CMSC 5724 Project #1: Decision Tree

1155215526 Ding Luxiao *1155215527 Chen Chen*
1155215555 Chen Yubin *1155215807 Zhang Jiarui*

Abstract

In this report, we made a predictive analysis using the Adult dataset. We employed Hung's algorithm to construct a decision tree, leveraging the GINI index as the criterion for optimal splits. The dataset was processed through a series of utility functions designed for data reading, frequency counting, label encoding, and model evaluation. Key optimizations reduced the time complexity for calculating ordered features to *O(nlogn)*, enhancing efficiency in identifying the best split points. The experimental setup involved training on a dataset of 30,162 records and evaluating on a separate set of 15,060 records, with specified parameters for minimum samples per split and maximum depth 10. Results indicated an accuracy of 86.72% on the training set and 85.60% on the evaluation set, with precision, recall, and F1 scores illustrating the model's performance. The decision tree's structure was documented, providing insights into the predictive relationships within the dataset.

Dataset

We will use the Adult dataset to predict whether an individual's annual income exceeds $50K/year based on census data, also known as the "Census Income" dataset. The training set is adult.data, and the evaluation set is adult.test.

Its description is available *https://archive.ics.uci.edu/dataset/2/adult*.

It can be downloaded in *https://archive.ics.uci.edu/static/public/2/adult.zip*.

Algorithm

Using the optimization logic from ex1 Problem 3, we reduced the time complexity for calculating ordered features to *O(n log n)*.The algorithm below is the original Hunt's algorithm.

Hunt's Algorithm:
/* S is the training set; the function returns the root of a decision tree */
1. if all the objects in S belong to the same class
2. return a leaf node with the value of this class

3. if (all the objects in S have the same attribute values) or (|S| is too small)
4. return a leaf node whose class value is the majority one in S
5. find the “best” split attribute A∗ and predicate P∗ using GINI
6. S1 ← the set of objects in R satisfying P∗; S2 ← S \ S1
7. u1 ← Hunt(R1); u2 ← Hunt(R2)
8. create a root u with left child u1 and right child u2
9. set Au ← A∗ and Pu ← P∗
10. return u

We’ve modified the optimized Hunt’s algorithm as below:

1. Define the Dataset: Let S be the set of records in the table. Each record r is in the form (rA, rB), representing its values on attribute A and class label B, respectively.
2. Sort the Dataset: Sort the dataset S in ascending order by the values of attribute A to efficiently calculate the GINI index for each possible split point.
3. Initialize a Counter: Set up a counter c to track the number of positive (yes) records up to and including the current split point.
4. Traverse the Records: After sorting, traverse the records of S in ascending order of A.
5. Update the Counter:
   If the class label B of the record r is positive (yes), increment the counter c by 1.
   For each record r, consider its A attribute value r.A as a potential split point and set c1y(a) to the current value of the counter c, representing the number of positive classes records up to and including the current record r.
6. Calculate the GINI Index for Each Split: Using the counts obtained in the previous steps, calculate the GINI index for each possible split point. The GINI index measures the impurity of a dataset; the lower the value, the higher the purity of the resulting subsets.
7. Select the Best Split Point: Go through all possible split points and choose the one that results in the smallest GINI index as the best split point.

The key to this algorithm is performing a single sorting operation (*O(n log n)* time complexity) and a single traversal (*O(n)* time complexity), allowing efficient calculation of the GINI index for all possible split points, thus identifying the best split point. This method avoids calculating the GINI index for each split point individually, significantly improving efficiency.

Experimental Process

Utility Functions

Includes functions for data reading, frequency counting, label encoding, and model evaluation. Can be referred from Table 1.

Table 1: Functions/Classes presented as utilities for data dealing

Function/Class name	Usage
read_data(path, tHead, paraNumeric, paraRemove, isTest)	Reads data files and converts each line into a dictionary format, stored in a list.
count_frequencies(data)	Counts the frequency of each attribute in the data
get_sorted_list_value(data, key, desending)	Sorts attribute frequencies by value
get_sorted_list_key(data, key, desending)	Sorts attribute frequencies by key
label_encode(data, value_dict)	Encodes nonnumeric labels using numeric substitutes
evaluate_model(labels, predictions)	Computes model evaluation metrics, including accuracy, precision, recall, and F1 score
print_model_evaluation_result(data, predictions)	Print the evaluation result of the model(decision tree) based on adult dataset
print_all_results_of_eval(data, predict)	Print all evaluation data's actual income and predicted income
TreePrinter(tree, paraLabel)	Used to print the structure of the decision tree

Decision Tree Construction

Uses the GINI index as the splitting criterion. Below is a detailed explanation of each part:

