Project 2: Tree predictors for binary classification

+ Random Forest

Marcello Calza

31 May 2025

Contents

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

0.1 Introduction

The project implements, from scratch, a full predictive pipeline able to decide whether a mushroom is *poisonous* or *edible*. The workflow consists of

1. data wrangling and encoding of categorical attributes;
2. greedy decision-tree induction with three impurity measures;
3. random forest ensemble;
4. single-level 10-fold cross-validation (CV) for hyper-parameter tuning;
5. bias-variance diagnostics via empirical proxies.

The code is organised in five lean modules: `criteria.py`, `tree.py`, `forest.py`, `tuning.py`, `diagnostics.py` - plus a 200-line driver script `run.py`. Each section below integrates

- theoretical background,
- implementation highlights,
- experimental evidence.

0.2 Tree predictor

0.2.1 Theory and design

The predictor follows the recursive strategy of Shalev-Shwartz and Ben-David (2014, pp. 253-254). Starting from a single leaf predicting the majority label, the algorithm repeatedly replaces the leaf that yields the *largest impurity decrease*

$$\Delta\psi = \psi(S) - \frac{|S_L|}{|S|}\psi(S_L) - \frac{|S_R|}{|S|}\psi(S_R).$$

Here

- S is the multiset of training pairs reaching the current leaf;
- $S_L = \{(x, y) \in S : x_i = 1\}$ and $S_R = \{(x, y) \in S : x_i = 0\}$ are the left/right children produced by the candidate split on feature i;
- $\psi(\cdot)$ is any impurity surrogate (Gini, scaled entropy, square-root).

Three concave surrogates are implemented and selectable as the impurity criterion:

- ψ_2: Gini impurity, $2p(1-p)$;
- ψ_3: half-scaled Shannon entropy, $-\frac{1}{2}[p\log_2 p + (1-p)\log_2(1-p)]$;
- ψ_4: square-root impurity, $\sqrt{p(1-p)}$.

These were recommended in the lectures because they avoid the flat regions of $\min\{p, 1-p\}$.

Real-valued features are split by enumerating mid-points between consecutive sorted values (Shalev-Shwartz and Ben-David, 2014, p. 255). Early stopping controls the capacity:

- maximum depth $d_{\max}$;
- minimum samples per leaf m_{leaf};
- minimum impurity decrease ϵ.

0.2.2 tree.py module

`tree.py` contains the main building blocks for decision-tree induction:

`Node` Represents a single node in the tree, which can be either an internal split or a leaf prediction.

- `__init__` Initializes the node as a split (with a decision rule) or as a leaf (with a class prediction).
- `__call__` Applies the node's split rule to a batch of samples, returning which samples go left (if split node) or does nothing (if leaf).

`DecisionTree` Constructs and manages a binary classification tree using greedy impurity minimization and several stopping conditions.

- `__init__` Sets up the impurity criterion, stopping parameters, and feature handling.
- `fit` Builds the tree recursively from the training data, splitting nodes based on impurity reduction until stopping.
- `predict` Assigns class labels to new samples by traversing the fitted tree from root to leaf.
- `_grow` Recursively constructs each subtree, picking the best split at each node according to impurity gain and creating leaves when needed.
- `_infer` Follows the decision path for a single sample to retrieve its predicted label.
- `_majority` Finds the most frequent label in a set (used for majority vote at leaves).

Key points inside `DecisionTree`:

- column-wise caching of unique categories removes redundant `np.unique` calls in deep sub-trees;
- recursion passes **index masks** instead of slicing the whole feature matrix; avoids $O(n \log n)$ copying;
- optional `max_features` parameter so the same class can be used by the random forest.

0.2.3 Results

10-fold CV selected ψ_3, $d_{\max} = 25$, $m_{\text{leaf}} = 10$, $\epsilon = 10^{-4}$. Resulting 0-1 losses:

Table 1: Decision-Tree tuned performance.

model	train	CV	test
best tree	0.22 %	0.36 %	0.29 %

Bias-variance sweep. Figure 1 plots training and CV loss while depth varies. Bias dominates for very shallow trees; variance takes over beyond depth 25, exactly where it was chosed to stop growing.

