Tree predictors for binary classification

Experimental project

Aleksandr Dudakov

2025-03-31

Abstract

This paper investigates decision trees and random forests for binary classification of mushroom edibility. Using a carefully preprocessed dataset of 61,069 mushroom observations, I implement custom tree predictor classes in Python with support for Gini and other impurity functions. Rigorous hyperparameter tuning-using both cross-validation (for decision trees) and out-of-bag estimation (for random forests)-effectively balances bias and variance. Experimental results show near-perfect classification by a single decision tree (test loss of 0.00082) and a fully accurate random forest (test loss of 0.00000), underscoring the high separability of the dataset and the ability of the models to exploit informative features. The results demonstrate that tree-based methods can achieve state-of-the-art performance.

University of Milan
Department of Economics, Management and Quantitative Methods (DEMM)
Machine Learning
Prof. Nicolò Cesa-Bianchi

UNIVERSITÀ
DEGLI STUDI
DI MILANO

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.

1 Introduction

Decision trees and their ensemble variants are fundamental techniques in machine learning, providing a balance between interpretability and predictive power. In this project, I investigate tree-based classification methods for determining whether mushrooms are edible or poisonous. Specifically, I develop and analyze custom implementations of decision trees and random forests, using systematic hyperparameter tuning - via cross-validation and out-of-bag error estimation - to effectively manage the bias-variance tradeoff.

The Mushroom dataset from the UCI repository will serve as the experimental environment. The main objectives are to (i) validate the correctness of the theoretical framework underlying tree-building and ensemble methods, (ii) systematically explore splitting and stopping criteria, and (iii) demonstrate that careful tuning can lead to near-perfect classification performance. The following sections detail the data preprocessing steps, describe the methodological approach, and present empirical results that highlight the effectiveness of tree-based predictors for this binary classification task.

2 Data

I use the Mushroom dataset provided by the UCI Machine Learning Repository, which consists of 61,069 mushroom observations evenly distributed across 173 hypothetical species (353 mushrooms per species). Each mushroom is labeled as either edible (e) or poisonous (p). The poisonous class also includes mushrooms of unknown edibility for safety reasons. The dataset has a slight class imbalance: 55.49% are poisonous and 44.51% are edible.

The dataset originally contains 20 features, of which 17 are categorical and 3 are numerical. Categorical features include characteristics such as cap shape, cap color, gill attachment, habitat, and seasonal occurrence, while numerical features include cap diameter (cm), stem height (cm), and stem width (mm). Several categorical characters have substantial missingness; in particular, `stem-root`, `spore-print-color`, and `gill-type` have less than 20% non-null observations.

Initially, I handled missing categorical values by introducing an "unknown" category and converting categorical variables to pandas `category` datatype to improve computational efficiency. However, preliminary tests showed that despite this optimization, prediction times remained unreasonably long. To significantly improve computational

efficiency, I adopted a one-hot encoding strategy for all categorical variables and converted all variables to integers. As a result, the processed feature matrix expanded from 20 original columns to 119 columns (with 116 dummy variables), significantly increasing dimensionality but dramatically improving runtime performance. The final preprocessed dataset consists entirely of numeric binary features, which facilitates fast computation.

For robust evaluation of the models, I partitioned the dataset into a training set (80%, 48,855 observations) and a test set (20%, 12,214 observations). Stratified sampling was used to maintain class proportions across both subsets, ensuring reliable estimation of model generalization performance. The splits were performed with a fixed random seed to ensure reproducibility. This careful data partitioning allows for unbiased evaluation and proper hyperparameter tuning, with no leakage from test data into the training process.

3 Methodology

In this project, I solve a binary classification problem using decision tree predictors implemented in Python. My approach is based on the theoretical framework presented in the lecture notes (Cesa-Bianchi, 2024).

