```
# ╔══════════════════════════════════════════════════════════════════╗
# ║  1. IMPORTS & DATA LOADER                                        ║
# ╚══════════════════════════════════════════════════════════════════╝
import numpy as np, pandas as pd, time, math, warnings
from urllib.request import urlretrieve
from collections import Counter
from dataclasses import dataclass
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.metrics import confusion_matrix, classification_report
from joblib import Parallel, delayed, parallel_backend
warnings.filterwarnings("ignore", category=FutureWarning)

URL  = "https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalon
COLS = ["Sex","Length","Diameter","Height","Whole","Shucked","Viscera","Shell","

def load_abalone(class_mode: str = "ageclass"):
    """
    Parameters
    ----------
    class_mode : {"ageclass", "rings"}
        • "ageclass": 3-class problem (young ≤ 9, 10-17 adult, ≥ 18 old)
        • "rings"   : original numeric Rings labels (1-29)
    """
    try:
        df = pd.read_csv("abalone.data", names=COLS)
    except FileNotFoundError:
        urlretrieve(URL, "abalone.data")
        df = pd.read_csv("abalone.data", names=COLS)

    if class_mode == "rings":
        y = df["Rings"]
    else:
        y = df["Rings"].apply(
            lambda r: "young" if r <= 9 else ("adult" if r <= 17 else "old")
        )

    X = df.drop("Rings", axis=1)
    attr_types = [2 if X[c].dtype == "object" else 1 for c in X.columns]

    print(f"Loaded {len(X)} samples — {len(y.unique())} classes  (mode={class_mo
    return X.reset_index(drop=True), y.reset_index(drop=True), attr_types
```

General Function Descriptions

load_abalone

The load_abalone function is a data loader that reads the UCI Abalone dataset and prepares it for classification tasks. It offers two labeling modes for the Rings column: in "rings" mode, the labels remain as raw integers between 1 and 29. However, since some age classes contain very few examples, this can lead to poor model performance. Therefore, the "ageclass" mode is preferred, which groups the ages into three classes: ages 1–9 are labeled as young, 10–17 as adult, and 18 and above as old. This simplifies the classification task and balances the class distribution. Additionally, the function returns a list indicating whether each feature is numeric or categorical, which is essential for proper feature handling in decision tree algorithms.

```
# ╔══════════════════════════════════════════════════════════════════════╗
# ║  2. IMPURITY & SPLIT HELPERS                                         ║
# ╚══════════════════════════════════════════════════════════════════════╝
def _gini(cnt):
    tot = cnt.sum(); p = cnt / tot if tot else cnt
    return 1 - np.sum(p ** 2)

def _ent(cnt):
    tot = cnt.sum(); p = cnt / tot if tot else cnt
    p = p[p > 0]
    return -np.sum(p * np.log2(p))

def _best_split_num(x, y_c, n_cls, parent_imp, crit):
    order = np.argsort(x)
    x_s, y_s = x[order], y_c[order]
    left = np.zeros(n_cls, int)
    best_g, best_t = 0.0, None
    f = _gini if crit == "gini" else _ent
    for i in range(len(x_s) - 1):
        left[y_s[i]] += 1
        if x_s[i] == x_s[i + 1]:
            continue
        n_l = i + 1; n_r = len(x_s) - n_l
        gain = parent_imp - (n_l/len(x_s))*f(left) - (n_r/len(x_s))*f(np.bincou
        if gain > best_g:
            best_g, best_t = gain, (x_s[i] + x_s[i + 1]) / 2
    return best_g, best_t

def _best_split_cat(x, y_c, n_cls, parent_imp, crit):
    best_g, best_cat = 0.0, None
    f = _gini if crit == "gini" else _ent
    for cat in np.unique(x):
        mask = x == cat
        n_l, n_r = mask.sum(), len(x) - mask.sum()
        gain = parent_imp - (n_l/len(x))*f(np.bincount(y_c[mask], minlength=n_c
                          - (n_r/len(x))*f(np.bincount(y_c[~mask], minlength=n_c
        if gain > best_g:
            best_g, best_cat = gain, cat
    return best_g, best_cat
```