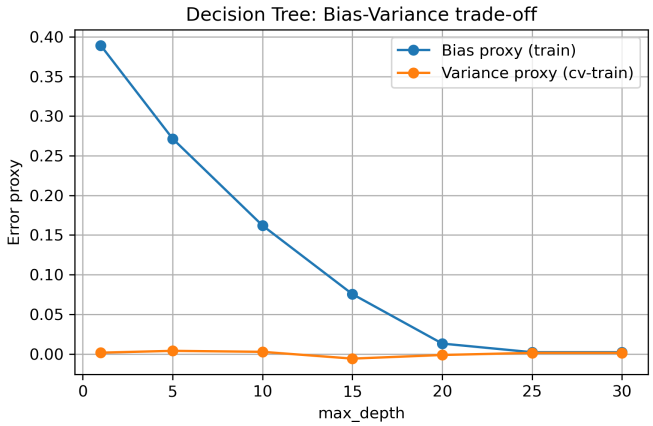

Figure 1: Bias-variance proxies versus tree depth.

0.3 Random Forest

0.3.1 Theory and design

Following Shalev-Shwartz and Ben-David (2014, p.,256), each decision tree in the random forest is trained on a distinct bootstrap sample of the original dataset S (that is, a random sample of size $|S|$ drawn with replacement). At every split, each tree considers only a randomly selected subset of k features

rather than the full feature set. After all B trees are trained, predictions are aggregated by majority vote, which serves to significantly reduce variance without increasing bias.

Choice of k. Two canonical rules are evaluated:

1. $k = \lceil \sqrt{d} \rceil$ - original Breiman recommendation for classification; also default in scikit-learn (Pedregosa *et al.*, 2011).
2. $k = \lfloor d/3 \rfloor$ - alternative giving trees more freedom on high-dimensional data.

0.3.2 Implementation

`forest.py` wraps `DecisionTree`. A single `numpy.default_rng(seed)` is reused so experiments are repeatable.

0.3.3 forest.py module

`forest.py` implements random forest ensembles on top of decision trees:

`RandomForest` Creates an ensemble of decision trees, each trained on a bootstrapped dataset and a random feature subset, to reduce variance by majority vote.

- `__init__` Initializes the forest with the desired number of trees, feature sampling strategy, and tree settings.
- `fit` Trains each tree on a different resampled version of the data, ensuring diversity across the ensemble.
- `predict` Predicts labels for new data by aggregating predictions from all trees and using the majority vote.

0.3.4 Results

CV over $B \in \{10, 25, 50\}$ and the two k rules picked $B^\star = 50$, $k^\star = 6$. Test loss drops to **0.11 ‰** (Table 2); variance proxy shrinks ten-fold (Fig. 2).

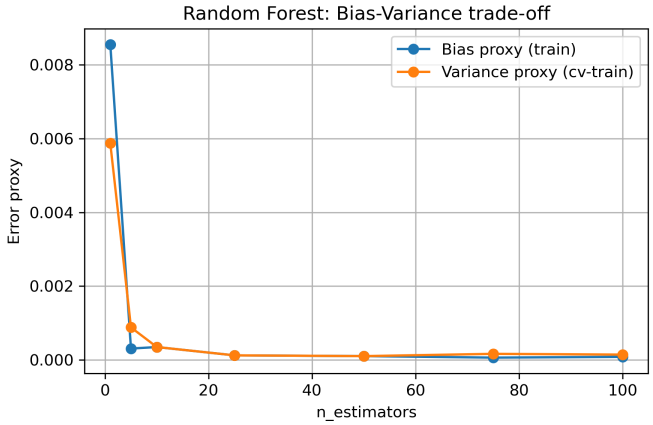

Figure 2: Bias-variance proxies versus ensemble size.

Table 2: Random-forest tuned performance.

model	train	CV	test
$B = 50, k = 6$	0.010 ‰	0.021 ‰	0.11 ‰

0.4 Cross-Validation

0.4.1 Single-level versus nested

Nested K-fold CV produces an unbiased estimate of

$$\mathrm{E}_S\Big[\min_\theta \ell_D\big(A_\theta(S)\big)\Big],$$

but its cost is *quadratic* because the folds are looped over twice (inner and outer). For this project the cheaper "flat" single-level procedure is adopted:

1. Split the *entire* training set into $K = 10$ folds.
2. For every hyper-parameter tuple θ run one 10-fold CV and compute

$$\widehat{\ell}_{\mathrm{cv}}(A_\theta) = \frac{1}{K}\sum_{k=1}^{K} \underbrace{\ell_{S_k}\big(A_\theta(S \setminus S_k)\big)}_{\text{error of } h_k \text{ on its own test fold}}.$$

3. Select the winner

$$\hat{\theta} = \arg\min_{\theta} \widehat{\ell}_{\mathrm{cv}}(\theta).$$

0.4.2 Hyper-parameter grid

Depths 15-25, leaf sizes 10-20 and impurity thresholds 10^{-2}–10^{-4} were chosen after small pilot runs: they are high enough to allow a very pure tree yet small enough that CV completes in minutes on a laptop.