1. **Data Preparation:** Use the split_dataset method to divide the dataset into two subsets based on attributes and split points.

```
def split_dataset(self, features, labels, feature_index, split) -> tuple[list]:
    left_features, left_labels = [], []
    right_features, right_labels = [], []
    for i in range(len(features)):
      if feature_index in self.numeric_feature_index: # For numeric
        if features[i][feature_index] <= split:
          left_features.append(features[i])
          left_labels.append(labels[i])
        else:
          right_features.append(features[i])
          right_labels.append(labels[i])
      else: # For non-numeric
        if features[i][feature_index] == split:
          left_features.append(features[i])
          left_labels.append(labels[i])
        else:
          right_features.append(features[i])
          right_labels.append(labels[i])
```

```
    return left_features, left_labels, right_features, right_labels
```

2. **GINI Index Calculation:** Use the gini and gini_split methods to calculate the GINI index for labels, assessing the quality of the splits.

```
def gini(slef, labels: list) -> float:
    total = len(labels)
    if total == 0:
      return 0
    return 1 - ((labels.count(0) / total) ** 2 + (labels.count(1) / total) ** 2)
```

3. **Best Split Point Selection:** The find_best_split_gini method traverses all features and possible split points to find the feature and split value that minimize the GINI index.

```
def find_best_split_gini(self, features, labels) -> tuple:
    best_feature_index = None
    best_split = None
    best_gini = float('inf')
    n_features = len(features[0])
    # optimized version
    for feature_index in range(n_features):
      # find numeric split for numeric data
      if feature_index in self.numeric_feature_index:
        # sort data based on feature
        feature_and_labels = sorted(zip(features, labels), key=lambda x: x[0][fe
ature_index])
        sorted_features = [f for f, _ in feature_and_labels]
        sorted_labels = [l for _, l in feature_and_labels]
        # count yes and no of two subsets
        left_yes_count = 0
        left_no_count = 0
        right_yes_count = sorted_labels.count(1)
        right_no_count = len(sorted_labels) - right_yes_count
        # foreach every possible split and update count number, calculate GINI
        for i in range(1, len(sorted_features)):
          if sorted_labels[i - 1] == 1:
            left_yes_count += 1
            right_yes_count -= 1
          else:
            left_no_count += 1
            right_no_count -= 1
          # ensure splie point available
          if sorted_features[i][feature_index] == sorted_features[i - 1][feature
_index]:
            continue
          # calculate GINI of present split
```

```
31.           total_left = left_yes_count + left_no_count
32.           total_right = right_yes_count + right_no_count
33.           gini_left = 1 - (left_yes_count / total_left) ** 2 - (left_no_count /
    total_left) ** 2
34.           gini_right = 1 - (right_yes_count / total_right) ** 2 - (right_no_coun
    t / total_right) ** 2
35.           weighted_gini = (total_left / len(labels)) * gini_left + (total_right
    / len(labels)) * gini_right
36.           # update best split
37.           if weighted_gini < best_gini:
38.             best_gini = weighted_gini
39.             best_feature_index = feature_index
40.             best_split = (sorted_features[i - 1][feature_index] + sorted_feature
    s[i][feature_index]) / 2
41.       # fing non-numeric split for non-numeric feature
42.       else:
43.         unique_values = set([row[feature_index] for row in features])
44.         for value in unique_values:
45.           _, left_labels, _, right_labels = self.split_dataset(features, labels,
     feature_index, value)
46.           # pass split if size is 0
47.           if len(left_labels) == 0 or len(right_labels) == 0:
48.             continue
49.           # calculate GINI of present split
50.           current_gini = self.gini_split(left_labels, right_labels)
51.           # update best split
52.           if current_gini < best_gini:
53.             best_gini = current_gini
54.             best_feature_index = feature_index
55.             best_split = value
56.     return best_feature_index, best_split
57.
58.     # traditional version
59.     # for feature_index in range(n_features):
60.     #   splits = set([row[feature_index] for row in features])
61.     #   for split in splits:
62.     #     _, left_labels, _, right_labels = self.split_dataset(features, labels,
     feature_index, split)
63.     #     if len(left_labels) == 0 or len(right_labels) == 0:
64.     #       continue
65.     #     current_gini = self.gini_split(left_labels, right_labels)
66.     #     if current_gini < best_gini:
67.     #       best_gini = current_gini
68.     #       best_feature_index = feature_index
```

```
    #       best_split = split
    # return best_feature_index, best_split
```

4. **Tree Construction:** The fit method recursively builds the decision tree until stop conditions are met (e.g., all labels are the same, sample size is less than the minimum split size, or maximum depth is reached).

```
def fit(self, features: list, labels: list, depth=0) -> tuple:
  n_samples = len(labels)
  # return the label if all labels the same
  if len(set(labels)) == 1:
    return labels[0]
  # check for hyper
  if len(features) == 0 or \
          n_samples < self.min_samples_split or \
          (self.max_depth is not None and depth >= self.max_depth):
    # ensure labels are not empty or return None
    return max(set(labels), key=labels.count) if labels else None
  # find the best split of dataset
  best_feature_index, best_split = self.find_best_split_gini(features, labels)
  # return majority label if best split not found
  if best_feature_index is None:
    return max(set(labels), key=labels.count) if labels else None
  # split based on best split point
  left_features, left_labels, right_features, right_labels = self.split_dataset(
features, labels, best_feature_index,

best_split)
  # in case for empty subset
  if len(left_labels) == 0 or len(right_labels) == 0:
    return max(set(labels), key=labels.count) if labels else None
  # generate left tree and right tree
  left_subtree = self.fit(left_features, left_labels, depth + 1)
  right_subtree = self.fit(right_features, right_labels, depth + 1)
  return (best_feature_index, best_split, left_subtree, right_subtree)
```