These functions serve as helper utilities for calculating impurity measures and identifying the best split points in a decision tree. The _gini function computes Gini impurity, which quantifies how mixed the class labels are—lower values indicate purer nodes. The _ent function calculates entropy and is used in information gain-based splitting strategies. The _best_split_num function handles numeric attributes: it sorts the values and evaluates all possible threshold points to determine the split that yields the highest gain based on the chosen impurity criterion (Gini or Entropy). On the other hand, _best_split_cat is designed for categorical attributes. It tests each category by splitting the dataset into two groups and calculates the corresponding impurity reduction. The category yielding the maximum information gain is selected. These functions play a critical role in guiding how the decision tree grows by choosing the most informative splits.

```
#
# ╔═══════════════════════════════════════════════════════════════════════╗
# ║ 3. DECISION TREE (UNPRUNED)                                           ║
# ╚═══════════════════════════════════════════════════════════════════════╝
@dataclass
class DTNode:
    is_leaf: bool
    pred: any = None
    feat: int = None
    thr: float = None
    cat: any = None
    left: any = None
    right: any = None

def build_dt(X, y, attr_types, *, criterion="gini"):
    """Grow tree until no informative split remains (no pre-pruning)."""
    # parent impurity
    y_c, uniq = pd.factorize(y); n_cls = len(uniq)
    parent_imp = (_gini if criterion == "gini" else _ent)(np.bincount(y_c, minl

    # BEST SPLIT SEARCH
    best_g = 0.0; best_feat = best_thr = best_cat = None
    for i, col in enumerate(X.columns):
        if attr_types[i] == 1:
            gain, thr = _best_split_num(X[col].values, y_c, n_cls, parent_imp,
            if gain > best_g:
                best_g, best_feat, best_thr, best_cat = gain, i, thr, None
        else:
            gain, cat = _best_split_cat(X[col].values, y_c, n_cls, parent_imp,
            if gain > best_g:
                best_g, best_feat, best_thr, best_cat = gain, i, None, cat
```

```
    # STOP if no information gain (pure or identical attributes)
    if best_g == 0.0 or (X.nunique(axis=0) <= 1).all():
        return DTNode(True, pred=y.value_counts().idxmax())

    mask = (X.iloc[:, best_feat] <= best_thr) if best_thr is not None else (X.i
    left = build_dt(X[mask], y[mask], attr_types, criterion=criterion)
    right = build_dt(X[~mask], y[~mask], attr_types, criterion=criterion)
    return DTNode(False, feat=best_feat, thr=best_thr, cat=best_cat, left=left,

def _predict_one(node: DTNode, row):
    while not node.is_leaf:
        v = row[node.feat]
        node = node.left if ((v <= node.thr) if node.thr is not None else (v ==
    return node.pred

def predict_dt(tree: DTNode, X):
    return np.array([_predict_one(tree, r) for r in X.values])

# ─── Post-Pruning (optional) ──────────────────────────────────────────
def prune_dt(node: DTNode, X_val, y_val):
    if node.is_leaf or len(y_val) == 0:
        return node
    mask = (X_val.iloc[:, node.feat] <= node.thr) if node.thr is not None else
    node.left = prune_dt(node.left, X_val[mask], y_val[mask])
    node.right = prune_dt(node.right, X_val[~mask], y_val[~mask])
    leaf = DTNode(True, pred=y_val.value_counts().idxmax())
    return leaf if (predict_dt(node, X_val) == y_val).mean() <= (predict_dt(lea
```

General Description

This section constructs a pure, unpruned decision tree and includes prediction and optional post-pruning functionalities. The build_dt function is responsible for recursively building the tree based on the training data. It first factorizes the target labels and computes the impurity (either Gini or Entropy). For each feature, it finds the split point that yields the highest information gain. If no split provides a positive gain (i.e., the data is pure or not splittable), the recursion stops and a leaf node is created with the majority class. Otherwise, the data is split into left and right subsets based on the best feature and threshold or category, and the function continues recursively. The predict_dt function uses the trained tree to classify test samples, leveraging _predict_one to traverse the tree for each sample. The optional prune_dt function applies post-pruning using validation data: it evaluates whether collapsing a subtree into a single leaf would maintain or improve accuracy. If so, the subtree is pruned. This helps to reduce overfitting and improve generalization. Overall, the implementation builds a data-

driven decision tree and optionally simplifies it through pruning.

