefficients toward 0 (rarely exactly 0).
- It is great for dealing with multicollinearity as it stabilises estimates and shares weight across correlated predictors.
- Reduces variance: often better test error; though it introduces small bias.
  - Standardise predictors before fitting (scale matters for L2).
  - Choose λ via cross-validation (typical U-shaped test error vs. λ).
- Works well when you have many small effects or $p > n$.

Lasso Regression (L1 Penalty)

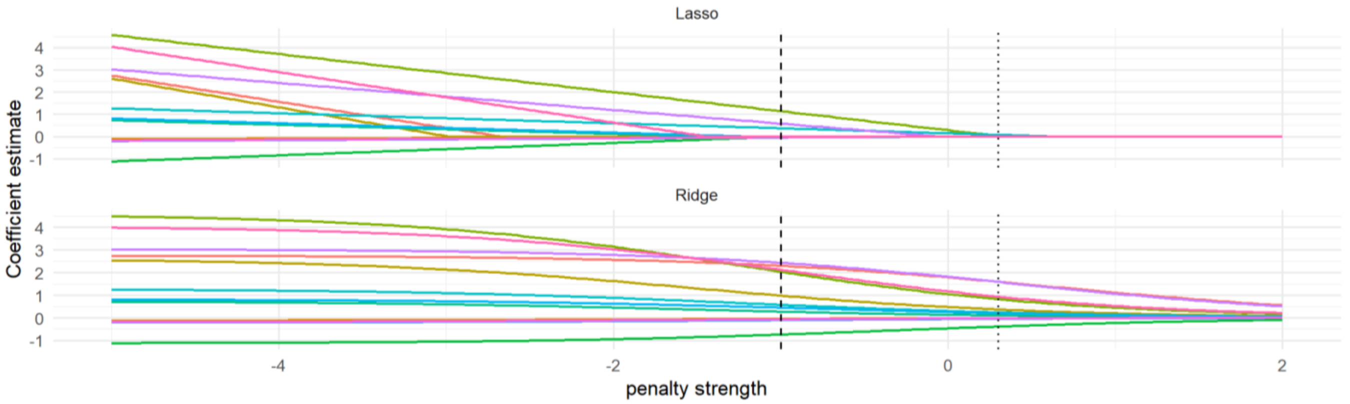
- Lasso adds the absolute values of coefficients:

$$Loss = RSS + \lambda \sum_{j} |\beta_j|$$

- Lasso drives some coefficients exactly to 0 ⇒ automatic variable selection.
- With correlated predictors, it tends to keep one and drop the rest (selection can be unstable).
- For groups of correlated features, consider Elastic Net (mix of L1/L2) for more stable selection.

Coefficient Path Plots

- A coefficient path shows each β_j as the penalty λ increases
- We use it to examine how shrinkage affects features
- Lasso (L1): many paths hit zero → variable selection.
- Ridge (L2): paths approach zero, but no exact zeros, stabilises collinearity.
- Features whose paths persist longer are stronger signals; early drop-offs are weak/noisy.



Interpreting λ

- λ controls the strength of regularisation in penalised regression.
- When λ = 0, the model reduces to ordinary least squares with no penalty applied. As λ increases, the penalty strengthens and all coefficients are pushed toward zero.
- In ridge regression, coefficients shrink smoothly toward zero but rarely become exactly zero, so all predictors remain in the model.
- In lasso regression, some coefficients are driven exactly to zero, which performs variable selection.
- With a small λ, the model has low bias and high variance, which increases the risk of overfitting.With a large λ, the model has higher bias and low variance, which increases the risk of underfitting.
- In cross-validation, the error plotted against log(λ) typically forms a U-shaped curve, so the best λ is near the minimum of this curve.

Elastic Net Regression

- Elastic Net combines ridge and lasso penalties:

$$Loss = \lambda[\alpha \sum_j |\beta_j| + (1-\alpha) \sum_j \beta_j^2]$$

- λ = overall penalty strength
- α controls the L1–L2 trade-off: Raising α leads to a more lasso-like model, while lowering α leads to a more ridge-like shrinkage.
- Shrinks coefficients and can set some to zero (variable selection).
- Stabilises estimates with correlated predictors (tends to keep groups).
- Often best when you have many, correlated features or $p \gtrsim n$.
- We need to tune both λ and α with cross-validation.

