Bonus exam problem.

Introduction

This notebook demonstrates our implementation of the decision tree algorithm. We will start by looking at the implementation of the algorithm through the Python code. Afterwards, an example dataset will be provided. Finally, the result of the dataset analysis will be shown and discussed using a plot. In the sources at the end of the notebook, the used AI tools will be named as well as a listing of which parts of the code and notebook are influenced by it.

This problem was solved by the group consisting of:

Tobias Bosl | 22203101

Tjorven Burdorf |

Bastian Höschl |

Mathias Kirschner |

Problem description

The objective was to create an algorithm to derive a classification decision tree based on input data. Additionally, the task required a visual representation of the tree.

Implementation

In the following, the created code will be shown and explained.

Imports

This code shows the required imports.

import math

from random import Random

import csv

import tkinter as tk

Project structure

The code for the decision tree is split into eight files. Each one will be discussed in the following. The files can be divided into five categories:

Data Modelling:

data\_model: Represents the input data stored as a list of items, each containing attributes and classification.

node: Defines a node in the decision tree.

Data Input:

table\_reader: Reads CSV files and provides a format parsed by the data\_model for creating the dataset.

Plot:

tree\_plot: Utilizes tkinter GUI to plot the decision tree root and its children recursively.

Tree Algorithm:

decision\_tree: Implements the decision tree algorithm, creating and organizing nodes.

information\_gain\_calculations: Provides methods to calculate the most important attribute based on the course slides' calculations.

util: Offers helper methods to retrieve unique attribute values and check if remaining examples share the same classification.

Execution

execution: Manages the main program, including dataset initialization, algorithm application, tree sorting, and plotting.

Data Modelling

The modelling of the input data is done within the data\_model. The representation for tree nodes is realized within node.

data\_model

# The possible result values of a positive classification.

positive\_results = [True, 1, "T", "True", "t", "true", "TRUE", "yes", "Yes", "YES", "y", "Y", "1"]

class DataItem:

"""A class representing a data item."""

def \_\_init\_\_(self, classification, attributes):

"""Initialize a new data item using the classification provided and its associated attributes.

Attributes is a dictionary of attribute names to the value in the item"""

self.classification = classification

self.attributes = attributes

class Dataset:

"""A class representing a dataset."""

def \_\_init\_\_(self, filepath, decision):

"""Initializes the dataset with the by reading the table at filepath.

The decision is the name of the classification attribute."""

self.items = [DataItem(classification=item[0], attributes=item[1]) for item in

read\_csv\_table(filepath, decision)]

self.decision = decision

self.positive = None

self.negative = None

def get\_attributes(self):

"""Returns the attributes of the dataset."""

return [key for key in self.items[0].attributes]

def positive\_classification(self):

"""The dataset's positive classification representation."""

if self.positive:

return self.positive

for i in self.items:

if i.classification in positive\_results:

self.positive = i.classification

return self.positive

def negative\_classification(self):

"""The dataset's negative classification representation."""

if self.negative:

return self.negative

for i in self.items:

if i.classification not in positive\_results:

self.negative = i.classification

return self.negative

node

class Node:

"""A node in a decision tree."""

def \_\_init\_\_(self, data, reason):

"""Initializes a node with the given data and reason."""

self.children = []

self.data = data

self.reason = reason

def insert(self, subtree):

"""Adds the provided subtree to the children list of the node."""

self.children.append(subtree)

@staticmethod

def \_\_sort\_by\_keys(atc):

"""Returns a sorted dictionary based on the keys of the given dictionary."""

return {k: atc[k] for k in sorted((atc.keys()))}

def \_\_value\_from\_reason(self):

"""The value of the most important attribute of

the parent used by the root of this subtree."""

return self.reason[self.reason.index(":") + 1:].strip()

def sort(self):

"""Sorts the children of this node."""

values\_to\_children = {c.\_\_value\_from\_reason(): c for c in self.children}

ordered\_values = self.\_\_sort\_by\_keys(values\_to\_children)

self.children = list(ordered\_values.values())

for child in self.children:

child.sort()

Data Input

the table\_reader enables the extraction and formatting of training data from CSV files, preparing it for further processing within the decision tree algorithm.

table\_reader

def attributes\_from\_row(items, classification\_attribute):

"""Returns a dictionary of attributes from a row, excluding the classification attribute."""

return {attribute: value for (attribute, value) in items if attribute != classification\_attribute}

def read\_csv\_table(file\_path, decision\_attribute):

"""Returns a list of tuples with the classification attribute

and a dictionary of attributes from the rows in the file."""

with open(file\_path, mode='r', newline='') as file:

return [(row[decision\_attribute], attributes\_from\_row(row.items(), decision\_attribute)) for row in

csv.DictReader(file) if isinstance(row, dict)]

Plot

The tree\_plot module is responsible for visualizing decision trees. It employs a recursive approach to add nodes, starting from the root, along with their respective children. The plot is shown in a GUI by tkinter.

tree\_plot

def calculate\_maximum\_offset\_for\_both\_sides(level, original):

