Is My Classification Tree Best?

EE6227 Assignment 2

Xueyi Ke
Matric G2303895E
xke001@e.ntu.edu.sg

Kezhi Mao
Assoc. Prof.
ekzmao@ntu.edu.sg

***Abstract*—How do we know if our model is the best? This study delves into constructing a classification tree from the ground up, deliberately eschewing the convenience of pre-existing packages, to predict outcomes on an unlabeled test dataset. Confronted with the challenge of absent test labels, we leverage a validation set to refine our model's parameters. Our bespoke model undergoes training and validation on a self-preprocessed dataset, utilizing k-fold validation and innovative preprocessing techniques. Key among these are Stochastic Imputation for addressing missing values and Capping to mitigate the effects of outliers. Remarkably, without access to labeled test data, our model secures a validation accuracy of an impressive 93.33%, equating the performance of models built on standard packages. This endeavor not only illustrates our learning odyssey but also contributes to ongoing research. Code and detailed findings are accessible.**[1]

***Index Terms*—Classification Tree, Data Pre-processing, K-fold Validation**

Disclaimer

This paper incorporates GPT as a tool for enhancing its composition and visual presentations. It is emphasized that all core code was *independently* developed by the author.

I. Introduction

Decision trees are a fundamental component of machine learning, offering intuitive models that mimic human decision-making processes [1]. Classification trees, a subtype of decision trees, are particularly valuable for their ability to categorize instances into discrete classes, making them indispensable in various applications ranging from medical diagnoses to financial forecasting [2].

However, the efficacy of a classification tree heavily depends on the quality of the data it is trained on. Data preprocessing is a critical step in the machine learning pipeline, addressing issues such as missing values and outliers, which can significantly skew the model's performance. To address this, our approach diverges from the elementary imputation methods commonly discussed in classroom settings, opting instead for *Stochastic Imputation*. This method introduces a degree of randomness in the imputation process, thereby better capturing the variability inherent in the data [3]. Outliers, on the other hand, potentially distorting model predictions. Instead of excluding these data points, we apply *Capping* (or Winsorizing), effectively diminishing their impact while preserving the dataset's integrity [4].

The refinement of our model's parameters is achieved through the use of a validation set, with k-fold validation [5] serving as a cornerstone of our validation strategy. This method not only facilitates an accurate assessment of the model's performance but also guards against overfitting, ensuring the model's applicability to unseen data.

Despite the absence of labeled test data, our bespoke model demonstrates predictive capabilities that align closely with those developed using existing machine learning packages, a testament to the effectiveness of our preprocessing and validation techniques.

II. Dataset Preprocessing

In this section, we describe the dataset and detail our approach to data preprocessing, addressing missing values and outliers.

A. Dataset Information

The training dataset consists of 120 samples, each featuring four attributes, categorized into three classes labeled as 1, 2, and 3, respectively. Figure 1 shows the class distribution across the dataset, indicating a balanced class representation.

The test dataset comprises 30 samples, each with four features with no training labels.

It is important to note the presence of missing values and outliers within the original training dataset.

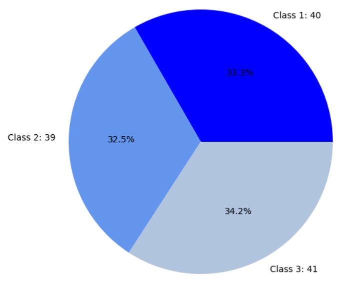

Fig. 1: Class distribution in the training set, with all three classes almost equally distributed.

[1]https://github.com/Kexueyi/EE6227-ML

B. *Preprocessing*

Our preprocessing strategy involves **Stochastic Imputation** for missing values and **Capping** for outliers in the dataset. Note that we are not using any off-the-shelf packages.

1) Missing Values Analysis: Using **Stochastic Imputation**, we address missing dataset values by substituting them with random samples from identical class distributions, maintaining class-specific data integrity.

The imputation is achieved by:

$$F_i^{s'} = \text{rand\_samp}(\{F_i^s \mid s \in D,\, \text{cls}(s) = C_k,\, F_i^s \neq \varnothing\}),$$

where $F_i^{s'}$ is the imputed value for a missing feature F_i in sample s' of class C_k, chosen randomly from non-null F_i values within C_k.