Choosing Between Ridge, Lasso and Elastic Net

- Ridge
  - Use when: you have many small, possibly correlated effects; $p \gtrsim n$.
  - Pros: stabilises coefficients under multicollinearity; rarely drops variables.
  - Cons: no sparsity (harder to interpret).
- Lasso
- Use when: you want a sparse, interpretable model; only a few predictors matter.
  - Pros: sets some coefficients exactly to 0 (feature selection).
  - Cons: with correlated predictors, selection can be unstable (keeps one, drops others).
- Elastic Net
  - Use when: predictors are correlated and you want selection + stability.
  - Pros: groups correlated features; balances ridge shrinkage with lasso sparsity.
  - Tune: both $\lambda$and α(mix). Start $\alpha \in [0.2, 0.8]$.

Defining Regularised Models with tidymodels

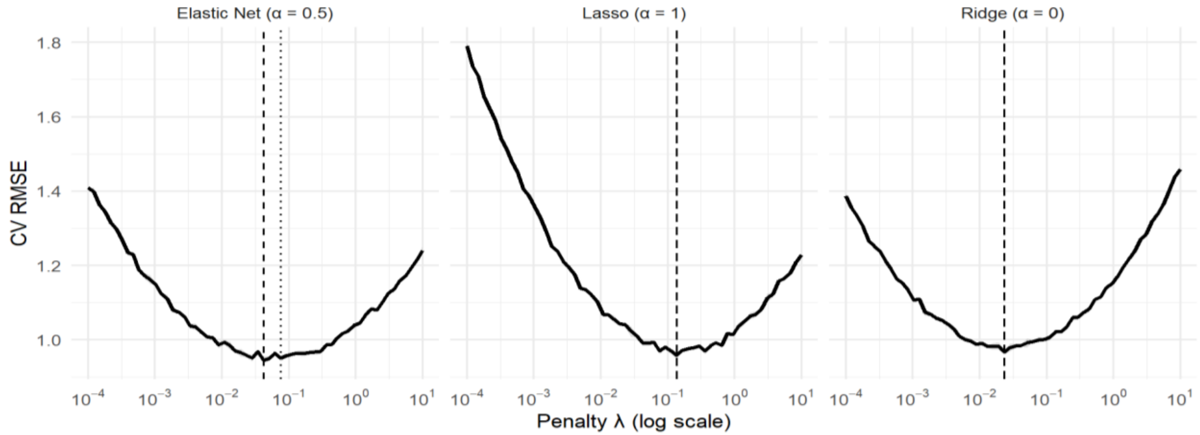
```
library(tidymodels)

ridge_spec <- linear_reg(penalty = tune(), mixture = 0)
%>% set_engine("glmnet")
lasso_spec <- linear_reg(penalty = tune(), mixture = 1)
%>% set_engine("glmnet")
elnet_spec <- linear_reg(penalty = tune(), mixture =
tune()) %>% set_engine("glmnet")
```

- Use tune_grid() with a recipe and resamples to tune hyperparameters.

Regularisation and Model Complexity

- The penalty λ acts as a dial for complexity. Smaller penalty means a more complex model, while larger penalties lead to simpler models (coefficients shrink)
- The curves show RMSE vs. λ for each mixture (α) panel - left side = overfitting, right side = underfitting
- The panel with the lowest minimum indicates whether ridge, lasso, or elastic net generalises best.



Limitations of Regularised Regression

- Still linear and additive: won’t learn curvature or interactions unless you engineer them.
- Bias from shrinkage: coefficients are pulled toward zero; large λ can underfit.
- Selection instability (lasso): with correlated predictors, lasso may keep one and drop others arbitrarily; results can vary by split/seed.
- Not robust to outliers/shift : penalties don’t fix outliers or distribution shift; consider robust losses (Huber/quantile) and monitoring.