"""The function will provide the maximum offset of both tree sides

based on the current level and the original offset."""

if level == 0:

return original / (level + 0.8)

else:

return original / (level + 1.5)

def calculate\_new\_x(i, subtrees, parent\_x, level):

"""Use this function to determine the currently selected

child's x position based on the index, the number of

subtrees, the parent x coordinate and the level."""

if subtrees % 2 == 1 and subtrees // 2 == i:

return parent\_x

children\_per\_side = subtrees // 2

offset = calculate\_maximum\_offset\_for\_both\_sides(level, 450)

if i < subtrees // 2:

numerator = children\_per\_side - i

scale = numerator / children\_per\_side

return parent\_x - (scale \* offset)

else:

index\_of\_child\_in\_current\_tree\_half = i + 1 if subtrees % 2 == 0 else i

numerator = index\_of\_child\_in\_current\_tree\_half - children\_per\_side

scale = numerator / children\_per\_side

return parent\_x + (scale \* offset)

class TreePlot:

"""A class for plotting decision trees."""

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

"""Initializes the plotter."""

self.window = tk.Tk()

self.window.title("Plot")

self.window.geometry("1800x900")

self.canvas = tk.Canvas(self.window, width=1800, height=900, bg="white")

self.texts = []

self.canvas.pack()

def \_\_add\_rectangle(self, x1, y1, x2, y2):

"""Adds a rectangle to the canvas."""

self.canvas.create\_rectangle(x1, y1, x2, y2, fill="white", width=0)

def \_\_add\_text(self, x, y, content):

"""Adds text to the canvas."""

self.texts.append(self.canvas.create\_text(x, y,

text=content,

fill="black", font=("Helvetica", 12)))

def add\_reason\_text(self, position\_x, x, position\_y, y, child):

"""Adds the reason text to the canvas."""

x\_begin = (position\_x + x) / 2

y\_begin = (position\_y + y) / 2

self.\_\_add\_rectangle(x\_begin - 30, y\_begin - 10, x\_begin + 50, y\_begin + 10)

self.\_\_add\_text(x\_begin, y\_begin, f"{child.reason[child.reason.index(':') + 1:].strip()}")

def texts\_to\_front(self, texts):

"""Brings the texts to the front."""

for t in texts:

self.canvas.tag\_raise(t)

def \_\_root\_jobs(self, x, goal):

"""Does the jobs only the root item should do once."""

self.canvas.create\_text(x, 20, text=f"Goal: {goal}", fill="black", font=("Helvetica", 18))

self.texts\_to\_front(self.texts)

self.window.mainloop()

def \_\_plot\_node(self, x, y, tree):

"""Plots the node."""

self.canvas.create\_rectangle(x - 50, y - 10, x + 50, y + 10, fill="white", width=0)

self.\_\_add\_text(x, y, f"{tree.data}")

def plot(self, tree: Node, x=900, y=50, level=0, goal=""):

"""Plots the tree."""

for i in range(len(tree.children)):

position\_x = calculate\_new\_x(i, len(tree.children), x, level)

position\_y = y + int((200 / (level + 1)))

self.canvas.create\_line(x, y, position\_x, position\_y, fill="blue", width=2)

self.add\_reason\_text(position\_x, x, position\_y, y, tree.children[i])

self.plot(tree.children[i], position\_x, position\_y, level + 1)

self.\_\_plot\_node(x, y, tree)

if level == 0:

self.\_\_root\_jobs(x, goal)

Tree Algorithm

The Tree Algorithm section cares about the decision tree construction process.

util:

Contains commonly used methods for filtering data, like filtering the unique values of an attribute and checking if all examples have the same classification.

information\_gain\_calculations:

This file contains functions to calculate the information gain and relevant sub-calculations based on the formulas provided on the courses slides.

decision\_tree:

In this section the main logic for the construction of the decision tree can be found. It includes the functions plurality\_val for determining the most common classification and dt\_learning for recursively building the tree based on examples and attributes.

util

def unique\_values(A, examples):

"""Creates the set of unique values of a selected attribute

in the example data set of the attribute A in the examples."""

return set([e.attributes[A] for e in examples])

def check\_if\_all\_examples\_have\_same\_classification(examples):

"""Returns True if all examples have the same classification, False otherwise."""

if not examples:

return True

return len([e for e in examples if e.classification != examples[0].classification]) == 0

information\_gain\_calculations

def ratio\_of\_positive\_classification\_for\_attribute\_value(attribute, examples, value):

"""Calculates the amount of positive classifications of the chosen value compared

to the amount for the given attribute value."""

matches = [e for e in examples if e.attributes[attribute] == value]

positives = len([m for m in matches if m.classification in positive\_results])

return positives / len(matches)

def entropy(attribute, examples, value):

"""Uses the formula for the calculation of the entropy to calculate it for the given attribute value."""

goal\_ratio = ratio\_of\_positive\_classification\_for\_attribute\_value(attribute, examples, value)

return b(goal\_ratio)

def cardinality\_ratio(A, a, examples):

"""Returns the ratio of the cardinality of the subset of examples where the attribute A

has the value a compared to the cardinality of the set of examples."""

