import pandas as pd

import numpy as np

import math

*pandas is used for data manipulation and reading datasets.*

*numpy is used for array operations.*

*math provides mathematical functions like logarithms (used for entropy calculation).*

class Node:

def \_\_init\_\_(self):

self.children = []

self.value = ""

self.isLeaf = False

self.pred = ""

*This class represents a node in the decision tree.*

*children: A list of child nodes.*

*value: The feature or decision at the current node.*

*isLeaf: A boolean flag that is True if the node is a leaf (final decision).*

*pred: Holds the prediction label if the node is a leaf.*

def entropy(data):

yes = 0.0

no = 0.0

for \_, rows in data.iterrows():

if rows["Play Tennis"] == "Yes":

yes += 1

elif rows["Play Tennis"] == "No":

no += 1

if yes == 0.0 or no == 0.0:

return 0

else:

py = yes / (yes + no)

pn = no / (yes + no)

return -(py \* math.log(py, 2) + pn \* math.log(pn, 2))

*This function computes the* ***entropy*** *of the dataset.*

*Entropy measures the impurity or uncertainty in the dataset.*

*If there are no instances of either "Yes" or "No," entropy is 0, implying no uncertainty.*

*Otherwise, it calculates entropy based on the proportion of "Yes" and "No" values using the formula:* 

**Initial Setup**

The entropy for a given dataset based on the target attribute (which has two possible values: "Yes" or "No"). Here, two counters yes and no are initialized to 0. These will count how many "Yes" and "No" values are present in the "Play Tennis" column of the dataset.

**Loop Over Dataset**

for \_, rows in data.iterrows():

if rows["Play Tennis"] == "Yes":

yes += 1

elif rows["Play Tennis"] == "No":

no += 1

The function iterates over every row in the dataset. For each row, it checks the value in the "Play Tennis" column:

If the value is "Yes", the yes counter is incremented.

If the value is "No", the no counter is incremented.

By the end of this loop, yes will contain the count of "Yes" values, and no will contain the count of "No" values in the "Play Tennis" column.

Handle Edge Case: All Yes or All No

If either yes or no is 0, it means all the labels in the dataset are either "Yes" or "No".

In such a case, the entropy is 0 because there's no uncertainty (perfect classification). For example, if all values are "Yes", then there's no randomness or impurity.

Thus, the function returns 0 in this case.

**Calculate Probabilities**

py is the probability of "Yes" (i.e., yes divided by the total number of records).

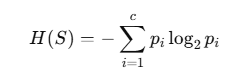
pn is the probability of "No" (i.e., no divided by the total number of records).

These are basic probabilities based on the counts of "Yes" and "No".

Calculate Entropy

return -(py \* math.log(py, 2) + pn \* math.log(pn, 2))

Entropy is calculated using the formula:



Where p\_i is the probability of class i. In this case, there are two classes ("Yes" and "No"):

py \* math.log(py, 2) is the contribution to entropy from the "Yes" class.

pn \* math.log(pn, 2) is the contribution to entropy from the "No" class.

The result is multiplied by -1 because entropy is always a positive value, and this formula gives a measure of uncertainty or randomness in the data.

def info\_gain(dataset, feature):

attributes = np.unique(dataset[feature])

gain = entropy(dataset)

for attr in attributes:

subdata = dataset[dataset[feature] == attr]

sub\_e = entropy(subdata)

gain -= (float(len(subdata)) / float(len(dataset)) \* sub\_e)

return gain

*Information gain is the reduction in entropy after splitting the dataset based on a feature.*

*It starts by calculating the entropy of the full dataset.*

*Then, it splits the dataset for each unique value of the feature, computes the entropy of each subset, and subtracts the weighted entropy from the original entropy to get the information gain.*

**1. def info\_gain(dataset, feature):**

Defines a function named info\_gain that takes two parameters:

dataset: The data on which the calculation is to be performed (usually a DataFrame).

feature: The feature (or attribute) for which we want to calculate the information gain.

**2. attributes = np.unique(dataset[feature])**

Finds all unique values (or categories) of the given feature in the dataset using np.unique().