When to Use Regularisation

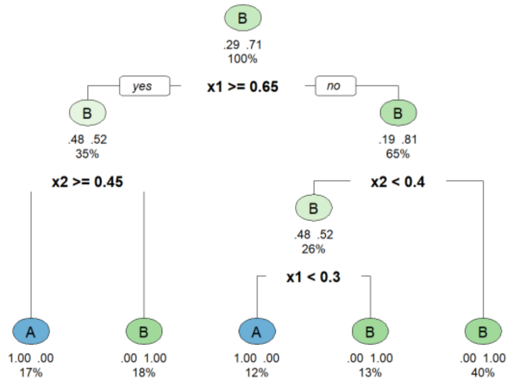
- Wide data: controls overfitting and keeps optimisation stable.
- Many irrelevant predictors: prefer lasso / elastic net (high α) for automatic variable selection.
- Multicollinearity / correlated groups: prefer ridge or elastic net (low–mid α) for stable coefficients.

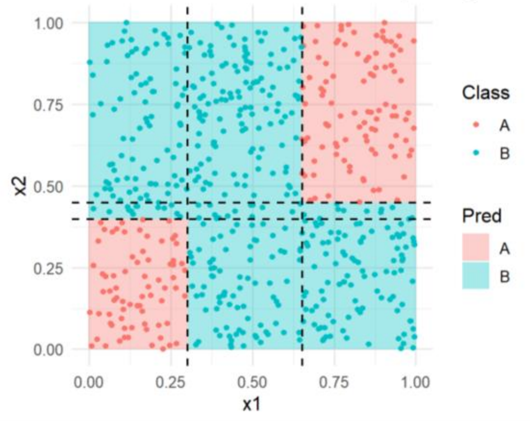
Rules of thumb:

- Ridge: many small, correlated effects or $p \gg n$.
- Lasso: few strong signals, want selection.
- Elastic Net: correlated predictors and want selection + stability.

Decision Trees for Regression

- The process begins with all data points at the root node.
- The algorithm evaluates all possible features and all possible split points for each feature.
- For each potential split, the algorithm calculates:
- $SSE(t) = \sum(y_i - \bar{y}_t)^2$
- $\Delta(s) = SSE(t) - \{SSE(left) + SSE(right)\}$.
- Pick the split with largest $\Delta(s)$ (biggest variance reduction between parent and child nodes).
- Repeat until a stopping condition is met.





Advantages of Trees

- Intuitive and visual: readable if-then rules and a picture of how predictions are made (follow a path to a leaf).
- Minimal preprocessing: no need to scale/standardise; handles numeric and categorical predictors.
- Nonparametric: no assumptions of linearity, additivity, or normal errors; naturally captures nonlinear effects and interactions.
- Great for insight/discovery: quick baseline, feature importance, and partial dependence/ICE to see which variables matter and how.