This method ensures data consistency across classes by replacing all missing values with contextually appropriate entries, as shown in Table I.

TABLE I: Missing Values and Imputation

RowIndex	Feature	Label	Imputations
37	Feature A	2	7.7
59	Feature A	1	4.7
81	Feature D	3	1.5

The implementaion is shown in below:

```
def Sto_Imp_fillna(df, seed=0):
    np.random.seed(seed) # set seed for
    ↪  reproducibility
    df = df.copy()
    class_label = df.columns[-1]
    # loop features
    for column in df.columns[:-1]:
        column_index = df.columns.get_loc(column)
        for label in df[class_label].unique():
            # find samples with NaN in this class
            ↪  label
            rows_with_nan = df[(df[class_label] ==
            ↪  label) &
            ↪  (df[column].isnull())].index
            for row in rows_with_nan:
                # find valid samples with same
                ↪  class label as the NaN samples
                valid_samples =
                ↪  df[(df[class_label] == label)
                ↪  & (~df[column].isnull())]
                if not valid_samples.empty:
                    # random replace NaN with
                    ↪  random value(fixed
                    ↪  feature) from valid samples
                    replacement_value =
                    ↪  valid_samples[column].samp⌋
                    ↪  le(n=1).values[0]
                    df.at[row, column] =
                    ↪  replacement_value
                    print(f"Replaced NaN at row
                    ↪  {row}, column
                    ↪  {column_index} with value
                    ↪  {replacement_value} from
                    ↪  label {label}")
                else:
                    print(f"No valid samples found
                    ↪  for {label} in {column}")
                    pass
    return df
```

2) Outliers Analysis: After Stochastic Imputation, we perform outlier analysis using **capping** to address potential increases in outlier frequency. This method involves:

1) Calculating quartiles (Q1 and Q3) and the Interquartile Range (IQR).
2) Defining normal range boundaries by extending 1.5 times the IQR from Q1 and Q3.
3) Adjusting outliers to these boundaries, ensuring values stay within a normal range.

This approach helps maintain the dataset's distribution post-imputation, preventing skewed analysis. Outliers, identified using the IQR, are adjusted via capping, as shown in table (II), with corrections ensuring values adhere to established limits.

TABLE II: Outlier Values and Corrections

RowIndex	Feature	OutlierValue	Correction
94	Feature A	10.6	8.35 (upper)
51	Feature B	4.1	4.05 (upper)
96	Feature B	4.4	4.05 (upper)
89	Feature C	16	10.35 (upper)
115	Feature D	5.3	4.05 (upper)

```
def cap_outliers(df, outliers):
    for index, row in outliers.iterrows():
        #feature as column index
        column = df.columns[int(row['Feature'])] #
        ↪  as integer
        # calculate IQR
        Q1 = df[column].quantile(0.25)
        Q3 = df[column].quantile(0.75)
        IQR = Q3 - Q1
        # calculate lower and upper bounds
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        value = df.at[row['RowIndex'], column]

        # cap outliers
        if row['OutlierValue'] < lower_bound:
            df.at[row['RowIndex'], column] =
            ↪  lower_bound
            print(f"Capped outlier in row
            ↪  {row['RowIndex']}, feature
            ↪  {row['Feature']}, value {value},
            ↪  with lower bound {lower_bound}")
        elif row['OutlierValue'] > upper_bound:
            df.at[row['RowIndex'], column] =
            ↪  upper_bound
            print(f"Capped outlier in row
            ↪  {row['RowIndex']}, feature
            ↪  {row['Feature']}, value {value},
            ↪  with upper bound {upper_bound}")
    return df
```

C. Preprocessing Selection Justification

1) Why Stochastic Imputation:

- The significance of missing values (3 instances) relative to our dataset size, making deletion impractical.
- The inadequacy of mean, mode, or median imputation, as they:
  - Oversimplify by ignoring in-class variations.
  - Fail to maintain the dataset's statistical integrity.

2) Why Capping for Outliers:

- Deleting 5 outliers would disproportionately affect our limited dataset.
- Alternatives were considered less effective due to:
  - The potential loss of valuable data.
  - The distortion of the dataset's distribution.

