CSE5AIF

Artificial Intelligence Fundamentals

Lab 8: Decision Trees

Lab Objectives

The objective of this lab is to implement the ID3 Decision Tree Machine Learning algorithm to predict an output classification based on some given dataset.

By the end of this lab, you should understand how a Decision Tree works including the concept of entropy and partitioning based on attributes. Utilizing the given material, you should be able to deploy a Decision Tree effectively to a wide range of applications.

Background

A Decision tree is a predictive model that uses a tree structure to represent a number of possible decision paths and an outcome for each path. It essentially operates by using decision nodes (or splits) to break down a dataset into successively smaller subsets, this tree structure terminates at a classification or decision known as a leaf node. The top-most node/split in a decision tree corresponds to the best predictor, this is the root node

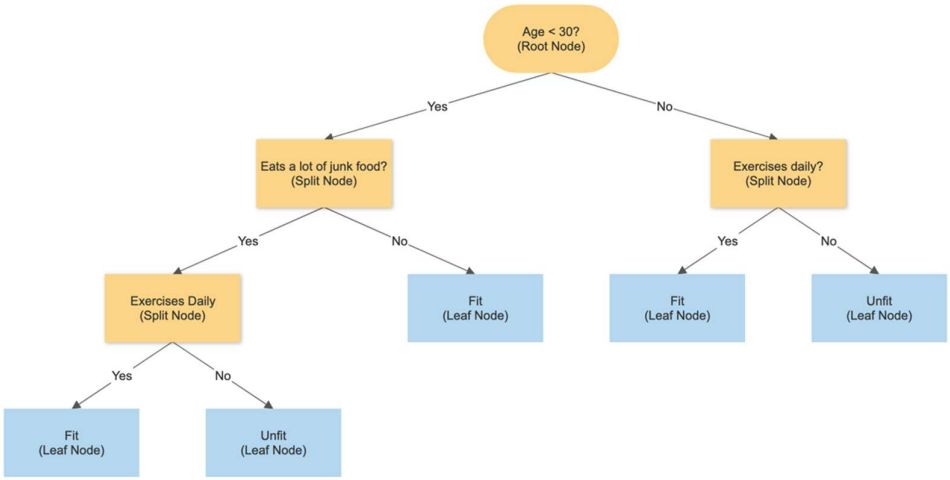


Figure 1. Basic structure of a Decision Tree

Decision trees are useful as they are very easy to understand and interpret, and the entire prediction path is transparent to the user, unlike other machine learning models. They can handle a mix of numerical (e.g. Age) and categorical (e.g. Exercise Daily Boolean) inputs and can even classify data where input attributes are missing.

Finding the optimal decision tree for a set of training data is computationally expensive (as the ID3 algorithm is greedy); to get around this problem, most decision trees are not made to be optimal but to operate well enough for the application, although even this can become problematic once the dataset size increase.

Most decision trees are classed into one of two categories:

- Classification trees – These produce categorical outputs (like in the figure above).
- Regression trees – These provide numerical outputs.

Implementation

This lab will focus on the implementation of a classification decision tree using the ID3 algorithm, as discussed previously during Lecture 10. While the code is generalized and as such can be applied to a variety of data, we will be employing our decision tree to predict if a person is likely to go for a run based on the following input parameters:

- **outlook** (Forecast): Sunny, Overcast, or Rain
- **temp**: Cold, Mild, or Hot
- **humidity**: High, or Normal
- **wind**: Weak, or Strong

Out dataset will also include a **did_run** bool parameter as training data for our tree.

To begin, launch Jupyter Notebook or your preferred IDE and create a new file named **lab_9** in your working directory.

The following code is explained in order of execution/implementation; the final code structure of your file should be:

- data_entropy
- partition_entropy
- partition_by
- partition_entropy_by
- class Leaf
- class Split
- classify
- build_tree
- Main Code

Please ensure you follow this structure as your code may otherwise result in an error.

Imports

At the top of your lab file, import the following:

```
from typing import NamedTuple
from collections import defaultdict, Counter
import math
```

This lab utilizes some new imports that we haven't yet encountered, these are a variety of *type* classes and are used for ease of implementation. They are described as:

NamedTuple:

A named tuple is similar to a regular tuple but allows the user to assign a name to each position; this allows you to increase code readability while also making it easier to reference stored variables.