3.1 Tree Predictor Implementation

3.1.1 Node Structure

Each node of the decision tree is encapsulated in a `Node` class with the following attributes

- **decision_fn**: A function that evaluates a single feature test, e.g. for numeric features $x[i] < \theta$ or for categorical features $x[i] == \theta$, and returns a boolean value.
- **test_feature** and **test_value**: The feature and associated threshold (or category) used to perform the split.
- **left** and **right**: Pointers to the left and right child nodes, respectively.
- **value**: For a leaf node, stores the predicted class label determined by a majority vote over the training examples routed to that node.
- **is_leaf**: A boolean flag indicating whether the node is a leaf.

This design allows for recursive prediction, where each input is routed from the root to a terminal leaf, yielding the corresponding prediction.

3.1.2 Splitting Criterion

The key step in growing the tree is the selection of an optimal split at each node. For a given node with training examples, we compute the impurity using a function ψ that evaluates the contribution of misclassification error. Several choices of ψ are considered:

- **Gini function:** $\psi_{\text{gini}}(p) = 2p(1-p)$.
- **Scaled Entropy:** $\psi_{\text{entropy}}(p) = -0.5\,(p\log_2(p) + (1-p)\log_2(1-p))$.
- **Square root function:** $\psi_{sqrt}(p) = \sqrt{p(1-p)}$.

Although the function defined as $\psi_{\min}(p) = \min\{p, 1-p\}$ is a natural candidate, its linear segments (the lack of curvature) can result in negligible error reduction, potentially stalling the growth process. Therefore, nonlinear ψ functions (with strictly negative second derivatives) are preferred, as they yield a more pronounced improvement when splitting.

3.1.3 Stopping criteria

The recursive splitting of nodes is controlled by several stopping conditions:

- **Maximum depth:** The recursion will stop when a specified maximum depth is reached.
- **Minimum samples per node:** Splitting will stop when the number of samples in a node falls below a specified threshold.
- **Minimum Impurity Reduction:** A split is only accepted if the impurity decrease exceeds a given threshold.
- **Purity of Node:** If a node becomes pure (i.e. all examples belong to the same class, resulting in zero impurity), no further splitting is performed.

These criteria ensure that the tree does not become unnecessarily complex, which is critical to avoid overfitting.

3.1.4 Best Split Selection and Tree Construction

The construction of the decision tree is driven by a recursive process that selects the optimal split at each node. This process is encapsulated in the `_build_tree` method of the

`TreePredictor` class. At each node, the algorithm searches for the best split by iterating over candidate features. For numerical features, candidate thresholds are generated by sorting the unique values of the feature; for categorical features, membership tests are considered (not used in the final version).

For a given candidate split on feature i with threshold θ, the training examples at the current node are divided into two subsets:

$$\begin{aligned} \text{Left subset:} \quad & \{x \mid x[i] < \theta\} \\ \text{Right subset:} \quad & \{x \mid x[i] \geq \theta\} \end{aligned}$$

The impurity of the parent node is computed using the chosen ψ function. For each candidate split, the impurity of the left and right child nodes is computed, and the weighted average impurity is computed:

$$\psi_{\text{weighted}} = \frac{n_{\text{left}}}{n}\psi(\text{left}) + \frac{n_{\text{right}}}{n}\psi(\text{right}),$$

where n, n_{left}, and n_{right} are the number of samples in the parent, left, and right nodes, respectively. The impurity reduction (or error reduction) due to the split is given by:

$$\Delta\psi = \psi(\text{parent}) - \psi_{\text{weighted}}.$$

The candidate split that maximizes $\Delta\psi$, provided it exceeds a given minimum impurity reduction threshold, is selected. If no candidate yields a sufficient decrease in impurity, or if a split would result in an empty child node, the node is declared a leaf, and its prediction is set as the majority class of the examples in that node.

3.2 Model Training and Hyperparameter Tuning

The tree predictor is trained by a recursive procedure implemented in the `fit` method. At each node, the algorithm selects an optimal split by evaluating a chosen impurity function ψ. Once a split is chosen, the node is partitioned into two child nodes, and the procedure continues recursively until one or more stopping criteria are met (e.g., maximum depth reached, node purity reached, minimum number of samples, or a minimum impurity decrease threshold).