Example: If the feature is "Wheather" in a weather dataset, attributes could be ["Sunny", "Cloudy", "Rain"].

3. gain = entropy(dataset)

Calls the entropy() function (defined elsewhere) to compute the entropy of the entire dataset.

This is the initial entropy before splitting by the chosen feature, representing the uncertainty in the target variable ("Dataset").

4. for attr in attributes:

Iterates through each unique value (category) of the given feature. The goal is to calculate the entropy for the subset of data where the feature takes each of its unique values.

5. subdata = dataset[dataset[feature] == attr]

Creates a subset of the dataset where the feature has the value attr.

Example: If the feature is "Wheather" and attr is "Sunny", subdata contains only the rows where the "Wheather" is "Sunny".

**6. sub\_e = entropy(subdata)**

Calls the entropy() function to calculate the entropy of this subset (subdata).

The goal is to understand how much uncertainty (entropy) remains in the subset after splitting by this particular feature value.

**7. gain -= (float(len(subdata)) / float(len(dataset)) \* sub\_e)**

Updates the gain by subtracting the weighted entropy of the subset.

len(subdata) is the number of rows in the subset (where the feature equals attr).

len(dataset) is the total number of rows in the original dataset.

float(len(subdata)) / float(len(dataset)) gives the proportion of the subset relative to the whole dataset (the "weight").

sub\_e is the entropy of the subset.

Weighted Entropy: The entropy of the subset is weighted by its proportion in the dataset.

The information gain decreases by this amount because it represents how much uncertainty remains in the subsets.

**8. return gain**

Returns the final information gain after iterating through all the unique values of the feature.

The information gain is the reduction in entropy achieved by splitting on this feature.

*[The function starts by calculating the total entropy of the dataset.*

*It then iterates over the unique values of the feature and calculates the entropy of each subset of data where the feature takes a particular value.*

*The weighted entropy of each subset is subtracted from the initial entropy to calculate the information gain.*

*The function returns the total information gain, which tells us how much the feature reduces uncertainty about the target variable].*

def ID3(dataset, features):

root = Node()

max\_gain = 0

max\_feature = ""

for feature in features:

gain = info\_gain(dataset, feature)

if gain > max\_gain:

max\_gain = gain

max\_feature = feature

root.value = max\_feature

at = np.unique(dataset[max\_feature])

for a in at:

subdata = dataset[dataset[max\_feature] == a]

if entropy(subdata) == 0.0:

newNode = Node()

newNode.isLeaf = True

newNode.value = a

newNode.pred = np.unique(subdata["Play Tennis"])

root.children.append(newNode)

else:

dummyNode = Node()

dummyNode.value = a

new\_attrs = features.copy()

new\_attrs.remove(max\_feature)

child = ID3(subdata, new\_attrs)

dummyNode.children.append(child)

root.children.append(dummyNode)

return root

*The ID3 function builds the decision tree recursively.*

*It selects the feature with the highest information gain as the root.*

*For each unique value of the chosen feature, the dataset is split into subsets.*

*If the subset has no entropy (all samples are of one class), a leaf node is created.*

*Otherwise, the process is repeated recursively on the subset with remaining features.*

1. ***def ID3(dataset, features):***

*Defines a recursive function ID3 that builds a decision tree based on the input dataset and a list of features.*

* ***Parameters****:*

*dataset: The dataset on which the decision tree is being constructed.*

*features: A list of features (attributes) that are considered for splitting at each node.*

***2. root = Node()***

*Creates a new Node object that represents the root node (or a node in the tree). Initially, this node is empty, and its properties (like value and children) will be set later.*

***3. max\_gain = 0***

*Initializes a variable max\_gain to store the maximum information gain observed among all features. It starts at 0 since we are looking for the highest gain.*

***4. max\_feature = ""***

*Initializes an empty string max\_feature to store the name of the feature with the highest information gain. This will be used to split the data at the current node.*

***5. for feature in features:***

*Loops through each feature in the list features to evaluate how good it is for splitting the dataset.*

*To find the feature that gives the maximum information gain at this point in the tree.*

***6. gain = info\_gain(dataset, feature)***

*Calls the info\_gain() function (defined earlier) to calculate the information gain for the current feature on the given dataset.*