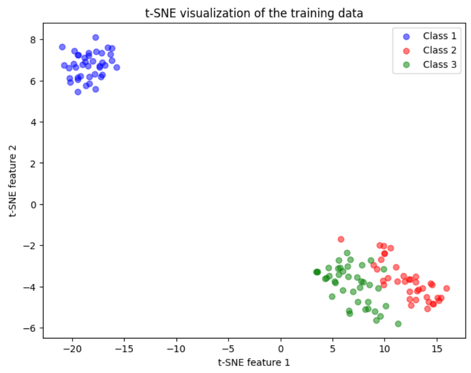

Fig. 2: t-SNE visualization of the training set after preprocessing. Note that class 2 and 3 seems very close to each other, but our classification tree can still distinguish them all.

III. Find Best Classification Tree

This section highlights our journey to discover the optimal configuration of our handmade classification tree , a binary decision tree model that represents our manual implementation of the CART algorithm principles. Utilizing Gini impurity for selecting the best feature and threshold for splits, this model is specifically tailored for our classification tasks. Note that we are not using any off-the-shelf packages.

A. Build a Binary Classification Tree from the Ground Up

Our handmade classification tree is meticulously designed with two crucial stopping conditions to prevent overfitting while ensuring computational efficiency: a predefined maximum depth ('max_depth') and a minimum number of samples ('min_size') required to split a node.

These parameters are essential for creating a model that is not only accurate in its predictive capabilities but also generalized enough to be applicable across various datasets.

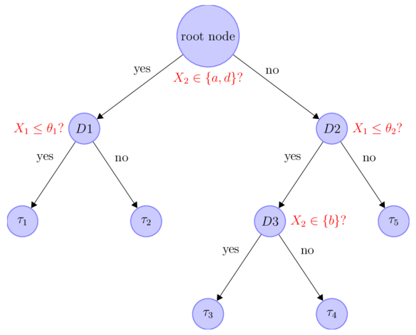

Fig. 3: A visualization of an example classification tree.

This delicate balance between precision and simplicity in our decision-making underscores the effectiveness of fundamental data science methodologies applied with a detailed, hands-on approach.

1) Formulation: Given a dataset D with features $X = \{x_1, x_2, \ldots, x_n\}$ and a target variable Y, a classification tree partitions D recursively. At each node N, the dataset is split into two subsets using the criterion:

$$\text{split}(D, x_i, \theta) = \{(X, Y)|x_i \leq \theta\} \text{ and } \{(X, Y)|x_i > \theta\}, \tag{1}$$

where x_i is a feature from X, and θ (Figure 3 is the threshold value for that feature. The goal is to find the pair (x_i, θ) that maximizes the information gain (IG) or minimizes the Gini impurity (I) across all features, defined as:

$$IG(D, x_i, \theta) = I(D) - \left(\frac{|D_{\text{left}}|}{|D|} I(D_{\text{left}}) + \frac{|D_{\text{right}}|}{|D|} I(D_{\text{right}}) \right), \tag{2}$$

$$I(D) = 1 - \sum_j (p_j)^2, \tag{3}$$

where D_{left} and D_{right} are the partitions of D based on the split, $|D|$ is the number of samples in D, and p_j is the proportion of class j samples in the subset. The process is repeated recursively for D_{left} and D_{right} until a stopping condition is met, such as reaching a maximum depth or a minimum number of samples.

2) Implementation: The core concept of Binary Classification Tree for binary classification is succinctly captured in the *pseudo code* below, which outlines the key functions involved in the process. For a comprehensive view of the implementation please refer to the repository link provided in the abstract.

```
# Calculate the Gini index for a split dataset
def gini_index(self, groups, classes):
    n_instances = float(sum([len(group) for group
    ↪  in groups]))
    gini = 0.0
    for group in groups:
        size = float(len(group))
        if size == 0:
            continue
        score = 0.0
        for class_val in classes:
            p = [row[-1] for row in
            ↪  group].count(class_val) / size
            score += p * p
        gini += (1.0 - score) * (size /
        ↪  n_instances)
    return gini

# Split a dataset
def test_split(self, index, value, dataset):
    left, right = list(), list()
    for row in dataset:
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)
    return left, right

# Create child splits for a node or make terminal
def split(self, node, max_depth, min_size, depth):
    left, right = node['groups']
    del(node['groups'])
    if not left or not right:
        node['left'] = node['right'] =
        ↪  self.to_terminal(left + right)
        return
    if depth >= max_depth:
        node['left'], node['right'] =
        ↪  self.to_terminal(left),
        ↪  self.to_terminal(right)
        return
    if len(left) <= min_size:
        node['left'] = self.to_terminal(left)
    else:
        node['left'] = self.get_split(left)
        self.split(node['left'], max_depth,
        ↪  min_size, depth+1)
    if len(right) <= min_size:
        node['right'] = self.to_terminal(right)
    else:
        node['right'] = self.get_split(right)
        self.split(node['right'], max_depth,
        ↪  min_size, depth+1)

# Build a decision tree
def build_tree(self, train):
    self.root = self.get_split(train)
    self.split(self.root, self.max_depth,
    ↪  self.min_size, 1)

# Make a prediction with this decision tree
def predict(self, node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return self.predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return self.predict(node['right'], row)
        else:
            return node['right']
```