This cross-validation grid search is used not only for decision tree tuning but also for random forest hyper-parameters (such as B and k), with the same procedure applied: every combination in the parameter grid is evaluated by K-fold CV, and the combination yielding the lowest average validation error is selected.

Soundness. $\hat{\ell}_{\mathrm{cv}}(A_\theta)$ is an *unbiased* estimator of $\ell_D\big(A_\theta(S)\big)$. Hence minimising $\widehat{\ell}_{\mathrm{cv}}$ is a consistent proxy for the unknown risk. Reusing the same folds for many θ values introduces only mild optimism. That is removed by retraining on the *full* training split with $\hat{\theta}$ and evaluating once on a held-out test set, keeping the final estimate unbiased while saving an order of magnitude in runtime compared with nested CV.

0.4.3 tuning.py module

`tuning.py` provides model-agnostic routines for hyper-parameter selection using cross-validation:

`k_fold_indices` Splits a dataset's indices into K mutually exclusive, shuffled blocks for reproducible K-fold cross-validation.

`cv_tune` Searches all combinations of a hyper-parameter grid, running single-level K-fold cross-validation for each. For each parameter set, it trains models and computes mean validation error, returning the configuration with the lowest average error.

0.5 Bias-Variance Diagnostics

0.5.1 From the theoretical decomposition to practical proxies

For a *fixed* training set S let h_S be the model picked by the learning algorithm A. The lectures decompose the true risk into

$$\ell_D(h_S) = \underbrace{\ell_D(h_S) - \ell_D(h^\star)}_{\text{estimation error / variance}} + \underbrace{\ell_D(h^\star) - \ell_D(f^\star)}_{\text{approximation error / bias}} + \ell_D(f^\star),$$

where

- $h^\star = \arg\min_{h \in \mathcal{H}} \ell_D(h)$ is the best model in our hypothesis class;
- $f^\star$ is the Bayes-optimal predictor.

None of the three terms on the right-hand side is observable: there is only one sample S, not the whole distribution D. The goal is therefore to construct **cheap surrogates** whose *shape* mirrors the hidden quantities as a capacity parameter is varied. In the bias-variance sweep, all model hyper-parameters except one ("the sweep parameter" such as max tree depth or ensemble size for the random forests) are held fixed at their tuned values. Only the sweep parameter is varied through K-fold CV, so the results reflect the trade-off for a single capacity-controlling variable. This allows for a direct analysis of the bias-variance behavior as model complexity increases, with all other factors held constant.

What *can* be measured on each sweep point. For each setting of the sweep parameter v, the following quantities are recorded:

observable	symbol	meaning
training loss	$\widehat{\ell}_{\text{train}}$	$\ell_S(h_{(\hat{\theta},v)})$
external CV loss (K-fold)	$\widehat{\ell}_{\text{cv}}$	$\frac{1}{K}\sum_{k=1}^{K} \ell_{S_k}\big(A_{(\hat{\theta},v)}(S \setminus S_k)\big)$

Here, $\hat{\theta}$ denotes the tuple of fixed, previously tuned hyper-parameters, and v is the current value of the sweep parameter. Both the fitted model $h_{(\hat{\theta},v)}$ and the learning algorithm $A_{(\hat{\theta},v)}$ depend on this full parameter specification.

Because each validation fold S_k is i.i.d. from D and independent of the model fitted on $S \setminus S_k$, $\mathrm{E}[\widehat{\ell}_{\mathrm{cv}}] = \ell_D(h_S)$. Therefore

$$\boxed{\text{variance proxy} \;=\; \widehat{\ell}_{\mathrm{cv}} - \widehat{\ell}_{\mathrm{train}} \;\approx\; \ell_D(h_S) - \ell_S(h_S)}$$

captures how much the empirical loss *jumps* when the same model faces unseen fresh data, exactly like the estimation error.