*This helps determine how useful each feature is for splitting the data at this point in the tree.*

***7. if gain > max\_gain:***

*Compares the calculated gain for the current feature with the max\_gain found so far.*

*If the current feature gives a higher information gain than any previous feature, it will be chosen for splitting.*

***8. max\_gain = gain***

*Updates max\_gain to the new highest information gain if the current feature’s gain is greater than the previous max\_gain.*

***9. max\_feature = feature***

*Updates max\_feature to the name of the feature that provided the highest information gain so far.*

***10. root.value = max\_feature***

*Sets the value of the root node to be the max\_feature (the feature with the highest information gain). This feature will be used to split the data at this node.*

***11. at = np.unique(dataset[max\_feature])***

*Gets all the unique values (or categories) for the feature max\_feature using np.unique().*

*The decision tree splits the dataset based on each unique value of the selected feature.*

***for a in at:***

*Loops through each unique value a of the feature max\_feature. For each value, the dataset will be split into a subset containing only the rows where the feature equals a.*

***13. subdata = dataset[dataset[max\_feature] == a]***

*Creates a subset of the dataset where max\_feature equals a. This subset will be further evaluated to see if it is pure (i.e., all rows have the same target label) or if it needs further splitting.*

***14. if entropy(subdata) == 0.0:***

*Checks if the subset subdata has an entropy of 0. This means all the rows in subdata belong to the same class (i.e., it's "pure").*

*If the entropy is 0, this node becomes a leaf, meaning no further splitting is needed.*

***15. newNode = Node()***

*Creates a new node newNode that will represent a leaf in the decision tree.*

***16. newNode.isLeaf = True***

*Marks this newNode as a leaf node by setting isLeaf to True. A leaf node indicates that no further splitting is needed.*

***17. newNode.value = a***

*Sets the value of the newNode to the current value a of max\_feature. This represents the branch (or decision) in the tree.*

***18. newNode.pred = np.unique(subdata["Play Tennis"])***

*Sets the pred attribute (prediction) of the leaf node to the class label (either "Yes" or "No"). Since the subset subdata is pure (entropy = 0), all rows will have the same class label.*

***19. root.children.append(newNode)***

*Adds this leaf node newNode to the list of children of the current root node. This indicates that newNode is a child of root.*

***20. else:***

*If the entropy of the subset subdata is not 0, it means the subset contains mixed classes and requires further splitting. In this case, the decision tree needs to grow further.*

***21. dummyNode = Node()***

*Creates a dummyNode that will act as an intermediate node representing the decision at this level.*

***22. dummyNode.value = a***

*Sets the value of the dummyNode to the current value a of max\_feature. This represents a branch in the decision tree for this value of the feature.*

***23. new\_attrs = features.copy()***

*Creates a copy of the features list (so we can safely modify it without affecting the original).*

***24. new\_attrs.remove(max\_feature)***

*Removes the max\_feature from new\_attrs because we don't want to split on the same feature again at lower levels of the tree.*

***25. child = ID3(subdata, new\_attrs)***

*Recursively calls the ID3() function on the subset subdata with the remaining features (new\_attrs). This will generate a subtree rooted at the dummyNode.*

*Recursion allows the tree to keep growing until all subsets are pure (or no features are left to split on).*

***26. dummyNode.children.append(child)***

*Appends the newly created child node (which is the subtree returned by the recursive call) to the dummyNode's list of children.*

***27. root.children.append(dummyNode)***

*Adds dummyNode as a child of the current root node, representing the branch for this value a of the max\_feature.*

***28. return root***

*Returns the root node (which represents either the root of the whole tree or a subtree).*

*This allows the decision tree to be built recursively, with each call constructing a part of the tree and returning it to its parent.*

* *The function recursively builds a decision tree using the ID3 algorithm.*
* *It chooses the feature with the highest information gain, splits the dataset based on this feature, and recursively builds subtrees for each subset.*
* *The function stops when it finds pure subsets (entropy = 0) and creates leaf nodes with predictions for the target variable.*

def printTree(root: Node, depth=0):

for i in range(depth):

print("\t", end="")

print(root.value, end="")

if root.isLeaf:

print(" -> ", root.pred)

print()

for child in root.children:

printTree(child, depth + 1)

*This function recursively prints the decision tree.*

*It indents the tree structure based on depth.*

*If the node is a leaf, it prints the predicted value (e.g., "Yes" or "No").*

**1. def printTree(root: Node, depth=0):**

Defines the printTree() function to print the structure of the decision tree recursively.