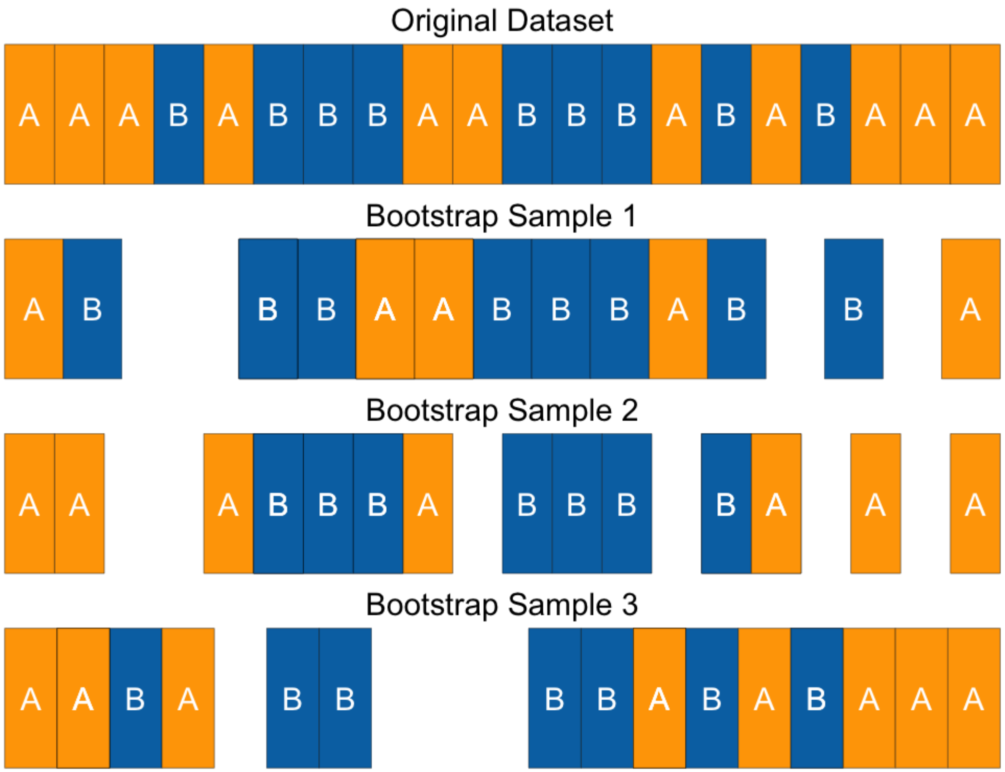
Limitations of Single Trees

- High variance / overfitting: deep trees fit noise; small data changes can yield very different trees.
- Stepwise, non-smooth predictions: poor extrapolation beyond the training range.
- Split bias: prefers variables with many possible cut points/levels; impurity importances can be misleading.
- Often weaker accuracy: a lone tree usually loses to ensembles on real tabular ML tasks.

Use-Cases for Tree-Based Models

- Many interactions, unclear functional form:
  - When effects depend on each other (e.g., price × season × location). Trees discover interactions automatically.
- Nonlinear relationships
  - Piecewise rules handle thresholds, plateaus, and saturation (e.g., “risk jumps after age 65”).
- Mixed, messy tabular data
  - Numeric and categorical features, outliers, missing values, and different scales with minimal preprocessing.

Bagging



Original Dataset

A A A B A B B B A A B B B A B A B A A A

Bootstrap Sample 1

A B B B A A B B B A B B A

Bootstrap Sample 2

A A A B B B A B B B B A A A

Bootstrap Sample 3

A A B A B B B B A B A B A A A

Random Forests

- A random forest builds many decision trees and combines their predictions.
- Each tree is trained on a bootstrap sample of the training data. While a tree grows, each split is chosen from a random subset of predictors of size mtry; the best split is picked only among those candidates.
- To predict, the forest aggregates the trees: average for regression, majority vote / mean class probability for classification.
- Averaging many noisy trees reduces variance. Randomly limiting predictors decorrelates the trees, so averaging cancels noise more effectively, lowering test error with only a small bias increase.

Hyperparameters in Random Forests

- mtry: predictors tried at each split - controls tree diversity.
  - Smaller mtry : trees see fewer features → more diverse, higher bias, lower variance.
  - Larger mtry : stronger individual trees but more correlation → can raise variance.
  - Start around √p (classification) or p/3 (regression)

- min_n: minimum samples in a terminal node - controls tree depth
  - Smaller min_n : deeper trees → lower bias, higher variance (risk overfitting).
  - Larger min_n : shallower trees → higher bias, lower variance (smoother predictions).
  - Start with a few values (e.g., 1–5–20) and pick via CV/OOB.

Gradient Boosting Machines

- GBMs build an additive model of many small trees, added stagewise to reduce a chosen loss.
- Start with a baseline prediction: mean(y) for regression.
- At iteration t, compute pseudo-residuals = y − ŷ
- Fit a shallow tree (weak learner) to those pseudo-residuals.
- Update predictions: $f_{t+1}(x) = f_t(x) + \eta \cdot tree_t(x)$
- Repeat for many iterations; each tree targets what the ensemble is still getting wrong.
- Predictions are aggregated by summing tree outputs (then applying the link: identity for regression, logistic for classification).