Counter:

A Counter is a subclass of a dict; it is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values.

The counter class also has some very useful methods, one of these being the **most_common** method; this returns a list of the n most common elements and their counts:

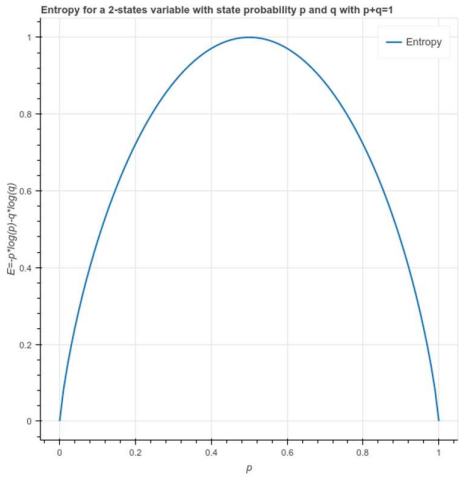
```
# Code   print(Counter('abracadabra').most_common(3))
# Output [('a', 5), ('b', 2), ('r', 2)]
```

More information about all of these imports can be found in the python documentation at: https://docs.python.org/3/library/collections.html#module-collections

Entropy

In order to build a decision tree, we must be able to decide what questions to ask and in what order. At each split node there are some possibilities that are eliminated and some that aren't. Ideally, we want to choose questions whose answers give the most information about the tree prediction (e.g. if there is a single question that a "True" answer always corresponds to a "True" output it would be the best question to ask).

In order to achieve this, we utilize what is known as entropy. In this context, entropy is essentially a measure of information gain for a specific split; mathematically this is defined as:



$$Entropy(S) = -p_1 \log_2 p_1 - \ldots - p_n \log_2 p_n$$

Where, p_i is the proportion of data labelled as class c_i. In this function, each term $(p_n \log_2 p_n)$ is non-negative and is close to 0 when p_i is either close to 0 or close to 1.

When most of the data is in a single class, the entropy will be small as every p_i is close to 0 or 1. Conversely, the entropy will be larger when many of the $p_i's$ are not close to 0 (i.e. when the data is spread across multiple classes.

To compute the entropy of our data, we use the following functions:

```
def data_entropy(labels):
    # Compute the class probabilities
    # Counter(labels).values() stores the value (False/Negative) as dict key and the count as dict value
    total_count = len(labels)
    class_probabilities = []
    for count in Counter(labels).values():
        class_probabilities.append(count / total_count)

    # Given a list of class probabilities, compute the entropy
    entropy = sum(-p * math.log(p, 2) for p in class_probabilities if p > 0)
    return entropy
```

The **data_entropy** function first computes the probability that an input belongs to a class, based on the total inputs passed in through the **labels** parameter. Next it takes these class probabilities and computes the entropy utilizing the equation outlined previously.

This allows us to compute the entropy of a single set of the labelled data. At each stage of the decision tree, we ask a question whose answer partitions data into one or more subsets. As such, we need to find a way to define the entropy that results from partitioning/splitting a set of data in a certain way.

We want a partition to have low entropy if it splits the data into subsets that themselves have low entropy, and high entropy if it contains subsets that have high entropy. Mathematically, we do this by calculating the entropy of the partition as a weighted sum; this is implemented as:

```
def partition_entropy(subsets):
    # Returns the entropy from this partition of data into subsets
    total_count = sum(len(subset) for subset in subsets)

    return sum(data_entropy(subset) * len(subset) / total_count for subset in subsets)
```

Main Code

We will start implementing the actual decision tree by first defining our main function; this will allow you to get an understanding of our input data structure.

Define a class named **Individual** to store all of the input parameters to our system, this will be a subclass of a NamedTuple. In this definition, we add our parameter names as keys and define the data types we will be storing.

Next, we define our training dataset. Typically, this will be a much larger set of data and would be stored in a separate file. Define the dataset *list* named **inputs,** each initializer in this list will be an **Individual** object with parameters matching our previously defined class.

Finally, define a variable named **target_attribute**, this will define the attribute we wish to predict. The **attributes** list, stores all of the dataset attributes we will be passing to the decision tree (note: the target_attribute is not in this list).