* **Parameters**:

root: A Node object representing the current node (root or subtree root).

depth: An optional parameter (default is 0), representing the current depth of the node in the tree. It is used to manage indentation and visually represent the tree hierarchy.

**2. for i in range(depth):**

Loops depth number of times.

This loop controls the indentation of the tree visualization, making it easier to see the structure. The deeper the node is in the tree, the more indented it will be.

**3. print("\t", end="")**

Prints a tab character (\t) for each iteration of the loop, without adding a new line (because of end="").

This indents the output based on the depth of the current node, visually showing which level of the tree the node belongs to.

**4. print(root.value, end="")**

Prints the value of the current node (root.value), which is the feature being used to split at this node (or a value at a leaf node). The end="" ensures that the cursor stays on the same line, so additional information can be printed if needed.

This outputs the feature name (or value) at the current node, helping us see the tree structure.

**5. if root.isLeaf:**

Checks if the current node is a leaf node (i.e., no further splits).

Leaf nodes contain the final classification (prediction), so if it is a leaf node, the predicted class will be printed next.

**6. print(" -> ", root.pred)**

If the node is a leaf, it prints " -> " followed by the prediction root.pred. The prediction represents the class label (like "Yes" or "No") assigned to this leaf.

This shows that the leaf node leads to a final decision, and the pred value is the result of that decision.

**7. print()**

Prints a newline, ensuring that the next node (or child nodes) will be printed on a new line.

This makes sure that each node and its children appear on separate lines for clarity.

**8. for child in root.children:**

Loops through all the child nodes of the current root node.

This allows the function to recursively print all the child nodes (i.e., the branches and subtrees) under the current node.

**9. printTree(child, depth + 1)**

Recursively calls the printTree() function for each child node, passing the child node as the new root and increasing the depth by 1.

The recursive call prints the subtree starting from each child. By increasing depth, the function ensures that child nodes are indented more than their parent nodes, which visually represents the hierarchy of the tree.

* The printTree() function recursively traverses and prints the decision tree.
* It uses indentation (\t) to visually represent the depth of each node in the tree.
* For each node, it prints the feature or value, and if it’s a leaf node, it also prints the predicted class (pred).
* The recursion ensures that the entire tree is printed, with each node properly indented according to its depth.

def classify(root: Node, new):

for child in root.children:

if child.value == new[root.value]:

if child.isLeaf:

print("Predicted Label for new example", new, " is:", child.pred)

return

else:

classify(child.children[0], new)

This function classifies a new instance using the decision tree.

It traverses the tree based on the features of the new example.

Once it reaches a leaf node, it prints the predicted label.

**1. def classify(root: Node, new):**

Defines the classify() function.

* **Parameters**:
  + root: A Node object representing the current node in the decision tree.
  + new: A dictionary-like object (e.g., a row of data) representing the new example that we want to classify. The keys are feature names, and the values are the corresponding feature values for the new instance.

**2. for child in root.children:**

Loops through each child node of the current root node.

Each child node represents one possible outcome of the feature split at the current node. We need to check which child node corresponds to the value of the feature in the new example.

**3. if child.value == new[root.value]:**

Compares the value of the child node with the value of the feature in the new example (new[root.value]).

* + root.value: This is the feature on which the current node splits.
  + new[root.value]: This retrieves the value of that feature in the new example.

The decision tree splits based on feature values, so we compare the feature value at the current node with the corresponding value in the new example. If they match, we follow that branch of the tree.

**4. if child.isLeaf:**

Checks if the child node is a leaf node.

A leaf node contains the final classification (i.e., it represents the decision or predicted class). If it's a leaf node, the prediction is made.

**5. print("Predicted Label for new example", new, " is:", child.pred)**

If the child node is a leaf, it prints the predicted class (child.pred) for the new example.

new: The new instance that we are classifying.

child.pred: The predicted class stored in the leaf node.