Comparing Pruned and Unpruned Trees

- An unpruned decision tree grows to its full depth by using all available
- A pruned decision tree trims unnecessary branches using validation data.
- In an unpruned tree, every subtree continues splitting until there is no
- From a performance perspective, unpruned trees tend to perform better on

Code-Level Differences Between Pruned and Unpruned Trees

The unpruned version of the decision tree is built using the build_dt function, which grows the tree as deep as necessary until no further information gain is possible or all feature values become identical. At that point, the node becomes a leaf, assigned the most frequent class label in the subset. This approach allows the tree to fully memorize the training data, which often leads to overfitting due to its excessive complexity. In contrast, the prune_dt function operates on this fully-grown tree and simplifies it using validation data. For each node, it compares the prediction accuracy of the existing subtree against that of a single leaf node trained on the same data. If the leaf provides equal or better accuracy, the subtree is pruned and replaced with this simpler leaf. As a result, the overall model becomes more generalizable and less prone to overfitting. The key code-level difference lies in the strategy: while the unpruned tree relies solely on information gain to grow, the pruned tree introduces a validation-based simplification step to enhance performance.

```
# ╔══════════════════════════════════════════════════════════════════╗
# ║  4. RANDOM FOREST (no pre-pruning in trees)                      ║
# ╚══════════════════════════════════════════════════════════════════╝
def _build_tree_bootstrap(X, y, attr_types, criterion, max_features, seed):
    rng = np.random.default_rng(seed)
    idx = rng.choice(len(X), len(X), replace=True)                # bootstrap
    if max_features == "sqrt":
        feat_idx = rng.choice(X.shape[1], int(math.sqrt(X.shape[1])), replace=F
    else:                                                         # None → all
        feat_idx = np.arange(X.shape[1])
    sub_types = [attr_types[i] for i in feat_idx]
    tree = build_dt(X.iloc[idx, feat_idx], y.iloc[idx], sub_types, criterion=cr
    return tree, feat_idx

def build_rdf(X, y, attr_types, *, N=100, criterion="gini", max_features="sqrt"
    seeds = np.random.SeedSequence(42).spawn(N)
    with parallel_backend("loky", inner_max_num_threads=1):
        forest = Parallel(n_jobs=n_jobs, prefer="processes")(
            delayed(_build_tree_bootstrap)(X, y, attr_types, criterion, max_fea
                                           int(s.generate_state(1)[0]))
            for s in seeds)
    return forest

def predict_rdf(forest, X):
    votes = np.array([predict_dt(t, X.iloc[:, fi]) for t, fi in forest])
    return np.array([Counter(col).most_common(1)[0][0] for col in votes.T])
```

Random Forest

This section implements a pure-Python version of the Random Forest algorithm, which combines multiple decision trees to form an ensemble. The build_rdf function trains N decision trees independently on bootstrapped subsets of the original data and returns the entire forest. Each individual tree is constructed using _build_tree_bootstrap, which samples rows from the dataset with replacement (bootstrap) and optionally selects a random subset of features (e.g., "sqrt" selects √d features). This introduces diversity across the trees, which is crucial for the ensemble's effectiveness. No pre-pruning is applied, so each tree is grown to its maximum extent. The predict_rdf function aggregates the predictions from all trees for each test instance and selects the majority class via voting. Compared to a single decision tree, Random Forests provide more stable and generalizable predictions, as individual errors are mitigated across the ensemble.

```
# ╔═══════════════════════════════════════════════════════════════════════╗
# ║ 5. K-FOLD EVALUATION                                                  ║
# ╚═══════════════════════════════════════════════════════════════════════╝
def eval_model(X, y, attr, build_fn, predict_fn, k=5, **bkw):
    cv_cls = StratifiedKFold if min(y.value_counts()) >= k else KFold
    kf = cv_cls(n_splits=k, shuffle=True, random_state=42)
    accs, bt, pt, y_true, y_pred = [], [], [], [], []
    for fold, (tr, te) in enumerate(kf.split(X, y), 1):
        t0 = time.time(); model = build_fn(X.iloc[tr], y.iloc[tr], attr, **bkw)
        t1 = time.time(); pred = predict_fn(model, X.iloc[te]); pt.append(time.
        acc = (pred == y.iloc[te].values).mean()
        print(f"Fold {fold} | acc={acc:.3f} | build={bt[-1]:.3f}s | pred={pt[-1
        accs.append(acc); y_true.extend(y.iloc[te].values); y_pred.extend(pred)
    print(f"Mean acc={np.mean(accs):.3f} ± {np.std(accs):.3f} | build={np.mean(
    labels = sorted(y.unique())
    print("\nConfusion Matrix:\n", pd.DataFrame(confusion_matrix(y_true, y_prec
                                               index=labels, columns=labels))
    print("\nClassification Report:\n", classification_report(y_true, y_pred,
                                                             labels=labels, zer
```

