# Program – 12

## AIM: To Implement Decision Tree classifier in python

### Introduction and Theory

Decision tree induction is the learning of decision trees from class-labeled training tuples. A decision tree is a flowchart-like tree structure, where each internal node nonleaf node) denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (or terminal node) holds a class label. The topmost node in a tree is the root node.

Given a tuple, X, for which the associated class label is unknown, the attribute values of the tuple are tested against the decision tree. A path is traced from the root to a leaf node, which holds the class prediction for that tuple. Decision trees can easily be converted to classification rules.

ID3, C4.5, and CART adopt a greedy (i.e., nonbacktracking) approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow a top-down approach, which starts with a training set of tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built.

**Algorithm: Generate_decision_tree.** Generate a decision tree from the training tuples of data partition, *D*.

**Input:**

- Data partition, *D*, which is a set of training tuples and their associated class labels;
- *attribute_list*, the set of candidate attributes;
- *Attribute_selection_method*, a procedure to determine the splitting criterion that “best” partitions the data tuples into individual classes. This criterion consists of a *splitting_attribute* and, possibly, either a *split-point* or *splitting subset*.

**Output:** A decision tree.

**Method:**

(1) create a node *N*;<br>
(2) **if** tuples in *D* are all of the same class, *C*, **then**<br>
(3)     return *N* as a leaf node labeled with the class *C*;<br>
(4) **if** *attribute_list* is empty **then**<br>
(5)     return *N* as a leaf node labeled with the majority class in *D*; // majority voting<br>
(6) apply **Attribute_selection_method**(*D*, *attribute_list*) to **find** the “best” *splitting_criterion*;<br>
(7) label node *N* with *splitting_criterion*;<br>
(8) **if** *splitting_attribute* is discrete-valued **and**<br>
    multiway splits allowed **then** // not restricted to binary trees<br>
(9)     *attribute_list* ← *attribute_list* − *splitting_attribute*; // remove *splitting_attribute*<br>
(10) **for each** outcome *j* of *splitting_criterion*<br>
    // partition the tuples and grow subtrees for each partition<br>
(11)     let $D_j$ be the set of data tuples in *D* satisfying outcome *j*; // a partition<br>
(12)     **if** $D_j$ is empty **then**<br>
(13)         attach a leaf labeled with the majority class in *D* to node *N*;<br>
(14)     **else** attach the node returned by **Generate_decision_tree**($D_j$, *attribute_list*) to node *N*;<br>
**endfor**<br>
(15) return *N*;

An attribute selection measure is a heuristic for selecting the splitting criterion that “best” separates a given data partition, D, of class-labeled training tuples into individual classes. If we were to split D into smaller partitions according to the outcomes of the splitting criterion, ideally each partition would be pure (i.e., all the tuples that fall into a given partition would belong to the same class). Conceptually, the “best” splitting criterion is the one that most closely results in such a scenario. Attribute selection measures are also known as splitting rules because they determine how the tuples at a given node are to be split.

ID3 uses Information gain:

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i)$$

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j)$$

$$Gain(A) = Info(D) - info_A(D)$$

C4.5 uses gain ratio

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2 \frac{|D_j|}{|D|}$$

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}$$

CART uses the GINI index

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2$$

$$Gini_A(D) = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2)$$

$$\Delta Gini(A) = Gini(D) - Gini_A(D)$$

## Strengths and Weakness of Decision Tree approach

**Strengths:**

- Decision trees are able to generate understandable rules.
- Decision trees perform classification without requiring much computation.
- Decision trees are able to handle both continuous and categorical variables.
- Decision trees provide a clear indication of which fields are most important for prediction or classification.

**Weaknesses:**

- Decision trees are less appropriate for estimation tasks where the goal is to predict the value of a continuous attribute.
- Decision trees are prone to errors in classification problems with many class and relatively small number of training examples.
- Decision tree can be computationally expensive to train. The process of growing a decision tree is computationally expensive. At each node, each candidate splitting field must be sorted before its best split can be found. In some algorithms, combinations of fields are used and a search must be made for optimal combining weights. Pruning algorithms can also be expensive since many candidate sub-trees must be formed and compared.