This provides the classification result for the new example.

**6. return**

Ends the function execution once the prediction is found.

Once we reach a leaf node and make a prediction, there's no need to continue the recursion. The function returns at this point.

**7. else:**

This else clause is executed if the child node is not a leaf node.

If the node is not a leaf, it means we need to go deeper into the tree to make the final decision.

**8. classify(child.children[0], new)**

Recursively calls the classify() function for the first child node of the current child node, passing the new instance as the input.

If the current node is not a leaf, we recursively follow the decision tree by continuing to classify with the child node. This moves further down the tree to continue the classification process until we reach a leaf.

* The classify() function is designed to traverse the decision tree recursively.
* It matches the values of features in the new example with the nodes in the tree to find the correct path.
* Once it reaches a leaf node, it prints the predicted class for the new example.
* The recursion ensures that the function explores the tree until a decision (leaf node) is reached.

dataset = pd.read\_csv("PlayFootball.csv")

print("The DATASET")

print(dataset)

print("----------------------------")

features = [feat for feat in dataset if feat != "PlayFootball"]

root = ID3(dataset, features)

print("Decision Tree is:")

printTree(root)

print("----------------------------")

The main part of the code loads the dataset from a CSV file (PlayFootball.csv).

It selects all features except the target (PlayFootball).

The ID3 function is called to build the decision tree, and printTree displays the tree.

**dataset = pd.read\_csv("PlayFootball.csv")**

Loads the dataset from a CSV file named "PlayFootball.csv" using the pandas library.

* + pd.read\_csv() reads the CSV file and creates a DataFrame object that holds the data.

This step is necessary to load the dataset into memory so that it can be used to build the decision tree.

**2. print("The DATASET")**

Prints the message "The DATASET" to the console.

This is used to indicate that the dataset will be displayed next. It helps users understand what is being printed.

**3. print(dataset)**

Prints the entire dataset to the console.

* + dataset is a pandas DataFrame containing the data from " PlayFootball.csv".

This allows users to see the loaded dataset before the decision tree is constructed. It’s useful for verification and understanding the structure of the data.

**4. print("----------------------------")**

Prints a line of dashes "----------------------------" to the console.

This is used as a separator to make the console output more readable by clearly dividing sections (dataset, decision tree, etc.).

**5. features = [feat for feat in dataset if feat != "** **PlayFootball "]**

Creates a list of all feature (column) names in the dataset, excluding " PlayFootball ".

* + feat for feat in dataset: This is a list comprehension that iterates over all column names (features) in the dataset.
  + if feat != " PlayFootball ": This ensures that the target column, " PlayFootball ", is excluded from the feature list.

The "Play Tennis" column is the target label (i.e., the class we want to predict), so it shouldn’t be treated as a feature in the decision tree. This step identifies all the input features that will be used for decision-making.

**6. root = ID3(dataset, features)**

Calls the ID3() function to build a decision tree using the dataset and the features.

* + ID3(dataset, features): The function takes in the dataset and the list of feature names, constructs a decision tree, and returns the root node of the tree.
  + root: This stores the root node of the decision tree.

This is the core step where the decision tree is built based on the ID3 algorithm. The decision tree will be used for classifying future examples.

**7. print("Decision Tree is:")**

Prints the message "Decision Tree is:" to the console.

This introduces the next section where the decision tree structure will be printed. It informs users that the decision tree will be displayed.

**8. printTree(root)**

Calls the printTree() function to print the structure of the decision tree.

* + printTree(root): The function takes the root node of the decision tree as input and recursively prints the entire tree structure.

After the decision tree is built, this step visually displays its structure for the user. It shows how the data is split at each node, and what the final decision rules are.

**9. print("----------------------------")**

Prints a line of dashes "----------------------------" to the console again.

This acts as a closing separator after the decision tree has been printed, making the output cleaner and easier to read.

* This block of code loads a dataset, builds a decision tree using the ID3 algorithm, and then prints both the dataset and the resulting decision tree structure to the console. The use of separators and messages makes the output more user-friendly and understandable.