The performance of the tree predictor is measured by the **0–1 loss**, defined as:

$$\text{Loss}_{0-1} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}\{h(x_i) \neq y_i\},$$

where $h(x_i)$ is the prediction for the i-th sample, y_i is the true label, and $\mathbf{1}\{\cdot\}$ is the indicator function. This loss function provides a direct measure of the misclassification rate.

To optimize the model and control for overfitting/underfitting and the bias-variance tradeoff, I run two types of experiments:

Baseline experiments. First, baseline experiments are performed by training the tree predictor on the full training set while varying key hyperparameters over predefined grids:

- **Maximum Depth Grid:** $\{15, 20, 25, 30, 35, 40, 45\}$.
- **Minimum Impurity Grid:** $\{0.0, 0.005, 0.01\}$.

For each hyperparameter configuration, a model is trained and its training loss is computed using the 0–1 loss. These experiments show the capacity of the model to fit the training data.

Cross-Validation Experiments. To obtain a robust estimate of the model's generalization performance and to balance the bias-variance tradeoff, I perform k-fold cross-validation (with $k = 5$) on the same hyperparameter grids. In each fold, the training set is partitioned into a training subset and a validation subset. The model is trained on the training subset, and its performance is evaluated on the validation subset using the same 0–1 loss. The cross-validation loss is then computed as the average loss over all folds. This approach allows us to select the optimal hyperparameter configuration that minimizes the validation loss, thereby reducing the risk of overfitting and ensuring that the model generalizes well to unseen data.

By comparing the training loss with the cross-validation loss, we gain insight into the overfitting or underfitting tendencies of our tree predictor. The selected hyperparameters - those with the lowest average cross-validation loss - are then used to retrain the model on the full training set before final evaluation on the test set.

3.3 Random Forest Predictor Implementation

To further improve performance, I extend the tree predictor to a Random Forest ensemble. In the `RandomForestPredictor` class, multiple tree predictors are aggregated and the final prediction is obtained by majority voting.

Key aspects of the Random Forest implementation include

- **Bootstrap Sampling:** Each tree is trained on a bootstrap sample drawn with replacement from the training set. This introduces variability among the trees, as each tree is exposed to a different subset of the data, which helps reduce overfitting.
- **Random Feature Selection:** A random subset of features, controlled by the parameter `max_features`, is considered at each split. This constraint reduces the correlation between individual trees, thereby reducing the variance of the ensemble.
- **Out-of-Bag (OOB) Loss Estimation:** Because each tree is trained on a bootstrap sample, about one-third of the training samples are omitted (i.e., they are "out-of-bag"). These OOB samples provide an unbiased estimate of the error for each tree. Aggregating the OOB loss over the ensemble provides an effective approximation of the generalization performance, eliminating the need for a separate validation set during hyperparameter tuning.

The `max_features` parameter is tuned over the grid {10, 15, 20, 30, 60, 90, 119} to achieve a favorable balance between variance reduction and potential bias increase. The OOB loss works well because the samples excluded from training each tree act as a natural validation set, reflecting the model's performance on unseen data. This built-in mechanism facilitates efficient hyperparameter tuning.

4 Results

Table 1 summarizes the optimal hyperparameter settings determined by 5-fold cross-validation (for decision trees) and out-of-bag (OOB) error (for random forest). The final decision tree and random forest models were retrained with these settings on the full training set and then evaluated on the test set.

Model	Impurity Function	max_depth	max_features	Test Loss
Decision Tree	`gini`	35	–	0.00082
Random Forest	`gini`	35	15	0.00000

Table 1: Optimal hyperparameters and test-set performance for the decision tree and random forest models.

4.1 Decision Tree Performance

Figures 1a and 1b show that the *training loss* decreases rapidly as `max_depth` increases or `min_impurity_decrease` decreases. As expected, once the trees exceed a certain complexity (e.g. `max_depth` ≥ 30), they fit the training data perfectly.

The more critical indicator is the *cross-validation* (CV) loss (see Figures 1c and 1d). For shallow trees (e.g., `max_depth` < 25), both the training and CV losses are high, indicating underfitting. In contrast, as the maximum depth reaches the interval between 25 and 35, both losses converge to values close to zero (on the order of 10^{-3}), confirming that the model is sufficiently expressive to capture the underlying structure of the data without overfitting.