The remaining code will be explained at the end of the implementation section.

```
if __name__ == "__main__":
    # Named tuples assign a name to each position in a tuple.
    # They allow the ability to access fields by this name instead of position index.
    class Individual(NamedTuple):
        outlook: str
        temp: str
        humidity: str
        wind: str
        did_run: bool = None
        # We allow "None" here, for when we are predicting this based on the other inputs.

    # Inputs using the Individual class for easy referencing
    #                 outlook     temp    humidity  wind      did_run
    inputs = [Individual('Sunny',    'Hot',  'High',   'Weak',   False),
              Individual('Sunny',    'Hot',  'High',   'Strong', False),
              Individual('Overcast', 'Hot',  'High',   'Weak',   True),
              Individual('Rain',     'Mild', 'High',   'Weak',   True),
              Individual('Rain',     'Cool', 'Normal', 'Weak',   True),
              Individual('Rain',     'Cool', 'Normal', 'Strong', False),
              Individual('Overcast', 'Cool', 'Normal', 'Strong', True),
              Individual('Sunny',    'Mild', 'High',   'Weak',   False),
              Individual('Sunny',    'Cool', 'Normal', 'Weak',   True),
              Individual('Rain',     'Mild', 'Normal', 'Weak',   True),
              Individual('Sunny',    'Mild', 'Normal', 'Strong', True),
              Individual('Overcast', 'Mild', 'High',   'Strong', True),
              Individual('Overcast', 'Hot',  'Normal', 'Weak',   True),
              Individual('Rain',     'Mild', 'High',   'Strong', False)
              ]

    # Generate the decision tree based on our inputs data.
```

```
# We pass did_run as our target, this is what we want to predict.
target_attribute = 'did_run'
attributes = ['outlook', 'temp', 'humidity', 'wind']

tree = build_tree(inputs, attributes, target_attribute)

# Test the tree on new data.
# This should predict True,
print(classify(tree, Individual("Sunny", "Mild", "Normal", "Weak")))

# and this should predict False.
print(classify(tree, Individual("Rain", "Cold", "High", "Strong")))
```

Decision Tree

Our decision tree will consist of decision nodes (splits), that ask a question and direct us depending on the answer, and leaf nodes, terminal nodes that give us a prediction. The ID3 algorithm we will be using for implementation operates in the following method:

1. If the data all have the same label, create a leaf node that predicts that label and then stop.
2. If the list of attributes is empty (i.e. there are no possible questions left), create a leaf node that predicts the most common label and then stop.
3. Else, partition the data by each of the input attributes.
4. Choose the partition with the lowest entropy.
5. Add a decision/split node based on the chosen attribute.
6. Run 1-5 recursively on each partitioned subset using all remaining attributes.

First, we will define the partitioning method:

```
def partition_by(inputs, attribute):
    # Partitions the inputs into two dictionary lists,
    # one set containing the individuals with the attribute as False
    # and one containing the individuals with the attribute as True
    partitions = defaultdict(list)
    # For each Individual in out inputs list
    for input in inputs:
        # get the value of the attribute for this Individual
        attribute_value = getattr(input, attribute)
        # Append the input to the partitions list based on the attribute value
        partitions[attribute_value].append(input)
```

```
    # The key of this partitions dict is True or False,
    # i.e. the True set contains all individuals whose attribute is True
    return partitions
```

This takes the list of inputs and returns a partitioned set based on the attribute parameter. Next, define a function that computes the entropy corresponding to the input partition, utilizing the **partition_entropy** function we previously defined:

```
def partition_entropy_by(inputs, attribute, label_attribute):
    # Calculates the entropy of a given partition
    # First, partition the inputs by the attribute we want to calculate entropy for
    partitions = partition_by(inputs, attribute)

    labels = []
    # partitions.values() gets the set of individuals corresponding to the True/False dict keys
    # For each partition in the partitions dict,
    for i, partition in enumerate(partitions.values()):
        labels.append([])
        # For each Individual in this partition,
        for input in partition:
            # append the value of label_attribute for this Individual to the labels list at the same partition index
            labels[i].append(getattr(input, label_attribute))

    # Return the entropy value for the partition
    return partition_entropy(labels)
```

Once we have these functions, all that we would need to do to select a decision/split for our decision tree is to find which partitioning attribute has the minimum-entropy partition over the whole dataset (i.e. which attribute would allow us to best predict the outcome given the input data).