# Program – 12

## Code

```
# importing libraries
import pandas as pd
import numpy as np
from pprint import pprint

# calculates the entropy
def entropy(target_col):
    elements, counts = np.unique(target_col, return_counts=True)
    result = np.sum([(-
counts[i]/np.sum(counts))*np.log2(counts[i]/np.sum(counts)) for i in
range(len(elements))])
    return result

# calculate the information gain = Gain(A) / SplitInfo_A(D)
def information_gain(data, split_attribute_name,
target_name="class"):
    """
    calculates the information gain of the dataset for attribute A
    :param data: the dataset for which we're calculating the
information gain
    :param split_attribute_name: name of the splitting attribute.
    :param target_name: name of the target / class.
    :return: the information gain
    """
    # Calculate the entropy of the total dataset
    total_entropy = entropy(data[target_name])

    # Calculate the values and the corresponding counts for the
split attribute
    vals, counts = np.unique(data[split_attribute_name],
return_counts=True)
    # Calculate the weighted entropy
    weighted_entropy = np.sum(
        [(counts[i] / np.sum(counts)) *
entropy(data.where(data[split_attribute_name] ==
vals[i]).dropna()[target_name])
         for i in range(len(vals))])
    # Calculate the information gain
    information_gain_ = total_entropy - weighted_entropy
    return information_gain_

# the ID3 Decision tree algorithm
def id3(data, original_data, features,
target_attribute_name="class", parent_node_class=None):
    """
    id3 algorithm
    :param data: the dataset for which we're running the id3
algorithm
    :param original_data: the original dataset needed to calculate
the mode target feature value of the original dataset
    in the case the dataset delivered by the first parameter is
empty
    :param features: the feature space of the dataset . This is
needed for the recursive call since during the tree

```

## Program – 12

```
    growing process we have to remove features from our dataset ->
Splitting at each node
    :param target_attribute_name: the name of the target attribute
    :param parent_node_class: This is the value or class of the mode
target feature value of the parent node for a
    specific node. This is also needed for the recursive call since
if the splitting leads to a situation that there are
     no more features left in the feature space, we want to return
the mode target feature value of the direct parent node.

    :return: The learnt decision tree
    """
    # Define the stopping criteria -> If one of this is satisfied,
we want to return a leaf node
    # If all target_values have the same value, return this value
    if len(np.unique(data[target_attribute_name])) <= 1:
        return np.unique(data[target_attribute_name])[0]
    # If the dataset is empty, return the mode target feature value
in the original dataset
    elif len(data) == 0:
        return np.unique(original_data[target_attribute_name])[

np.argmax(np.unique(original_data[target_attribute_name],
return_counts=True)[1])]
    # If the feature space is empty, return the mode target feature
value of the direct parent node -> Note that
    # the direct parent node is that node which has called the
current run of the ID3 algorithm and hence
    # the mode target feature value is stored in the
parent_node_class variable.
    elif len(features) == 0:
        return parent_node_class
    # If none of the above holds true, grow the tree!
    else:
        # Set the default value for this node -> The mode target
feature value of the current node
        parent_node_class = np.unique(data[target_attribute_name])[
            np.argmax(np.unique(data[target_attribute_name],
return_counts=True)[1])]
        # Select the feature which best splits the dataset
        item_values = [information_gain(data, feature,
target_attribute_name) for feature in
                       features]  # Return the information gain
values for the features in the dataset
        best_feature_index = np.argmax(item_values)
        best_feature = features[best_feature_index]
        # Create the tree structure. The root gets the name of the
feature (best_feature) with the maximum information
        # gain in the first run
        tree = {best_feature: {}}
        # Remove the feature with the best information gain from the
feature space
        features = [i for i in features if i != best_feature]
        # Grow a branch under the root node for each possible value
of the root node feature
        for value in np.unique(data[best_feature]):
            value = value
```