B. *Applying k-fold Validation for Parameter Optimization*

Central to refining our handmade classification tree is k-fold validation, which divides the training set into k subsets for validating the model on one set while training on the others (see Figure 4).

This approach helps fine-tune ‘max_depth‘ and ‘min_size‘, ensuring robust performance across different dataset subsets. By employing k-fold validation, we gain insights into consistency and overfitting resistance, identifying optimal parameters for highest validation accuracy. This strategy enhances our classification tree’s effectiveness, showcasing the value of our comprehensive model development approach.

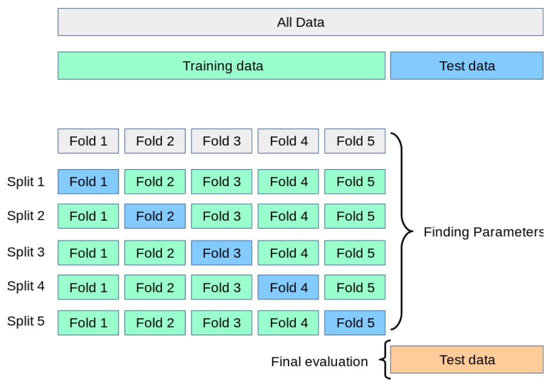

Fig. 4: Work flow of k-fold validation in our study.

1) Formulation: Let D be the dataset consisting of n samples, and Model be the predictive model to be evaluated.

Divide D into k disjoint subsets $\{D_1, D_2, \ldots, D_k\}$, each containing approximately $\frac{n}{k}$ samples.

For i from 1 to k:

- Train Model on the union of all subsets except D_i, denoted as $\cup_{j\neq i} D_j$.
- Validate Model on D_i and compute the performance metric(s) of interest.

After evaluating the model using k-fold validation, select the model with the best performance metric as the ”best” model for making predictions in our case. Since we only perform a single prediction and cannot evaluate test results, this selected model represents the optimal choice for our predictive task.

2) Implementation: The training dataset will be split into validation and training set, with k-fold validation where k=5 (*n_split*).We set few parameters for depths and sizes then comparing models built by these subset of parameters using accuracy on validation dataset (validation score).

```
def cross_validate(X, y, n_splits=5,
↪   max_depth=None, min_size=1, random_state=0):#
↪   set random_state to ensure reproducibility
    kf = KFold(n_splits=n_splits,
    ↪   random_state=random_state, shuffle=True)
    scores = []
    for train_index, test_index in kf.split(X):
        train_X, test_X = X.iloc[train_index],
        ↪   X.iloc[test_index]
        train_y, test_y = y.iloc[train_index],
        ↪   y.iloc[test_index]

        model = HandmadeDecisionTree(max_depth=max⌋
        ↪   _depth,
        ↪   min_size=min_size)
        model.fit(train_X, train_y)
        scores.append(model.score(test_X, test_y))
    return scores

max_depths = [1, 2, 3, 4, 5]
min_sizes = [1, 2, 5, 10]

best_score = 0
best_params = {'max_depth': None, 'min_size': 1}

for max_depth in max_depths:
    for min_size in min_sizes:
        scores = cross_validate(X_train, y_train,
        ↪   n_splits=5, max_depth=max_depth,
        ↪   min_size=min_size)
        avg_score = np.mean(scores)
        print(f"Max Depth: {max_depth}, Min Size:
        ↪   {min_size}, Validation Score:
        ↪   {avg_score}")
        if avg_score > best_score:
            best_score = avg_score
            best_params['max_depth'] = max_depth
            best_params['min_size'] = min_size

print(f"Best Parameters: {best_params}, Best
↪   Score: {best_score}")
```