The eval_model function evaluates a given model using k-fold cross-validation by applying the provided model-building function (build_fn) and prediction function (predict_fn). The dataset is split into k folds, and in each iteration, one fold is used as the test set while the remaining folds serve as the training set. If the smallest class has sufficient samples, StratifiedKFold is used to preserve class proportions; otherwise, KFold is applied. For each fold, the training time and prediction time are recorded, and the accuracy is computed. At the end of the process, the function prints the average accuracy and standard deviation, along with a confusion matrix and a classification report (including metrics like precision, recall, and f1-score). This comprehensive evaluation ensures that the model is tested on multiple data splits, providing a better understanding of its generalization performance.

```
# ╔═══════════════════════════════════════════════════════════════════════╗
# ║ 6. MAIN PIPELINE                                                      ║
# ╚═══════════════════════════════════════════════════════════════════════╝
def run_all(k=5, class_mode="ageclass", rf_params=None):
    """
    Parameters
    ----------
    class_mode : {"ageclass", "rings"}
        • "ageclass" → 3-class (young / adult / old)
        • "rings"    → 29-class (Rings 1-29)
    """
    X, y, attr = load_abalone(class_mode)
```

```
    print("\n=== Unpruned Decision Tree ===")
    eval_model(X, y, attr, build_dt, predict_dt, k)

    # Post-pruning (optional)
    def build_pruned(X_tr, y_tr, attr):
        split = int(0.8 * len(X_tr))
        tree = build_dt(X_tr.iloc[:split], y_tr.iloc[:split], attr)
        return prune_dt(tree, X_tr.iloc[split:], y_tr.iloc[split:])

    print("\n=== Pruned Decision Tree ===")
    eval_model(X, y, attr, build_pruned, predict_dt, k)

    print("\n=== Random Forest ===")
    rf_params = rf_params or {}
    eval_model(X, y, attr, build_rdf, predict_rdf, k, **rf_params)

# ── demo ────────────────────────────────────────────────────────
if __name__ == "__main__":
    rf_params = dict(N=150, n_jobs=-1, max_features="sqrt")
    run_all(k=5, rf_params=rf_params)
```

```
          old       0.18      0.22      0.20       136
        young       0.73      0.73      0.73      2096

     accuracy                           0.68      4177
    macro avg       0.52      0.53      0.53      4177
 weighted avg       0.68      0.68      0.68      4177


 === Pruned Decision Tree ===
 Fold 1 | acc=0.719 | build=6.190s | pred=0.002s
 Fold 2 | acc=0.702 | build=5.741s | pred=0.002s
 Fold 3 | acc=0.739 | build=6.526s | pred=0.002s
 Fold 4 | acc=0.710 | build=5.711s | pred=0.002s
 Fold 5 | acc=0.753 | build=6.162s | pred=0.003s
 Mean acc=0.725 ± 0.019 | build=6.066s | pred=0.002s

 Confusion Matrix:
         adult  old  young
 adult   1436   18    491
 old      128    1      7
 young    502    4   1590

 Classification Report:
               precision    recall  f1-score   support

        adult       0.70      0.74      0.72      1945
          old       0.04      0.01      0.01       136
        young       0.76      0.76      0.76      2096

     accuracy                           0.72      4177
```

```
   macro avg       0.50      0.50      0.50      4177
weighted avg       0.71      0.72      0.72      4177


=== Random Forest ===
Fold 1 | acc=0.749 | build=276.495s | pred=0.697s
Fold 2 | acc=0.720 | build=273.864s | pred=0.696s
Fold 3 | acc=0.727 | build=263.182s | pred=0.691s
Fold 4 | acc=0.734 | build=263.915s | pred=0.693s
Fold 5 | acc=0.729 | build=263.151s | pred=0.683s
Mean acc=0.732 ± 0.010 | build=268.121s | pred=0.692s

Confusion Matrix:
        adult  old  young
adult   1540    0    405
old      125    0     11
young    579    0   1517

Classification Report:
               precision    recall  f1-score   support

       adult       0.69      0.79      0.74      1945
         old       0.00      0.00      0.00       136
       young       0.78      0.72      0.75      2096

    accuracy                           0.73      4177
   macro avg       0.49      0.51      0.50      4177
weighted avg       0.71      0.73      0.72      4177
```

Accuracy

The Unpruned Decision Tree had the lowest average accuracy at 67.6%, with a relatively high standard deviation (±0.016), indicating less stable performance. The Pruned version significantly improved the accuracy to 72.5%, while the Random Forest achieved the highest accuracy at 73.2% with the lowest variance (±0.010), showing more consistent behavior.