A detailed comparison of the impurity functions (Gini, scaled entropy, and square root impurity) reveals a consistent trend: while all functions eventually reach near-zero training and CV loss with increasing depth, the Gini and scaled entropy measures exhibit a smoother reduction in error compared to the square root function.

Bias-Variance Perspective. For low values of `max_depth`, the model suffers from high bias, as evidenced by both high training and CV losses. Increasing the depth reduces the bias, while the variance, typically a concern with complex models, remains negligible in this case. The close alignment of training and CV losses (as shown in Figures 1c and 1e) indicates that the trees are not overfitting. The inherent separability of the data allows the model to achieve excellent performance without the need for additional pruning.

Final Decision Tree. Based on cross-validation, the optimal decision tree uses the `gini` impurity measure with `max_depth` set to 35. This model achieves a training loss of 0.0000 and a CV loss of approximately 0.00090. When evaluated on the 12,214-sample test set, the decision tree yields a misclassification rate of 0.00082 (10 misclassifications), as shown in Table 1.

4.2 Random Forest Performance

The random forest was constructed by aggregating multiple decision trees trained on bootstrap samples. The hyperparameter `max_features`, which determines the number of features considered at each split, was optimized by minimizing the OOB error (see Figure 1f). Even though the individual trees are powerful and the data is easily separable, using a smaller subset of features (in this case, 15) per split is sufficient to capture the relevant patterns, thus maintaining a good bias-variance balance.

With `max_features` set to 15, the OOB error becomes close to zero. Retraining the random forest on the full training set with this configuration produces a model that achieves perfect classification on the test set (0 misclassifications or 0.00000 test loss).

4.3 Interpretation and Overfitting Considerations

Despite the complexity of the learned trees, the CV results show minimal evidence of overfitting. In typical scenarios, unpruned trees could eventually overfit the training data, leading to increased variance in the validation data. However, the clear separability of the mushroom dataset allows each leaf to effectively discriminate between classes, thereby suppressing variance. The hyperparameter tuning process itself - adjusting `max_depth`, `min_impurity_decrease`, and `max_features` - acts as a natural regularizer, ensuring that the models remain well tuned.

In essence, both the decision tree and random forest models exhibit low bias and controlled variance, with the random forest achieving perfect classification. As a result, additional post-training pruning is unnecessary, as the models already generalize exceptionally well on unseen data.

In summary, the final decision tree model (with `gini` impurity and `max_depth` $= 35$) achieves a near-perfect test loss of 0.00082, while the optimized random forest yields a perfect test performance. These results underscore that robust hyperparameter tuning via cross-validation and OOB error estimation effectively regularizes the models, even in a problem where classical overfitting concerns are largely mitigated by the intrinsic simplicity of the dataset.

5 Conclusion

This project investigated decision trees and random forests for binary classification on a one-hot encoded Mushroom dataset. The final decision tree, using the Gini impurity and a maximum depth of 35, achieved near perfect classification on the test set (0.00082 loss). A random forest with 15 features considered at each split further improved the result to a perfect 0.00000 test loss. These successes are due to both the inherent separability of the dataset and a robust hyperparameter tuning scheme using cross-validation (for single trees) and out-of-bag error estimation (for random forests).

From a theoretical perspective, the rapid convergence to near-zero cross-validation loss underscores how nonlinear partitioning criteria (e.g., Gini or scaled entropy) can effectively reduce bias while controlling variance in highly discriminative domains. In this dataset, a high degree of separability minimized overfitting concerns, making advanced pruning unnecessary. However, in more complex, less separable domains, regularization through constraints on maximum depth, minimum impurity reduction, or pruning would become more critical.

In summary, the project demonstrates that well-designed decision trees and their ensembles can combine interpretability with strong generalization-especially when hyperparameters are systematically tuned and data preprocessing is carefully managed. Future extensions could explore feature engineering techniques or alternative ensemble methods to further refine performance and computational efficiency.

