```python
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
# Load the abalone dataset from the newly uploaded file
abalone_data = 'abalone.data'
columns = ["Sex", "Length", "Diameter", "Height", "Whole weight",
"Shucked weight", "Viscera weight", "Shell weight", "Rings"]
abalone_data_new = pd.read_csv(abalone_data, header=None,
names=columns)


# Convert the 'Sex' column to numerical values
sex_encoder = LabelEncoder()
abalone_data_new['Sex'] =
sex_encoder.fit_transform(abalone_data_new['Sex'])

# Show the first few rows of the newly loaded and processed dataset to
verify changes
abalone_data_new.head()

from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
X = abalone_data_new.drop('Rings', axis=1).values
y = abalone_data_new['Rings'].values
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)



# Redefine the decision tree functions with necessary imports
def calculate_entropy(y):
    if len(y) == 0:
        return 0
    class_labels, counts = np.unique(y, return_counts=True)
    probabilities = counts / counts.sum()
    entropy = -np.sum(probabilities * np.log2(probabilities + 1e-9))
# avoid log2(0)
    return entropy

def calculate_information_gain(X, y, feature_index):
    total_entropy = calculate_entropy(y)
    values, counts = np.unique(X[:, feature_index],
return_counts=True)
    weighted_entropy = sum((count / y.size) * calculate_entropy(y[X[:,
feature_index] == value]) for value, count in zip(values, counts))
    return total_entropy - weighted_entropy

def build_tree(X, y, features, depth=0, max_depth=5):
```

```
    if len(np.unique(y)) == 1 or len(features) == 0 or depth ==
max_depth:
        return np.bincount(y).argmax() if y.size > 0 else None
    gains = [calculate_information_gain(X, y, feature) for feature in
features]
    best_feature = features[np.argmax(gains)]
    tree = {best_feature: {}}
    features = [f for f in features if f != best_feature]
    for value in np.unique(X[:, best_feature]):
        sub_X = X[X[:, best_feature] == value]
        sub_y = y[X[:, best_feature] == value]
        subtree = build_tree(sub_X, sub_y, features, depth + 1,
max_depth)
        tree[best_feature][value] = subtree
    return tree

def predict_dt(tree, x_test):
    for key, value in tree.items():
        feature_val = x_test[key]
        if feature_val in value:
            result = value[feature_val]
            if isinstance(result, dict):
                return predict_dt(result, x_test)
            else:
                return result
    return None

def print_tree(tree, depth=0):
    """ Recursively print the decision tree in a formatted manner """
    if not isinstance(tree, dict):
        print("\t" * depth + "-> Prediction: " + str(tree))
        return
    for feature_index, branches in tree.items():
        for value, subtree in branches.items():
            print("\t" * depth + "Feature {}:
{}".format(feature_index, value))
            print_tree(subtree, depth + 1)




# Rebuild the decision tree and evaluate it
features_indices = list(range(X.shape[1]))
tree = build_tree(X_train, y_train, features_indices, max_depth=5)
test_predictions = [predict_dt(tree, dict(enumerate(x))) for x in
X_test]
accuracy = np.mean(np.array(test_predictions) == y_test)
#print_tree(tree,0)
print("Native tree:",accuracy)
```

```
Native tree: 0.08492822966507177
```

Buiid Tree And Predict Tree

The above code aims to build and evaluate a decision tree model for predicting the age of abalones using the Abalone dataset. Initially, the code loads the dataset and converts the 'Sex' column into numerical values, making it processable for machine learning algorithms. After splitting the data into training and testing sets, the user defines their own decision tree functions to calculate entropy and information gain, and builds the tree based on these values.