Common GBM Implementations

- XGBoost:
  - Widely used, stable defaults, great documentation; supports regularisation (L1/L2),

- LightGBM:
  - Very fast via histogram-based splits and leaf-wise growth; good for large datasets.
  - Leaf-wise growth can overfit if unconstrained : use num_leaves, min_data_in_leaf, feature_fraction, bagging_fraction.

Fitting a Tree Model

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

xgb_spec <- boost_tree(trees = 500,  learn_rate = tune(),
  tree_depth = tune(),   mtry = tune(),  min_n = tune()
) %>%
  set_engine("xgboost") %>%
  set_mode("regression")
```

Hyperparameters

- Hyperparameters are model settings chosen before training (not learned from the data).
- Examples: number of trees, learning rate, tree depth, regularisation strength (λ).
- Parameters vs. hyperparameters:
  - Parameters are fit by the algorithm (e.g., regression coefficients, node values in trees).
  - Hyperparameters control how the algorithm learns (capacity, regularisation, speed).
- Hyperparameters set the bias-variance trade-off and can make or break generalisation. A model that is too complex will overfit, while a model that is too simple will underfit.

Hyperparameters

Regularised linear models (ridge, lasso, elastic net)

- The penalty (λ) controls how strongly the coefficients are shrunk toward zero. Increasing λ applies more shrinkage, which typically increases bias and reduces variance.
- The mixture (α) determines the blend between ridge and lasso. An α of zero gives pure ridge, an α of one gives pure lasso, and values in between produce elastic net.

Hyperparameters

Random forest

- The trees setting determines how many trees are grown in the forest. Using more trees usually reduces variance until the out-of-bag or cross-validated error stops improving.
- The mtry setting specifies how many predictors are considered at each split. Using a smaller value increases tree diversity and often lowers variance at the cost of higher bias, whereas a larger value makes individual trees stronger but more correlated.
- The min_n (minimum samples per leaf) setting controls how deep trees can grow. Larger values create shallower, smoother trees, which increases bias and reduces variance.

Hyperparameters

Gradient boosting / XGBoost (xgboost)

- The learn_rate (eta) sets the step size for each boosting iteration. Smaller values usually generalise better, but they require more trees.
- The trees setting is the number of boosting rounds, and it should be paired with early stopping on a validation set to prevent overfitting.
- The tree_depth setting controls the complexity of each individual tree; for tabular data, shallow trees are usually preferred.
- The min_n and the loss_reduction settings act as regularisation.
- Row and column subsampling (sample_size and mtry/colsample_bytree) introduce randomness that reduces variance.
- A learning rate of 0.01-0.3, tree depth of 2-8, and subsampling rates of 0.5-1.0 are reasonable places to start.

Why Tune Hyperparameters?