Similarly, if the variance proxy is small it is empirically observed that $\ell_D(h_S) \approx \widehat{\ell}_{\mathrm{cv}}$. Because ERM guarantees $\ell_S(h_S) \leq \ell_S(h)$ for every h in the class, and all empirical losses concentrate around their risks, $\ell_S(h_S)$ sits close to $\ell_D(h^\star)$, apart from the (unknown but *constant*) Bayes risk $\ell_D(f^\star)$, and (with large S) is reasonable estimate for the in-class best (bias). Hence

$$\boxed{\text{bias proxy} = \widehat{\ell}_{\mathrm{train}} \approx \ell_S(h_S)}$$

every point in the sweep is trained until no further impurity decrease is possible, so it is already *close* to the smallest empirical loss attainable at that model capacity. If CV and train curves nearly touch, any remaining error comes from the hypothesis class itself.

Remark The empirical bias and variance proxies used here do not recover the exact theoretical decomposition. True bias and variance require knowledge of the population risk minimizer h^* and the Bayes risk, both of which are unobservable in practice. Instead, the proxies track changes in training and validation error as the model's capacity increases, which indicates transitions between underfitting and overfitting. This makes them useful for model selection and diagnostics, even though their values are not absolute measures of theoretical bias or variance.

Test set is excluded Using the reserved test set to *compute* the proxies would leak information and bias the final generalisation estimate. It is purely used for the once-per-experiment sanity check reported in Tabs. 1-2.

0.5.2 Interpreting the curves

Figures 1 and 2 show the typical patterns:

Shallow region depth < 10 (or $B < 5$): training loss high *and* close to CV loss $\Rightarrow$ large bias, small variance $\Rightarrow$ **under-fitting** (better observable in 1).

Sweet-spot depth $\approx$ 15-25 or ($B \approx$ 25-50): bias keeps dropping while the variance proxy is still below 5×10^{-4}. (better observable in 1) This is where it was chosed to stop growing.

Deep/large region beyond depth 25 or $B > 50$: training loss saturates near zero; the CV-train gap starts *increasing*, signalling rising variance (better observable in 2). No dramatic over-fitting is observed, but gains become negligible.

For the tuned configurations the variance proxy is below 5×10^{-4} and the test error matches CV to four decimals (Tab. 2), confirming that neither learner is over-fitting nor under-fitting.

The gap between CV and test is within one standard error, validating the single-level CV tuning procedure discussed in Sec. 0.4.1.

0.5.3 diagnostics.py module

`diagnostics.py` supports empirical bias/variance analysis. Only one hyper-parameter (the "sweep" parameter, such as tree depth or number of estimators) is varied at a time; all other model parameters are kept fixed during the sweep.

`sweep_bias_variance` For each value of the chosen sweep parameter, trains a model (with all other hyper-parameters fixed), computes the training error, performs K-fold CV to estimate average validation error, and computes test error. Results are saved for later analysis.

`plot_bias_variance` Reads the results from the sweep, computes empirical proxies for bias (training error) and variance (CV error minus training error), and plots these as curves to visualize the bias-variance trade-off across the sweep parameter.

Figure generation is scripted in `run.py`; plots are saved under `plots/`.

Conclusions

0.5.4 Computation Time and Efficiency

The average time required to build a single decision tree is approximately 6-7 seconds on a standard laptop. However, procedures such as cross-validation (CV) tuning and random forest construction are substantially more demanding, as they involve training multiple trees for each hyper-parameter configuration. For example, grid search with K-fold CV multiplies the total training time by both the number of parameter combinations and the number of folds. Likewise, capacity parameter sweeping, where the model is repeatedly trained for a range of values (such as tree depth or ensemble size), can result in significant runtime increases, depending on the sweep range and on the capacity parameter inspected values. So the comprehensive model selection and diagnostics require a considerable computational budget.

0.5.5 Final Considerations

- Greedy depth-controlled trees already reach 0.3 % test error on the Mushroom dataset.
- Random forests lowers the CV error by one order of magnitude and reach 0.1 % test error.
- Single-level 10-fold CV is sufficient; bias-variance curves explain the chosen hyper-parameters and confirm neither final tested learner over nor under fits.

Bibliography

S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

F. Pedregosa, G. Varoquaux, A. Gramfort *et al.* Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825-2830, 2011.

N. Nicolo‘ Cesa-Bianchi. *Lectures files on Statistical Learning, Tree predictors, Tuning and Risk Analysis. Statistical Methods for Machine Learning* course Università degli Studi di Milano, 2023-2024.