IV. Results and Discussion

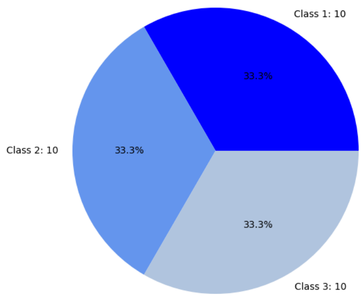

Fig. 5: Class Distribution in the Test Dataset

Our customized classification tree 7, configured with a maximum depth of four and a minimum size of five, achieved a noteworthy accuracy of 93.33% on the validation set. Figures 8 and Table III elucidate the model's performance with a detailed confusion matrix and a comprehensive classification report, respectively.

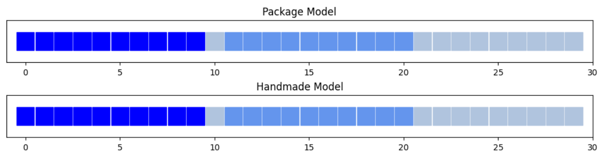

Fig. 6: Per-Sample Classification Outcomes (Indices Starting at 0)

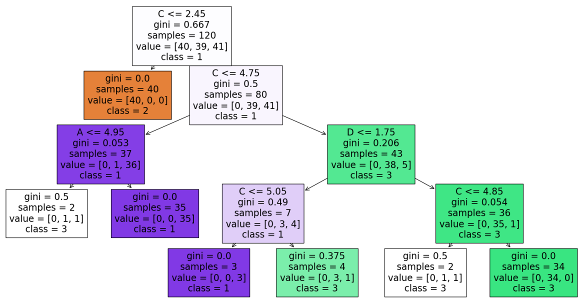

Fig. 7: Visualization of our classification tree

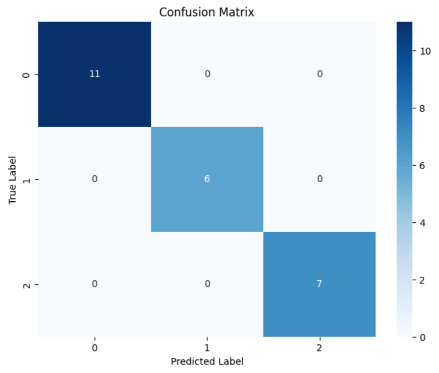

Fig. 8: Confusion Matrix on Validation Set

TABLE III: Classification Report

Class	Precision	Recall	F1-score	Support
1	1.00	1.00	1.00	11
2	1.00	1.00	1.00	6
3	1.00	1.00	1.00	7
Accuracy			1.00	
Macro avg			1.00	
Weighted avg			1.00	

Comparison with existing packaged models indicates that our classification results are on par. Figure 5 depicts the distribution of classes in the test dataset, indicating an equal representation across all three categories. Figure 6 displays the individual classification outcomes for each sample, beginning with index zero.

V. Conclusion

In our quest to determine the optimal model, this study meticulously crafted a classification tree from scratch, sidestepping the use of pre-packaged algorithms. Our journey was marked by the challenge of unlabeled test data and addressed through a careful tuning of the model on a validation set. Employing k-fold cross-validation and bespoke preprocessing methods, such as Stochastic Imputation and Capping, we refined our model to achieve a commendable validation accuracy of 93.33%. This accomplishment mirrors the efficacy of conventionally-used packaged models, despite the absence of labeled test data. This research serves as a testament to our analytical and methodological rigor, and extends its utility to the broader scientific discourse.

References

[1] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[2] Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. *Classification and Regression Trees*. Wadsworth International Group, 1984.

[3] Stef van Buuren. Multiple imputation of discrete and continuous data by fully conditional specification. *Statistical Methods in Medical Research*, 16(3):219–242, 2007.

[4] WJ Dixon. Processing data for outliers. *Biometrics*, 9(1):74–89, 1953.

[5] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *Ijcai*, 14(2):1137–1145, 1995.