- Maximise out-of-sample accuracy: Good settings reduce error on unseen data, not just the training set.
- Control bias–variance: Options like depth/λ/learning rate balance underfitting (too simple) vs overfitting (too complex).
- Stabilise the model: Proper regularisation and constraints make predictions less sensitive to noise and resampling.
- Fair model selection: Tuning puts each model on a level playing field so you compare their best versions.
- Adapt to data shape: Different datasets need different capacity (e.g., shallow vs deep trees; small vs large λ).

The 1-SE Rule

- From the CV results, find the model with the lowest mean error (e.g., RMSE).
- Compute its standard error (SE) across folds. Then choose the simplest model whose mean error is within 1 SE of that minimum.
- Tiny differences in CV means are often just noise. The 1-SE rule biases selection toward a simpler, more regularised model that's statistically indistinguishable from the best, improving robustness and generalisation.
- If the best CV RMSE is 0.210 with SE = 0.005, any model with CV RMSE ≤ 0.215 qualifies; among those, choose the simplest (e.g., larger λ, shallower trees, smaller depth).

Grid Search

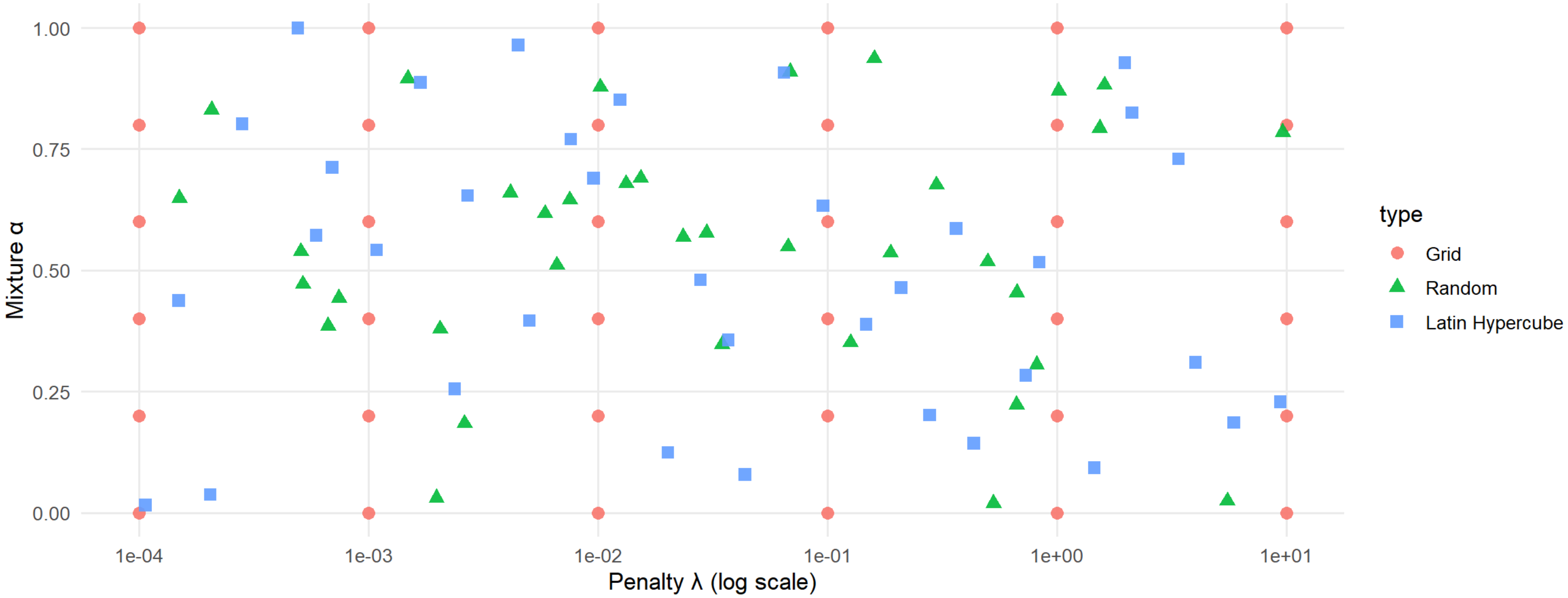
- Grid search involves picking a small, discrete set of values for each hyperparameter and train a model at every combination (one cell per combo).
- For each cell, compute metrics on the training data (same folds for every model) and summarise (mean ± SE).
- Select the lowest mean error (or the 1-SE rule for a simpler, more regularised model), then refit on all training data and evaluate once on the test set.
- Trade-offs: simple and reproducible, but computationally expensive as parameters multiply; for bigger spaces prefer random or Bayesian search.

Random Search

- Rather than providing values to test, random search takes a range of possible values and samples combinations at random.
- In many problems only a few hyperparameters really matter; random search finds good regions faster than an exhaustive grid and scales better as you add parameters.
- Nice properties: it’s anytime (stop when budget runs out), parallel-friendly, and lets you prioritise ranges (e.g., more mass on small learning rates).

Latin Hypercube

- Instead of testing every combo (grid) or sampling freely (random), LHS stratifies each hyperparameter’s range into equal-probability bins and takes one sample from each bin, then pairs them to spread points evenly across the joint space.
- The result is space-filling coverage with far fewer evaluations than a full grid and less clumping than pure random.
- Finds good regions quickly when you have several hyperparameters (3–20+) and a fixed budget.
- Great default when you want coverage without combinatorial blow-up.


1.00
0.75
0.50
0.25
0.00
Mixture α
1e-04
1e-03
1e-02
1e-01
1e+00
1e+01
Penalty λ (log scale)
type
Grid
Random
Latin Hypercube

Creating a Grid in tidymodels

- Build an evenly spaced grid of values for each tunable parameter:

```
grid <- grid_regular(penalty(), mixture(), levels = 10)
```

- Keep an eye on how big the grid is. Total fits = product of levels across parameters

- Random alternative: `grid_random()` samples from those ranges.

- Latin hypercube: `grid_latin_hypercube()`

Finalising and Fitting the Model

- Pick the winner:
  - `select_best(results, metric = "rmse")`
- Lock the settings and refit on all training data:
  - `finalize_workflow(workflow, best_params)`
  - `fit(final_model, data = training_data)`
- Evaluate exactly once on test:
  - Use `predict()` on the test set and score with yardstick metrics. If you transformed the outcome (e.g., log-price), either back-transform predictions or compute metrics on the same scale used for training—just be consistent.

Efficient Tuning

- Plan the search
  - Start coarse and then refine
  - Use random search for large spaces; reserve grids for few parameters.
- Make each fit cheaper
  - For GBMs, use early stopping on a validation fold; cap trees but allow stopping early.
  - Fix sensible defaults (e.g., enough RF trees to stabilise OOB error) and tune the few parameters that matter (mtry, min_n, λ, learn rate, depth).
- Search smarter
  - Prefer the 1-SE rule for a simpler, more robust model if differences are tiny.
  - If the “best” lies on a boundary, expand the range and rerun.

Overfitting During Tuning

- When you try many hyperparameter settings, one might “win” on the CV folds partly by luck. You’ve effectively fit to the folds. This can mean great CV score for the chosen settings but worse performance on unseen data.
- Hold out a test set and evaluate once after tuning is finished
- For an unbiased performance estimate, use nested cross-validation:
  - Inner loop: tune hyperparameters.
  - Outer loop: assess the tuned model on held-out folds (no tuning here).
- Prefer simpler settings when differences are tiny (use the 1-SE rule).
- Keep search spaces reasonable; huge grids amplify overfitting to the folds.

Automating Tuning

- tidymodels supports adaptive resampling:
  - Don’t fully evaluate every candidate on all folds. Instead, evaluate on a few folds, run a statistical screen (e.g., ANOVA on CV errors), drop clearly worse settings, and continue only with the survivors.
- Bad configurations are pruned early, so compute is spent where it matters; this is great for large grids or expensive models.