References

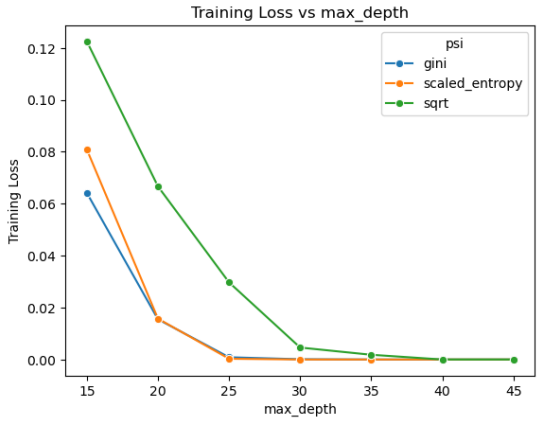
Cesa-Bianchi, N. (2024). *Statistical methods for machine learning: Course notes*. University of Milan.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning: With applications in r*. Springer.

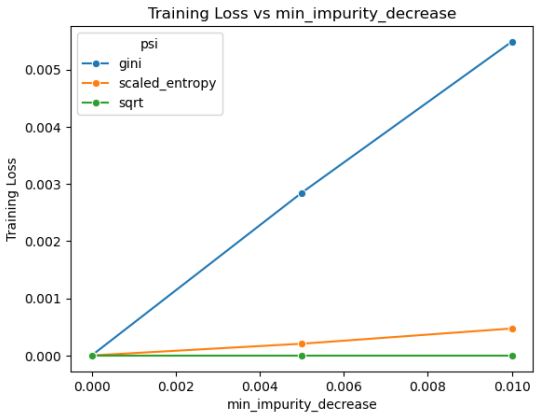
Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms*. Cambridge university press.

Wagner, D., Heider, D., & Hattab, G. (2021). Mushroom data creation, curation, and simulation to support classification tasks. *Scientific reports*, *11*(1), 8134.

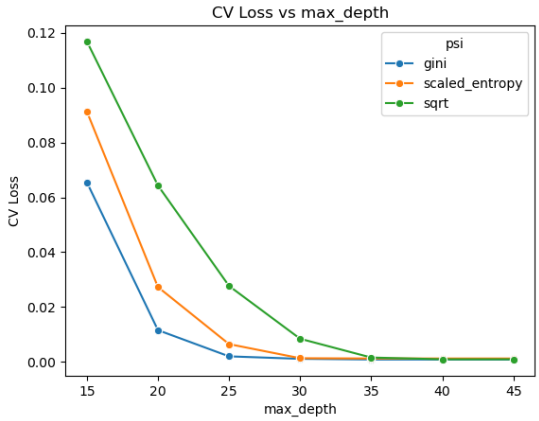
A Appendix: Graphs from the Experiments



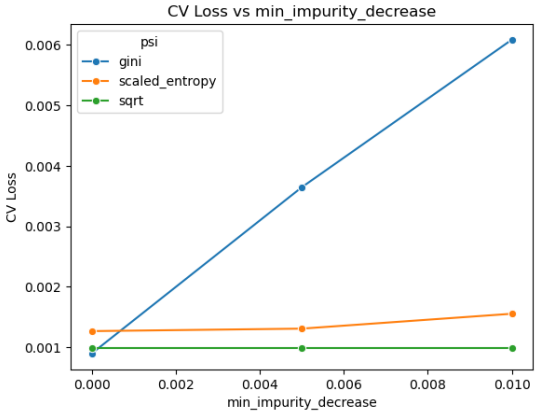
(a) Training Loss vs. `max_depth` for the baseline experiments with various impurity functions.