## Program – 12

```
            # Split the dataset along the value of the feature with
the largest
            # information gain and then create sub_datasets
            sub_data = data.where(data[best_feature] ==
value).dropna()

            # Call the ID3 algorithm for each of those sub_datasets
with the
            # new parameters -> Here the recursion comes in!
            subtree = id3(sub_data, dataset, features,
target_attribute_name, parent_node_class)

            # Add the sub tree, grown from the sub_dataset to the
tree under the root node
            tree[best_feature][value] = subtree

        return tree


def predict(query, tree, default=1):
    """
    prediction function for new unseen data
    :param query: a dictionary entry, {"feature_name" : value, ... ,
"feature_name" : value}
    :param tree: the ID3 tree
    :param default: base value
    :return: predicted class
    """
    # 1.Check for every feature in the query instance if this feature is
existing in the tree.keys() for the first call, tree.keys() only contains
the value for the root node -> if this value is not existing, we can not
make a prediction and have to return the default value which is the majority
value of the target feature
    for key in list(query.keys()):
        if key in list(tree.keys()):
            # 2. First of all we have to take care of a important
fact: Since we train our model with a database A and then show our
model a unseen query it may happen that the feature values of these
query are not existing in our tree model because non of the training
instances has had such a value for this specific feature.
            #    For instance imagine the situation where your model
has only seen animals with one to four legs - The "legs" node in
your model will only have four outgoing branches (from one to four).
If you now show your model a new instance (animal) which has for the
legs feature the vale 5, you have to tell your model what to do in
such a situation because otherwise there is no classification
possible
            #    because in the classification step you try to run
down the outgoing branch with the value 5 but there is no such a
branch. Hence: Error and no Classification! We can address this
issue with a classification value of for instance (999) which tells
us that there is no classification possible or we assign the most
frequent target feature value of our dataset used to train the
model. Or, in for instance medical application we can return the
most worse case - just to make sure...  We can also return the most
frequent value of the direct parent node. To make a long story
short, we have to tell the model what to do in this situation. In
```

# Program – 12

```
our example, since we are dealing with animal species where a false
classification is not that critical, we will assign the value 1
which is the value for the mammal species (for convenience).
            try:
                result = tree[key][query[key]]
            except:
                return default

            # 3. Address the key in the tree which fits the value
for key --> Note that key == the features in the query. Because we
want the tree to predict the value which is hidden under the key
value (imagine you have a drawn tree model on the table in front of
you and you have a query instance for which you want to predict the
target feature - What are you doing? - Correct: You start at the
root node and wander down the tree comparing your query to the node
values. Hence you want to have the value which is hidden under the
current node. If this is a leaf, perfect, otherwise you wander the
tree deeper until you get to a leaf node. Though, you want to have
this "something" [either leaf or sub_tree] which is hidden under the
current node and hence we must address the node in the tree which ==
the key value from our query instance.  This is done with
tree[keys]. Next you want to run down the branch of this node which
is equal to the value given "behind" the key value of your query
instance e.g. if you find "legs" == to tree.keys() that is, for the
first run == the root node. You want to run deeper and therefore you
have to address the branch at your node whose value is == to the
value behind key. This is done with query[key]
            #    e.g. query[key] == query['legs'] == 0 --> Therewith
we run down the branch of the node with the value 0. Summarized, in
this step we want to address the node which is hidden behind a
specific branch of the root node (in the first run) this is done
with: result = [key][query[key]]
            result = tree[key][query[key]]
            # 4. As said in the 2. step, we run down the tree along
nodes and branches until we get to a leaf node.  That is, if result
= tree[key][query[key]] returns another tree object (we have
represented this by a dict object -> that is if result is a dict
object) we know that we have not arrived at a root node and have to
run deeper the tree.  Okay... Look at your drawn tree in front of
you... what are you doing?...well, you run down the next branch...
exactly as we have done it above with the slight difference that we
already have passed a node and therewith have to run only a fraction
of the tree --> You clever guy! That "fraction of the tree" is
exactly what we have stored under 'result'.  So we simply call our
predict method using the same query instance (we do not have to drop
any features from the query instance since for instance the feature
for the root node will not be available in any of the deeper
sub_trees and hence we will simply not find that feature) as well as
the "reduced / sub_tree" stored in result.
            if isinstance(result, dict):
                return predict(query, result)
            else:
                return result
# splitting the data into test-train splits for checking performance
on unseen data
def train_test_split(dataset):