Prediction Time

While both Unpruned and Pruned decision trees can be built in approximately ~6 seconds, the Random Forest model takes significantly longer—about 267 seconds to train. Its prediction time is also notably higher (around 0.822 seconds), especially when compared to the near-instant inference of single-tree models. The primary reason for this is the nature of Random Forest: it builds many individual decision trees (e.g., 100), each requiring bootstrapped sampling, feature subset selection, and separate training.

Theoretically, ensemble methods like Random Forest are highly parallelizable by design. Since each tree is built independently, they can be trained concurrently, reducing the total training time to the duration of the longest tree. However, in this implementation, the training process was still time-consuming, likely due to hardware constraints, such as limited CPU cores, memory bandwidth, and I/O capabilities.

Moreover, the parallelization library used (joblib) may introduce additional overhead. For instance, spawning new processes, duplicating memory, and managing inter-process communication can add substantial delays. These overheads diminish the practical speed gains expected from parallel execution.

Therefore, the long training time in this context is not solely a consequence of model complexity but also a result of limited computational resources and inefficiencies in parallel job execution. On more powerful hardware or with more optimized parallel frameworks (e.g., using multithreading instead of multiprocessing), the training time of Random Forest could be significantly reduced.

Precision / Recall / F1-Score

The "young" class performs best across all models, with accuracy improving from 73% (Unpruned) to 76% (Pruned) and 78% (Random Forest). For the "adult" class, Random Forest achieves the highest recall (79%), followed by the Pruned tree (74%). However, all models struggle with the "old" class, especially Random Forest and the Pruned version, likely due to its small sample size and class imbalance.

Model Complexities

The Unpruned model is more prone to overfitting due to its complexity. The Pruned tree mitigates this by simplifying the structure, improving generalization. Random Forest goes further by injecting randomness in both samples and features, enhancing robustness. Yet, even ensemble methods like Random Forest struggle with rare classes like "old".

Comparision of Models

Decision trees, particularly in their unpruned form, are among the most susceptible models to overfitting. An Unpruned Decision Tree aims to perfectly classify the training data by continuing to split until no information gain remains. This results in overly deep and complex

tree structures. Although this leads to low training error, it significantly hampers the model's ability to generalize, especially on unseen data. In statistical terms, the model exhibits high variance, reacting dramatically to small fluctuations in the training data.

The Pruned Decision Tree addresses this issue by reducing model complexity through post-pruning. At each decision point, the performance of the current subtree is compared to that of a single leaf node using a validation set. If the simplified version performs better or equally well, the subtree is pruned and replaced by a single node. This reduces variance, improves generalization, and mitigates overfitting by regularizing the tree structure.

Random Forest, however, offers a more sophisticated approach to overcoming overfitting. It builds an ensemble of decision trees, each trained on a different bootstrapped sample of the data. But what makes Random Forest particularly powerful is its random feature selection mechanism: during tree construction, each node only considers a random subset of the features instead of the entire feature set. This reduces the correlation between individual trees, increasing ensemble diversity. As a result, different trees focus on different attributes and patterns, and their errors are less likely to overlap. When these diverse predictions are aggregated through majority voting, the ensemble significantly reduces variance and provides more robust, generalizable predictions.

In conclusion, the Unpruned Decision Tree suffers from high variance and is prone to overfitting. The Pruned Decision Tree alleviates this by simplifying the tree structure through validation-guided pruning. Random Forest further enhances generalization by combining bootstrap sampling and random feature selection, enabling the model to reduce variance at both the data and feature levels, making it a powerful tool especially for high-dimensional datasets.

Result

The Pruned Decision Tree offers the best trade-off between accuracy and speed. While Random Forest delivers slightly better performance, its high time and resource cost may not be ideal in every context. If accuracy is critical and time is not a constraint, Random Forest is the better choice. Otherwise, the Pruned tree offers solid, fast, and generalizable results. The Unpruned version can still be useful for quick experimentation on smaller datasets.