```
results <- tune_race_anova(
workflow, resamples = folds, grid = grid)
```

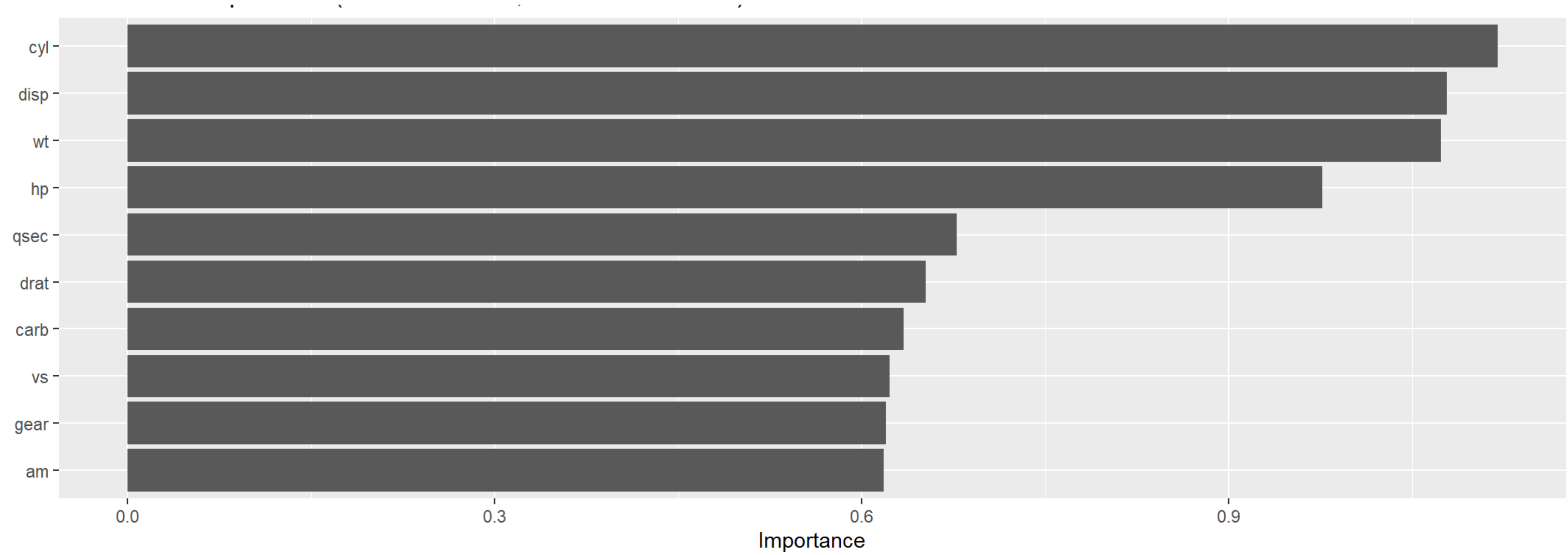
Practical Considerations

- Decide how much time/compute you can spend; use random/racing searches and early stopping to stay within it.
- Small datasets need restraint. Prefer (repeated) CV or nested CV, use the 1-SE rule, and avoid huge grids.
- Preprocess inside the workflow. Standardise where needed, impute, and keep any outcome transforms consistent between training and scoring.
- Prefer simpler settings if performance is within tolerance; check latency/memory and keep a fallback baseline.

Variable Importance

- Variable importance ranks predictors by how much they contribute to the model's predictive accuracy. Trees/ensembles expose split-based scores. These are quick diagnostics but can be biased and are not always comparable across models/datasets.
- Permutation importance is model-agnostic and directly tied to predictive impact on unseen data.
  - Compute a baseline metric on validation data.
  - Shuffle a feature (leave all other features intact) to break its association with the target.
  - Recompute the metric; VI = (permuted loss − baseline loss)
  - Repeat the shuffle several times, average, and report a standard error. Larger increases in loss ⇒ stronger predictive dependence. VI reflects association, not causation.

Variable Importance



Evaluating Models

- Focus on generalisation, not fit: Low training error can still mean overfitting. Judge models by performance on unseen data.
- Use proper splits: Train/validation/test or k-fold CV.
- Tune hyperparameters only on training/CV. After choosing the best fit, refit on all training data and evaluate once on the test set.
- Pick the right metrics: Report RMSE/MAE (units of y); use R^2 for explanatory power. Prefer MAE if outliers matter less; RMSE if large errors are costly.

Ranking Models with yardstick

- Idea: Instead of comparing raw CV metrics, rank each candidate within every fold
- In each fold, rank all candidates by the primary metric (e.g., RMSE: rank 1 = lowest RMSE in that fold). Average the per-fold ranks to get a mean rank for each model or tuning setting; lower mean rank is better.
- Ranking is more stable than raw metrics across folds because it is less sensitive to scale differences and outliers, and it works well when comparing many settings with very similar performance.
- However, ranks hide magnitude. Two models can have similar ranks while one has a much better metric. Always show mean ± SE of the raw metric alongside mean rank.

Overfitting Check

- Train–validation gap: Training RMSE is much lower (or R^2much higher) than CV/test; the model is fitting to noise in the training data.
- Learning curves: As training size grows, train error rises and validation error falls; a persistent wide gap indicates high variance/overfit.
- Unstable CV: Large fold-to-fold SD/SE, or model ranks flip across repeats/resamples.
- Too good to be true: Near-perfect train R^2or tiny train RMSE with only mediocre validation performance.