Cairo University

Faculty of Engineering

Systems and Biomedical Department

Decision tree Implementation

Submitted by:

Ashar Seif Al-Naser Saleh

Sec: 1 BN: 9

SBE452_AI

Dr.Inas A. Yassine

11 December, 2021

- **Problem (1:B)**

  Recommend another type of trees than ID3 that would build a less deep tree than ID3.

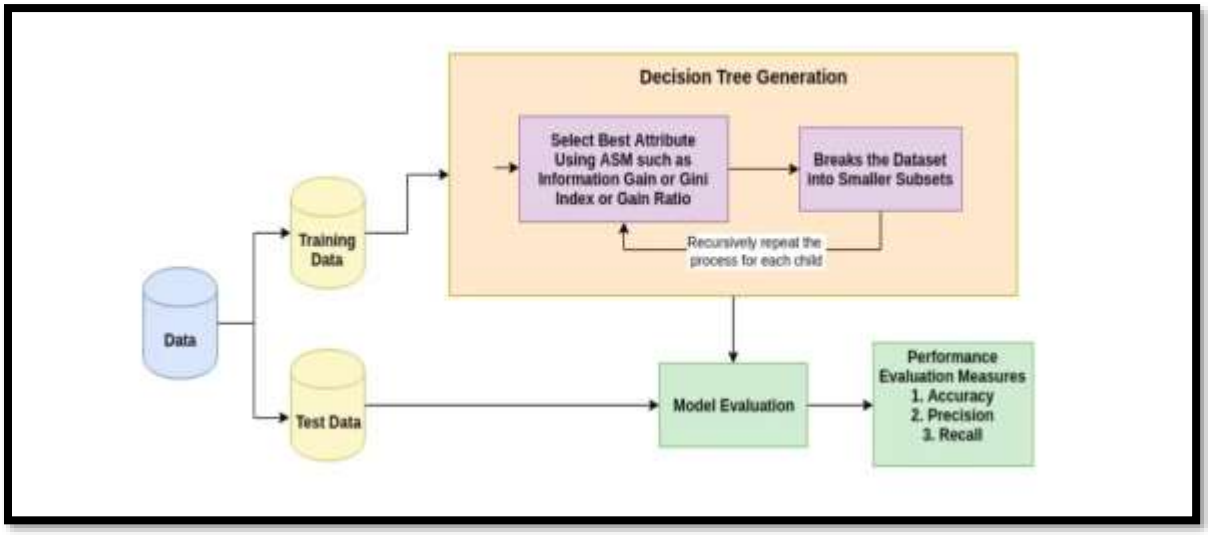
  The recommended type is C4.5 where it has the characteristic of Error Based pruning which minimize the trees size by cutting the less important leaves out unlike the ID3 where No pruning is done.

- **Problem (2)**

How does the Decision Tree algorithm work?

The basic idea behind any decision tree algorithm is as follows:

- Select the best attribute using Attribute Selection Measures (ASM) to split the records and we will use entropy or gini_index in this step.
- Make that attribute a decision node and breaks the dataset into smaller subsets.
- Starts tree building by repeating this process recursively for each child until one of the condition will match:
  - All the tuples belong to the same attribute value.
  - There are no more remaining attributes.
  - There are no more instances.



The sequence of code implementation functions

- Reading data using dataframe

```
1.    rows=[]
2.
3.    with open(r"CardioVascular\cardio_train.csv", 'r') as file:
4.      reader = csv.reader(file, delimiter = ';')
5.      for df_row in reader:
6.          rows.append(df_row)
7.
8.      # Create the dataframe
9.      df = pd.DataFrame(rows)
10.     df=df.iloc[1:, 2:]
```

- Splitting the data into train and test data using the train_test_split function which split data corresponding to the percentage user enter to determine the size of train and test data.

```
train_data,test_data=train_test_split(df,0.9)
```

```
def train_test_split(data,percentage):
        length=data.shape[0]
    percent=int(length*percentage)
    train_data=data.iloc[:percent+1,:]
    test_data=data.iloc[percent+1:,:]
    return  train_data, test_data
```

- Use fit_predict function to first train the data and build the tree and this is done by multiple steps :
    1. build_tree function which builds the decision tree with the desired depth and size of tree using two other functions (best_split and child_nodes)

```
def build_tree(train, max_depth, min_size,split_method):
    root = best_split(train,split_method)
```

```
    child_nodes(root, max_depth, min_size, 1,split_method)
    return root
```