To implement our tree, we must define how we will represent it. We will do so utilizing two classes **Leaf**, and **Split**, both of which will be NamedTuples:

```
class Leaf(NamedTuple):
    # A Leaf is a prediction, i.e. Did Run or Didn't Run
    value: None


class Split(NamedTuple):
    # A Split contains an attribute to split on,
    # the subtrees for specific values of that attribute,
    # and a default value if we see and unknown value
    attribute: str
```

```
    subtrees: dict
    default_value: None
```

Given these representations, we can classify our input utilizing:

```
def classify(tree, input):
    # Classify the input using the given decision tree

    # If this is a leaf node, return its value loop is done
    if isinstance(tree, Leaf):
        return tree.value

    # Otherwise this tree consists of an attribute to split on
    # and a dictionary whose keys are values of that attribute
    # and whose values are subtrees to consider next
    subtree_key = getattr(input, tree.attribute)

    if subtree_key not in tree.subtrees:   # If no subtree for key,
        return tree.default_value          # return the default value.

    subtree = tree.subtrees[subtree_key]   # Choose the appropriate subtree
    return classify(subtree, input)        # and use it to classify the input.
```

This **classify** function provide our prediction based on the input **Individual** and the decision tree.

All that is left to do it to build the decision tree from our training data:

```
def build_tree(inputs, split_attributes, target_attribute):
    # Count target labels, these are stores in label_counts under the value key
    # (i.e. True or False)
    label_counts = Counter(getattr(input, target_attribute)
                           for input in inputs)

    # Find most common label
    # most_common(number of values to return; i.e. most common 1 or 3 etc.)
    most_common_label = label_counts.most_common(1)[0][0]

    # If there is only one label in label_counts,
    # this ends the loop and we return the prediction
    if len(label_counts) == 1:
        return Leaf(most_common_label)
```

```
    # If there are no more splits available in our decision tree,
    # We return the prediction for the majority label
    if not split_attributes:
        return Leaf(most_common_label)

    # Otherwise we split by the best attribute

    def split_entropy(attribute):
        # Helper function for finding the best attribute
        return partition_entropy_by(inputs, attribute, target_attribute)

    # Find the best_attribute to split by; i.e. the attribute with the lowest entropy
    # This "min" function will return the minimum attribute in the split_attributes list
    # based on the return value of split_entropy for each attribute.
    best_attribute = min(split_attributes, key=split_entropy)

    # Partition based on the best_attribute
    partitions = partition_by(inputs, best_attribute)
    # Create a new list of attributes to pass through in the recursive call
    # (removing the best_attribute we just split on)
    new_attributes = [a for a in split_attributes if a != best_attribute]

    # Create the subtrees for this split
    # Subtrees are stored using the attribute_value as the key in the subtrees dict
    # The subtree stored under each key is created by recursively calling this same function,
    # passing the subset of input values stored in each partition of our partitions set from above.
    subtrees = {attribute_value: build_tree(subset, new_attributes, target_attribute)
                for attribute_value, subset in partitions.items()}

    # For each recursive call of this function, we continue down the tree until we hit a leaf node
    # at which point one of the previous return conditions are met and a leaf prediction is returned.
    # Once the recursive call to populate subtrees above is completed, we return the parent split of these
subtrees.
    return Split(best_attribute, subtrees, default_value=most_common_label)
```

Referring back to the main block of code above; you can see that we utilize **build_tree** to generate our **tree** structure based on the input individuals and attributes.

This code also has two print statements containing calls to the classify function. These calls utilize the generated **tree** along with a newly defined test **Individual** to obtain a prediction/classification.

This concludes the implementation of our Decision Tree. Ensure all of your code is in the correct order as outlined previously, then run your file.

The current configuration of the code will output two predictions based on the test individuals.

Extension

1. To extend on your implementation, you may wish to alter the application; all that this requires is modification of the **inputs** and **attributes** in the main code block.

2. The ID3 decision tree algorithm is a greedy algorithm that is not optimal, you may want to try you understanding of the concepts outlined in the lab by implementing the C4.5 or C5 algorithms