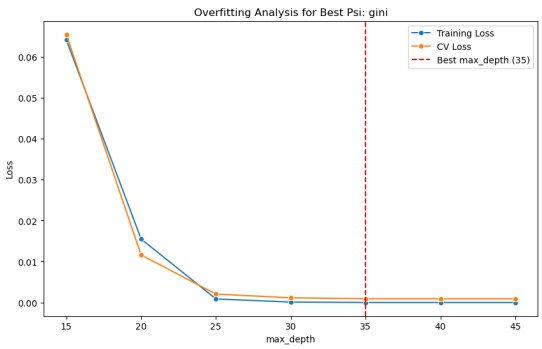
(b) Training Loss vs. `min_impurity_decrease` for the baseline experiments across different psi functions.



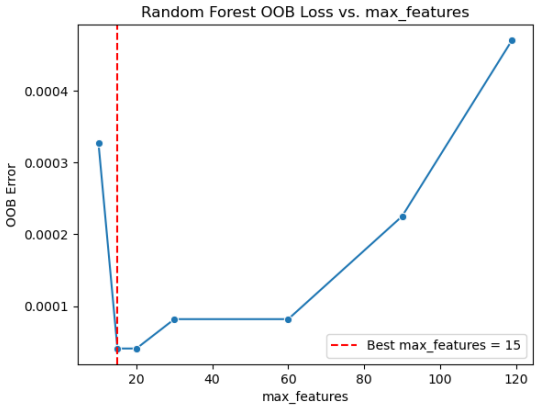
(c) Cross-validation (CV) Loss vs. `max_depth` for the Decision Tree experiments. The different lines represent the various psi functions evaluated.



(d) Cross-validation (CV) Loss vs. `min_impurity_decrease` for the Decision Tree experiments.



(e) Combined plot for the best psi function showing both Training Loss and CV Loss versus `max_depth`. The red dashed line indicates the optimal `max_depth` selected from the CV experiments.



(f) OOB Loss vs. `max_features` for the Random Forest experiments. The optimal `max_features` is highlighted with a red dashed vertical line.