example\_subset = [e for e in examples if e.attributes[A] == a]

return len(set(example\_subset)) / len(set(examples))

def remainder(A, examples):

"""Determines the entropy expected to be left after testing the attribute A."""

return sum([cardinality\_ratio(A, a, examples) \* entropy(A, examples, a) for a in unique\_values(A, examples)])

def positive\_goal\_possibility(examples):

"""Finds the amount of positive classifications in the example dataset

compared to the overall data items in the set."""

count\_pos = len([e for e in examples if e.classification in positive\_results])

return count\_pos / len(examples)

def b(positive\_goal\_ratio):

"""Returns the entropy of the given ratio of positive classifications."""

if positive\_goal\_ratio in [0, 1]:

return 0

return -(positive\_goal\_ratio \* math.log2(positive\_goal\_ratio) + (1 - positive\_goal\_ratio) \* math.log2(

1 - positive\_goal\_ratio))

def information\_gain(A, examples):

"""Calculates the impact of the classification caused by A."""

return b(positive\_goal\_possibility(examples)) - remainder(A, examples)

def most\_important\_attribute(examples, attributes):

"""Returns the attribute with the highest information gain and its information gain."""

max\_gain = max((information\_gain(attribute, examples), attribute) for attribute in

attributes) # max returns tuple where first element is max

return [max\_gain[1], max\_gain[0]]

decision\_tree

def plurality\_val(examples):

"""Returns the most common classification in the examples. If there is a tie, a random decision is made."""

true\_count = len([e for e in examples if e.classification in positive\_results])

if true\_count == len(examples) / 2:

decision = Random().randint(0, 1)

else:

decision = ds.positive\_classification() if true\_count > len(examples) / 2 else ds.negative\_classification()

return Node(decision, "")

def dt\_learning(examples: list, attributes, parent\_examples, start=False):

"""Returns a decision tree based on the examples and attributes."""

if not examples:

return plurality\_val(parent\_examples)

elif check\_if\_all\_examples\_have\_same\_classification(examples):

return Node(examples[0].classification, "")

elif not attributes:

return plurality\_val(examples)

else:

A = most\_important\_attribute(examples, attributes)[0]

reason = A if not start else "root"

my\_tree = Node(A, reason)

for v in unique\_values(A, examples):

exs = [e for e in examples if e.attributes[A] == v]

subtree = dt\_learning(exs, [a for a in attributes if a != A], examples)

if subtree:

subtree.reason = f"{A}: {v}"

my\_tree.insert(subtree)

return my\_tree

Execution

The final snippet demonstrates the execution of the decision tree algorithm. It initializes a dataset from a CSV file, applies the decision tree learning algorithm, sorts the resulting tree, and plots it using the TreePlot class.

ds = Dataset(filepath="tables/example.csv", decision="PlayTennis")

tree = dt\_learning(ds.items, ds.get\_attributes(), ds.items, True)

tree.sort()

TreePlot().plot(tree, goal=ds.decision)

Data Set

In the following, the used dataset will be shown and explained.

Table from

datatable.png

Explanation

Outlook: The weather expected.

Temp: The temperature level.

Humidity: The humidity level in the air.

Wind: The strenght of wind.

PlayTennis (Classification): Determines whether you should play tennis.

Result

The following image shows the plotted decision tree.

result.png

Result explanation and discussion:

The image shows a typical decision tree. Each node is filled with the most important attribute at the current level. The branches lead to the children or classification results based on the value of the attribute which is shown in the middle of the line.

Considerng the example data, the tree determines that you can play tennis if:

The outlook is overcast.

It's sunny wiht normal humidity.

It's raining with a weak wind.

According to the result you should avoid playing tennis if:

It's sunny with high humidity.

It's raining with a strong wind.

Notably, the temperature attribute is not taken into account in any of the nodes. Taking a deeper look into the dataset, it becomes obvious that the classification of each element remains unchanged regardless of temperature variations. Therefore, there missing of the tempature attribute is not a problem. However, changing the classification of dataset item (e.g., item 2 from 'No' to 'Yes') would lead to a different looking decision tree which includes the temperature attribute, as visualized in the image below.

result2.png

Sources

AI tools

During the creation of the code and this notebook, the following AI tools have been used:

ChatGPT

GitHub Copilot

Usage

The named AI tools were mostly used for refactoring our developed code to improve readability and to make use of Python-specific features like list comprehensions "which allowed the creation of lists or filtered sublists with less extensive code. They have also been used for creating docstrings to help explain the key features of the provided code. Additionally, we used ChatGPT to improve the grammar and coherence of the report provided in this notebook. The development of the algorithm was started from scratch by us. The named A. I. tools came into use after we implemented a working version of the algorithm.

Dataset

The dataset used for the testing and visualization of the algorithm has been taken from the Video "1. Decision Tree |ID3 Algorithm | Solved Numerical Example | by Mahesh Huddar" which can be found at the channel "Mahesh Huddar" on this page: https://www.youtube.com/watch?v=coOTEc-0OGw