5. **Prediction**: The predict method uses the constructed decision tree to make predictions on new datasets.

```
def _predict(self, sample, tree):
    if not isinstance(tree, tuple):
      return tree
    feature_index, split, left_subtree, right_subtree = tree
    if feature_index in self.numeric_feature_index:
      if sample[feature_index] <= split:  # For numeric
        return self._predict(sample, left_subtree)
      else:
        return self._predict(sample, right_subtree)
```

```
    else:  # For non-numeric
      if sample[feature_index] == split:
        return self._predict(sample, left_subtree)
      else:
        return self._predict(sample, right_subtree)

def predict(self, features: list[list]) -> list:
    return [self._predict(sample, self.tree) for sample in features]
```

Experimental Result

We've conducted evaluations on the training and evaluation sets with training size: 30,162 and evaluation size: 15,060, using different MIN_SAMPLE_SPLIT(MSS) and MAX_DEPTH(MD) value pairs. Results for training sets are experimented and shown in Table 2.

Table 2: Experimental Result of Different Hyperparameter Pairs

MD \ MSS	5	10	15	20	25
5(train)	0.84275	0.84275	0.84275	0.84258	0.84258
5(evaluate)	0.84064	0.84064	0.84064	0.84064	0.84064
10(train)	0.86778	0.86718	0.86652	0.86546	0.86493
10(evaluate)	0.85564	0.85604	0.85578	0.85558	0.85604
15(train)	0.89643	0.89079	0.88631	0.88373	0.88210
15(evaluate)	0.84017	0.84243	0.84420	0.84502	0.84542
20(train)	0.92726	0.91423	0.90624	0.90109	0.89679
20(evaluate)	0.82324	0.82875	0.83353	0.83552	0.83772
25(train)	0.95103	0.93057	0.91874	0.91121	0.90587
25(evaluate)	0.81308	0.82131	0.82709	0.82968	0.83227

From the result mentioned in Table 2, we can easily find out that when MAX_DEPTH is larger than 15, the decision tree is more probable of overfit while the expansion of MIN_SAMPLE_SPLIT can mitigate overfit to some extent. For the result with best evaluation accuracy, having set MIN_SAMPLE_SPLIT=10/25 and MAX_DEPTH=10, the other evaluation scores are shown in Table 3 and Table 4.

Table 3: Evaluation Result of MSS=10 and MD=10

Dataset	Accuracy	Precision	Recall	F1 Score
Training Set	0.86718	0.79779	0.62480	0.70078
Evaluation Set	0.85604	0.76505	0.59757	0.67102

Table 4: Evaluation Result of MSS=25 and MD=10

Dataset	Accuracy	Precision	Recall	F1 Score
Training Set	0.86718	0.79171	0.62067	0.69583
Evaluation Set	0.85604	0.76560	0.59676	0.67072

Besides, we've kept the complete structure of the decision tree and the evaluation set prediction with MSS=10 and MD=10 in the file res.txt.

Appendix:

1. GitHub repo: https://github.com/Chenchen2001/5724Proj1

2. Part of Output in res.txt Demo

traing data size: 30162 evaluation data set: 15060
min_samples_split=10, max_depth=10
========== TRAINING MODEL on Training SET ==========
========== TRAINING MODEL FINISHED ==========
========== EVALUATION on Training SET ==========
Accuracy: 0.86718
Precision: 0.79779
Recall: 0.62480
F1 Score: 0.70078
========== EVALUATION on Evaluation SET ==========
Accuracy: 0.85604
Precision: 0.76505
Recall: 0.59757
F1 Score: 0.67102
========== DECISION TREE RESULT ==========
|- martial-status is Married-civ-spouse
L |- education-num <= 12.5
L L |- capital-gain <= 5095.5
L L L |- education-num <= 8.5
L L L L |- capital-loss <= 1791.5
L L L L L |- age <= 36.5
L L L L L L |- occupation is Tech-support
.....
R R R L R |- INCOME >50k
R R R R |- capital-gain <= 30961.5
R R R R L |- INCOME >50k
R R R R R |- capital-gain <= 67047.0
R R R R R L |- INCOME <=50k
R R R R R R |- INCOME >50k

========== PREDICT RESULT OF EVALUATION SET ==========
SAMPLE 1 : actual result - <=50k predicted result - <=50k predict True.
SAMPLE 2 : actual result - <=50k predicted result - <=50k predict True.
SAMPLE 3 : actual result - >50k predicted result - <=50k predict False.
SAMPLE 4 : actual result - >50k predicted result - >50k predict True.
SAMPLE 5 : actual result - <=50k predicted result - <=50k predict True.
SAMPLE 6 : actual result - >50k predicted result - >50k predict True.
...