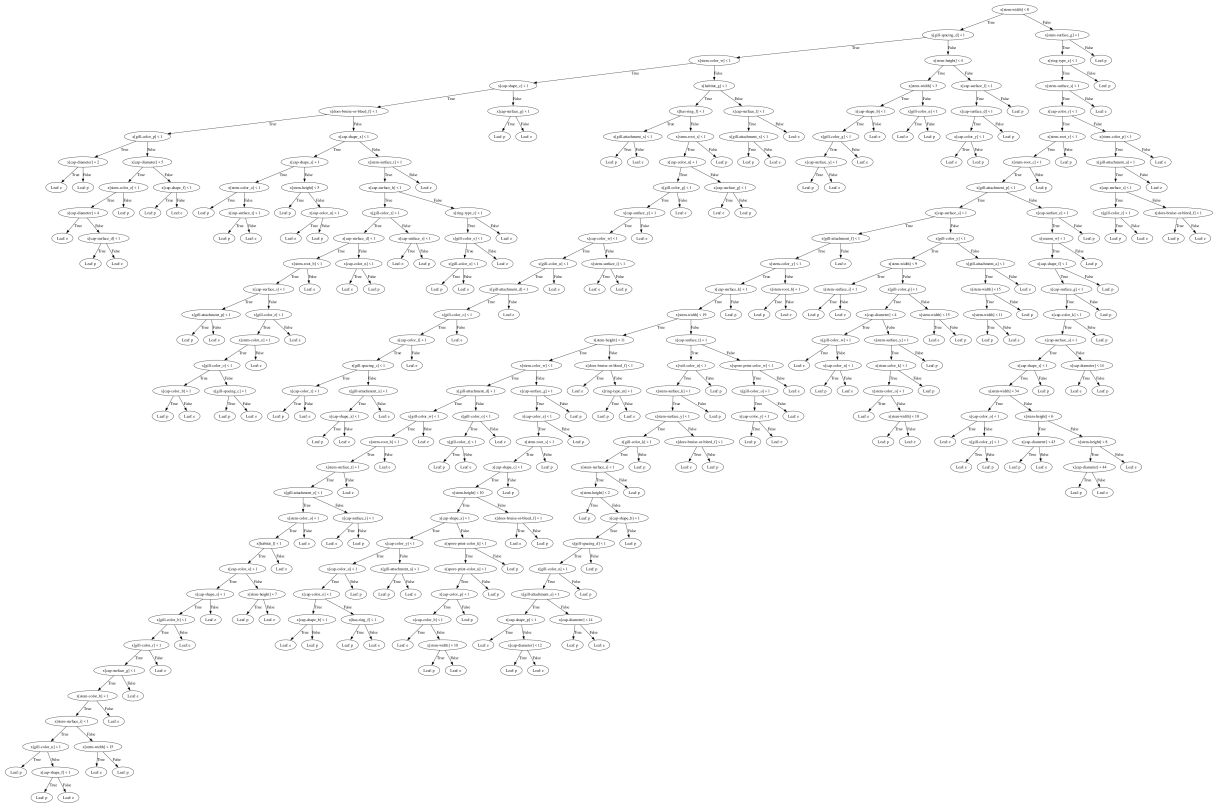


Figure 2: Visualization of the final decision tree learned with the best hyperparameters (as determined via cross-validation). This figure was produced by calling the `visualize` method on the fitted `TreePredictor`.

B Appendix: Detailed Experimental Results

Psi	min_impurity_decrease	Training Loss
gini	0.0	0.00000
gini	0.005	0.00285
gini	0.01	0.00549
scaled_entropy	0.0	0.00000
scaled_entropy	0.005	0.00020
scaled_entropy	0.01	0.00047
sqrt	0.0	0.00000
sqrt	0.005	0.00000
sqrt	0.01	0.00000

Table 2: Baseline training losses for decision trees with varying `min_impurity_decrease` values and different impurity functions.

Psi	max_depth	Training Loss
gini	15	0.06417
gini	20	0.01549
gini	25	0.00086
gini	30	0.00010
gini	35	0.00000
gini	40	0.00000
gini	45	0.00000
scaled_entropy	15	0.08087
scaled_entropy	20	0.01574
scaled_entropy	25	0.00029
scaled_entropy	30	0.00000
scaled_entropy	35	0.00000
scaled_entropy	40	0.00000
scaled_entropy	45	0.00000
sqrt	15	0.12253
sqrt	20	0.06667
sqrt	25	0.02970
sqrt	30	0.00461
sqrt	35	0.00182
sqrt	40	0.00000
sqrt	45	0.00000

Table 3: Baseline training losses for decision trees with varying `max_depth` values and different impurity functions.

Psi	min_impurity_decrease	CV Loss
gini	0.0	0.00090
gini	0.005	0.00364
gini	0.01	0.00608
scaled_entropy	0.0	0.00127
scaled_entropy	0.005	0.00131
scaled_entropy	0.01	0.00156
sqrt	0.0	0.00098
sqrt	0.005	0.00098
sqrt	0.01	0.00098

Table 4: Cross-validation losses for decision trees with varying `min_impurity_decrease` values and different impurity functions.

Psi	max_depth	CV Loss
gini	15	0.06550
gini	20	0.01161
gini	25	0.00205
gini	30	0.00115
gini	35	0.00090
gini	40	0.00090
gini	45	0.00090
scaled_entropy	15	0.09129
scaled_entropy	20	0.02731
scaled_entropy	25	0.00651
scaled_entropy	30	0.00137
scaled_entropy	35	0.00127
scaled_entropy	40	0.00127
scaled_entropy	45	0.00127
sqrt	15	0.11684
sqrt	20	0.06431
sqrt	25	0.02769
sqrt	30	0.00845
sqrt	35	0.00162
sqrt	40	0.00098
sqrt	45	0.00098

Table 5: Cross-validation losses for decision trees with varying `max_depth` values and different impurity functions.

max_features	OOB Loss
10	0.0003275
15	0.00004094
20	0.00004094
30	0.00008187
60	0.00008187
90	0.00022516
119	0.00047078

Table 6: OOB losses for the random forest model with different `max_features` values.

C Appendix: Code

All Python code used for data preprocessing, model implementation, and experimental analysis is publicly available on GitHub. The complete repository can be accessed at:

Tree predictors for binary classification