As a result, the decision tree model achieves a very low accuracy rate of 8.49% on the test dataset. This indicates that the model fails to generalize well and is unsuccessful in making accurate predictions. The low accuracy could suggest that the model might be overfitting, or that the dataset may not be suitable for modeling with a decision tree algorithm. To address this issue, one could try different parameter settings, add more data preprocessing steps to better prepare the data for modeling, or perhaps use a different algorithm.

```
def calculate_accuracy(tree, X, y):
    predictions = [predict_dt(tree, dict(enumerate(row))) for row in 
X]
    return np.mean(np.array(predictions) == y)

def prune_tree(tree, X_valid, y_valid):
    """ Prune the decision tree based on reduced error pruning 
strategy using the validation set. """
    if not isinstance(tree, dict):
        return  # If the tree is not a dictionary, it's a leaf node, 
return immediately

    # Try to prune each subtree recursively
    for feature in list(tree.keys()):
        subtrees = tree[feature]
        for value, subtree in list(subtrees.items()):
            sub_X_valid = X_valid[X_valid[:, feature] == value]
            sub_y_valid = y_valid[X_valid[:, feature] == value]

            if isinstance(subtree, dict) and sub_y_valid.size > 0:  # 
Ensure there are samples to work with
                prune_tree(subtree, sub_X_valid, sub_y_valid)

                # Check if replacing the subtree with a leaf improves 
the accuracy
                leaf_value = np.bincount(sub_y_valid).argmax() if 
sub_y_valid.size > 0 else np.bincount(y_valid).argmax()
                subtree_as_leaf = leaf_value
```

```
                tree_with_leaf = {feature: {value: subtree_as_leaf}}

                # Calculate accuracy with the subtree replaced by the
leaf
                current_accuracy = calculate_accuracy(tree, X_valid,
y_valid)
                new_accuracy = calculate_accuracy(tree_with_leaf,
X_valid, y_valid)

                # If accuracy is improved or stays the same, replace
the subtree with the leaf
                if new_accuracy >= current_accuracy:
                    tree[feature][value] = subtree_as_leaf

prune_tree(tree, X_train, y_train)

pruned_accuracy = calculate_accuracy(tree, X_test, y_test)
print("Pruned tree:",pruned_accuracy)
```

```
Pruned tree: 0.08492822966507177
```

Pruned Tree

The above code attempts to improve the performance of a decision tree model by pruning the tree using a reduced error pruning strategy with a validation set. The code iteratively evaluates each node of the tree, and if replacing a particular subtree with a leaf node maintains or improves accuracy, it replaces that subtree with the leaf value. This process is intended to help prevent overfitting and enhance the model's generalization capability.

However, according to the output, the pruning process has not effectively increased the model's accuracy, remaining at 8.49%. This might indicate that the pruning strategy used may not be suitable for the current dataset and the structure of the problem, or that the model was already too simple for pruning to have a significant effect. To develop a more effective modeling strategy, it may be worth considering different machine learning techniques or improving the data preprocessing steps.

```
def build_rdf(X, y, attribute_types, N, options):
    trees = []
    for _ in range(N):
        # Bootstrap sample
        indices = np.random.choice(np.arange(len(X)), size=len(X),
replace=True)
        sample_X = X[indices]
        sample_y = y[indices]
        tree = build_tree(sample_X, sample_y, attribute_types,
options.get('max_depth', 10))
        trees.append(tree)
    return trees

def predict_single(tree, x):
```

```
    while isinstance(tree, dict):
        if x[tree['attribute']] <= tree['threshold']:
            tree = tree['left']
        else:
            tree = tree['right']
    return tree

def predict_rdf(rdf, X, options):
    predictions = []
    for x in X:
        votes = [predict_single(tree, x) for tree in rdf]
        predictions.append(max(set(votes), key=votes.count))
    return predictions

# Build the Random Decision Forest
rdf = build_rdf(X_train, y_train, features_indices, N=10,
options={'max_depth': 5})

# Predict and evaluate the model
predictions = predict_rdf(rdf, X_test, options={})
accuracy = np.mean(predictions == y_test)
print('RDF Accuracy:', accuracy)

RDF Accuracy: 0.16985645933014354
```

RDF Tree

The code snippet above performs a series of operations to construct a Random Decision Forest (RDF) model. This process involves drawing bootstrap samples from the training data to independently construct each tree, using parameters such as the indices of attributes and maximum depth. Each sample is then used to build a decision tree, which is added to a list to form the forest. During the prediction phase, predictions are made for each test instance across all trees in the forest, and the most frequent prediction is selected as the final result.

As a result, the model achieves an accuracy of 16.99% on the test dataset. This indicates that the model still performs poorly, but there is an improvement over the previous decision tree model. Although the Random Decision Forest method generally has better generalization capabilities compared to a single decision tree, the accuracy remains quite low in this case. To enhance performance, a forest with more trees could be utilized, the depth of the trees could be increased, or different feature selection and data preprocessing techniques could be explored.