- best_split function gets the best root node to split data on it using one of Attribute Selection Measures (ASM) determined by user .
- It calculates the ASM (entropy (information gain) or gini_index) for each feature to get the node impurity in case of using it as a root node and takes the minimum value of the gini to consider its feature as a root node or the maximum value of information gain .
- In case of features with multiple values (Which can be considered as continuous, Ex: height) , I put a condition which discretize the values into categories to avoid entering an infinite loop. The condition state that if there are more than nine different values for a feature then we split these values into number of groups which equaled to :

number_of_categories=int(len(uniqueValues)/5)

```
def best_split(data,split_method):

    """
    Get the best root node to split data on it

    Parameters
    ----------
         data: n dimensional array-like, shape (n_samples, n_features)

         Returns
         -------
         index : The index of the root
         value : The value at which we split the node and put the spliiting
condition on it .
         groups : The splitted feature values groups after comparing with the
splitting condition.
    """
    data=np.array(data)
    class_values = list(set(row[-1] for row in data))
    max_gini=0.5
    max_entropy=0.3
```

```
    splitted_groups=None
    features_length=len(data[0])-1
    for feature in range(features_length):
        uniqueValues = np.unique(data[:,feature])
        if len(uniqueValues)>9:
            descritized_data=data[:,feature]
            sorted_feature_values=np.sort(data[:,feature])
            number_of_categories=int(len(uniqueValues)/5)
            feature_value_groups=[ sorted_feature_values[i:i +
number_of_categories] for i in range(0, len(sorted_feature_values),
number_of_categories)]
            for category in range(len(feature_value_groups)):
                for i in range(len(descritized_data)):
                    if descritized_data[i] in feature_value_groups[category]:
                        descritized_data[i]=category
            data[:,feature]= descritized_data

        for row in data:
            splitted_groups=feature_split(feature,row[feature],data)
            if split_method == "gini_index":
               gini=gini_index(splitted_groups,class_values)
               print(gini)
               if(gini<max_gini):
                   feature_index=feature
                   feature_value=row[feature]
                   max_gini=gini
                   feature_groups=splitted_groups
            elif split_method == 'entropy':
                Information_Gain=entropy(splitted_groups,class_values)
                if(Information_Gain>max_entropy):
                   feature_index=feature
                   print(1)
                   feature_value=row[feature]
                   max_IG=Information_Gain
                   feature_groups=splitted_groups

    return {'index':feature_index, 'value':feature_value, 'groups':
feature_groups}
```

- 2- child_nodes : Create child nodes after determine the root node or the decision node, It do multiple functions , first :if it is the first split , it put the groups (left, right) as leaf nodes, second :it checks

if the tree got to its desired size, if not it process a node as a leaf node (right or left according to the uncompleted size direction.

```
def child_nodes(node, max_depth, min_size, depth,split_method):
    # Create child nodes after determine the root node or the decision node
    left, right = node['groups']
    del(node['groups'])
    # check for first split
    if not left or not right:
        node['left'] = node['right'] = leaf_node(left + right)
        return
    # check for number of desired splits
    if depth >= max_depth:
        node['left'], node['right'] = leaf_node(left), leaf_node(right)
        return
    # make a left leaf node if the desired size is not achieved
    if len(left) <= min_size:
        node['left'] = leaf_node(left)
    else:
        node['left'] = best_split(left,split_method)
        child_nodes(node['left'], max_depth, min_size, depth+1,split_method)
    # make a right leaf node if the desired size is not achieved
    if len(right) <= min_size:
        node['right'] = leaf_node(right)
    else:
        node['right'] = best_split(right,split_method)
        child_nodes(node['right'], max_depth, min_size, depth+1,split_method)
```

- After these steps , the decision tree is ready and then we predict values for the test data using two functions :
  1. Compare: which apply the tree on every row on test data and return the prediction values.

```
2. def compare(tree, row):
3.     # Compare test features groups with the created tree to make a
   prediction.
4.     if row[tree['index']] < tree['value']:
5.         if isinstance(tree['left'], dict):
6.             return compare(tree['left'], row)
7.         else:
```

```
            return tree['left']
    else:
        if isinstance(tree['right'], dict):
            return compare(tree['right'], row)
        else:
            return tree['right']

```

2. accuracy_metric : which takes the predicted values and the actual values of test data and evaluates the accuracy of the prediction .

```
def accuracy_metric(actual, predicted):
    # Get the performance measure (accuracy) of the implemented algorithm
    # Parameters : actual: 1D-array ,shape(n_samples,) presents the actual values
of classes (labels) of the testing trials.
    # predicted  : 1D-array ,shape(n_samples,) presents the values of classes
(labels) of the testing trials.
    true = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            true += 1
    return ((true/float(len(actual))*100))
```

Observation on accuracy :The accuracy of the implementation increase with the decreasing of number of samples from the data (It reaches 100% using only 15 sample in both gini and entropy) but in general it gives around 66%.