```

## Program – 12

```
    training_data = dataset.iloc[:80].reset_index(drop=True)   # We drop the index respectively relabel the index
    # starting form 0, because we do not want to run into errors regarding the row labels / indexes
    testing_data = dataset.iloc[80:].reset_index(drop=True)
    return training_data, testing_data

# testing the tree model, get prediction accuracy
def test(data, tree):
    # Create new query instances by simply removing the target feature column from the original dataset and
    # convert it to a dictionary
    queries = data.iloc[:, :-1].to_dict(orient="records")

    # Create a empty DataFrame in whose columns the prediction of the tree are stored
    predicted = pd.DataFrame(columns=["predicted"])

    # Calculate the prediction accuracy
    for i in range(len(data)):
        predicted.loc[i, "predicted"] = predict(queries[i], tree, 1.0)
    print('The prediction accuracy is: ', (np.sum(predicted["predicted"] == data["class"]) / len(data)) * 100, '%')


"""
Train the tree, Print the tree and predict the accuracy
"""

if __name__ == '__main__':
    # loading the dataset
    dataset = pd.read_csv('zoo.csv', names=['animal_name', 'hair', 'feathers', 'eggs', 'milk',
                                             'airbone', 'aquatic', 'predator', 'toothed', 'backbone',
                                             'breathes', 'venomous', 'fins', 'legs', 'tail', 'domestic', 'catsize',
                                             'class'])
    print(dataset.head(10))
    # dropping the class column
    dataset = dataset.drop('animal_name', axis=1)
    print(dataset.head(10))
    # splitting data
    training_data = train_test_split(dataset)[0]
    testing_data = train_test_split(dataset)[1]
    # training the tree
    tree = id3(training_data, training_data, training_data.columns[:-1])
    # printing the learnt tree in the form of a dictionary
    pprint(tree)
    # get test performance of the tree
    test(testing_data, tree)
```

# Program – 12

## Results and Outputs:

```
DWDM_LAB — -bash — 128×41

(ML) Anurags-MacBook-Air:DWDM_LAB jarvis$ python DecisionTree.py
  animal_name  hair  feathers  eggs  milk  airbone  aquatic  ...   venomous  fins  legs  tail  domestic  catsize  class
0    aardvark     1         0     0     1        0        0  ...          0     0     4     0         0        1      1
1    antelope     1         0     0     1        0        0  ...          0     0     4     1         0        1      1
2        bass     0         0     1     0        0        1  ...          0     1     0     1         0        0      4
3        bear     1         0     0     1        0        0  ...          0     0     4     0         0        1      1
4        boar     1         0     0     1        0        0  ...          0     0     4     1         0        1      1
5     buffalo     1         0     0     1        0        0  ...          0     0     4     1         0        1      1
6        calf     1         0     0     1        0        0  ...          0     0     4     1         1        1      1
7        carp     0         0     1     0        0        1  ...          0     1     0     1         1        0      4
8     catfish     0         0     1     0        0        1  ...          0     1     0     1         0        0      4
9        cavy     1         0     0     1        0        0  ...          0     0     4     0         1        0      1

[10 rows x 18 columns]
   hair  feathers  eggs  milk  airbone  aquatic  predator  ...   venomous  fins  legs  tail  domestic  catsize  class
0     1         0     0     1        0        0         1  ...          0     0     4     0         0        1      1
1     1         0     0     1        0        0         0  ...          0     0     4     1         0        1      1
2     0         0     1     0        0        1         1  ...          0     1     0     1         0        0      4
3     1         0     0     1        0        0         1  ...          0     0     4     0         0        1      1
4     1         0     0     1        0        0         1  ...          0     0     4     1         0        1      1
5     1         0     0     1        0        0         0  ...          0     0     4     1         0        1      1
6     1         0     0     1        0        0         0  ...          0     0     4     1         1        1      1
7     0         0     1     0        0        1         0  ...          0     1     0     1         1        0      4
8     0         0     1     0        0        1         1  ...          0     1     0     1         0        0      4
9     1         0     0     1        0        0         0  ...          0     0     4     0         1        0      1

[10 rows x 17 columns]
{'legs': {0: {'fins': {0.0: {'toothed': {0.0: 7.0, 1.0: 3.0}},
                       1.0: {'eggs': {0.0: 1.0, 1.0: 4.0}}}},
          2: {'hair': {0.0: 2.0, 1.0: 1.0}},
          4: {'hair': {0.0: {'toothed': {0.0: 7.0, 1.0: 5.0}}, 1.0: 1.0}},
          6: {'aquatic': {0.0: 6.0, 1.0: 7.0}},
          8: 7.0}}
The prediction accuracy is:  85.71428571428571 %
(ML) Anurags-MacBook-Air:DWDM_LAB jarvis$
(ML) Anurags-MacBook-Air:DWDM_LAB jarvis$
(ML) Anurags-MacBook-Air:DWDM_LAB jarvis$
(ML) Anurags-MacBook-Air:DWDM_LAB jarvis$
```

## Findings and Learnings:

1. We have Implemented Decision tree through the ID3 algorithm in python 3.
2. We have learned the nuances of the Decision tree learning.
3. We have learnt about the applications, strengths and weaknesses of Decision tree Learning.