**DECISION TREE INDUCTION**

Rubén Ventura

9305420195

**CONTENT**

[**SUMMARY OF THE PROJECT** 3](#_Toc39002830)

[**DESCRIPTION OF THE CODE** 4](#_Toc39002831)

[class Leaf: 4](#_Toc39002832)

[class Node: 4](#_Toc39002833)

[class Attribute: 4](#_Toc39002834)

[read\_file(file\_name): 5](#_Toc39002835)

[write\_file(file\_name, labels, data, result): 5](#_Toc39002836)

[get\_unique\_values(data, column): 6](#_Toc39002837)

[count\_values(data): 6](#_Toc39002838)

[build\_decision\_tree(data): 6](#_Toc39002839)

[best\_split(data): 7](#_Toc39002840)

[divide(data, attribute): 8](#_Toc39002841)

[information\_gain(right\_rows, left\_rows, accumulated\_uncertainty): 8](#_Toc39002842)

[gini(data): 9](#_Toc39002843)

[classify(data, node): 9](#_Toc39002844)

[main(): 9](#_Toc39002845)

[**INSTRUCTIONS FOR COMPILING** 10](#_Toc39002846)

[**ADDITIONAL SPECIFICATIONS** 10](#_Toc39002847)

# **SUMMARY OF THE PROJECT**

This time we had to implement a decision tree, which given an amount of data would be able to classify (to a certain reliability) similar data.

This project was more challenging than the last one and I found it a lot more complex since fixing and improving the code (especially when something went wrong or the results were not as expected) could not be done as easily, and using the debugger to keep track of the values and the data flux was key for me finishing the project.

Once again, I decided to use Python for this project in order to improve my skills. I also tried to split my code so that it is as self-explanatory as possible, refactoring the functions as much as I could and naming them so that what they do is clear enough and it is easy to follow the algorithm steps.

For the algorithm implementation I followed the pseudo-code in the textbook, and the splitting method I chose was the Gini Index (or Gini Impurity).

# **DESCRIPTION OF THE CODE**

## class Leaf:

# Node that holds a dictionary with a label and the number of

# times it appears in the dataset

class Leaf:

    def \_\_init\_\_(self, data):

        self.predictions = count\_values(data)

This class is used to store the information of the data in the leaves of the tree. The attribute “predictions” contains a dictionary with the pair label:count.

## class Node:

# Decision node. Stores the attribute used for the partition and

# has references to the best partition branch (bpb) and the other branch (leftovers)

class Node:

    def \_\_init\_\_(self, attribute, bpb, leftovers):

        self.attribute = attribute

        self.best\_partition\_branch = bpb

        self.other\_branch = leftovers

This class is used to store the information on the split nodes of the tree. It stores the attribute used for the split and points to the two branches.

## class Attribute:

# Stores the column number for each attribute and the label

class Attribute:

    def \_\_init\_\_(self, column, value):

        self.column = column

        self.value = value

    def compare(self, target):

        return target[self.column] == self.value

This class stores the attribute information: the column position of the attribute and the value (name). I also included a little function which I use to check if one object matches this attribute (has the same value). As a clarification: I only took the value match as true when comparing if the target has exactly the same value, since in the instructions we were told all the attributes are categorical (so numeric values are treated the same way).

## read\_file(file\_name):

# Get header first, then read data and store in a list

def read\_file(input\_file\_name):

    attributes = []

    data = []

    with open(input\_file\_name, 'r') as f:

        attributes = f.readline().split()

        for line in f.read().split('\n'):

            if line != '':

                data.append(line.split('\t'))

    return attributes, data

Given an input file, we read the first line and store its content (it is expected to be the header of the database, containing the list of labels). After that, we read each line and store each value in that line in a list. When done, returns both the list of labels and data (a list of lists with all the values in the file).

## write\_file(file\_name, labels, data, result):

def write\_file(output\_file\_name, labels, test\_data, results):

    with open(output\_file\_name, 'w') as f:

        # Write labels

        for column in range(len(labels)):

            f.write(labels[column] + "\t")

        f.write("\n")

        # Write results

        for row in range(len(test\_data)):

            test\_data[row].append(results[row])

            for column in range(len(test\_data[row])-1):

                f.write(test\_data[row][column] + "\t")

            f.write(test\_data[row][column +1] + "\n")

Once we are finished with the main algorithm, we will have the labels and the test data we read from the input file stored in two variables, and the classifying function will also give us a list of values which will be stored as results. Given all these variables, we proceed to write the output file following the format specified in the instructions: we first write the header, each label separated by a tabulation; after that, we add the results we got for each row to the original data, and then write each value for that row in the document (separated by a tabulation).

## get\_unique\_values(data, column):

# Select all the different attribues in a specific column (to get)

def get\_unique\_values(data, attribute\_column):

    return set([different\_values[attribute\_column] for different\_values in data])

This one is a little helper function I decided to separate from the main one for clarity. Basically, given a database in the form of list of rows (lists) with values, it iterates through all the values in a specified column and stores each different value found (using one property of the set structure type).

## count\_values(data):

# Count the number of each attribute in the given data

def count\_values(data):

    count = {}

    for row in data:

        # the attribute we want to track is in the last position always

        label = row[-1]

        if label not in count:

            count[label] = 0

        count[label] += 1

    return count

This is another one of the little helper functions. Given some data in the form of a list of rows (lists) with different values, we store the pairs of value:count in a dictionary and return it.

## build\_decision\_tree(data):

#Build the tree recursively

def build\_decision\_tree(data):

    # 1) Find best split - Calculate information gain for each attribute

    gain, attribute =  best\_split(data)

    # 2) Divide into branches

    # 2.1) If no information gained, return Leaf

    if gain == 0:

        return Leaf(data)

    # 2.2) Else, we divide the data

    best\_partition\_rows, other\_rows = divide(data, attribute)

    # 3) Build branches recursively

    best\_partition\_branch = build\_decision\_tree(best\_partition\_rows)

    other\_branch = build\_decision\_tree(other\_rows)

    # 4) Return node with the branches

    return Node(attribute, best\_partition\_branch, other\_branch)

This is the main function of this project. I tried to follow the pseudo-code written in the textbook. Given the data, we first try to find the best attribute to split it (and store the information gained with the operation) and then check if we get no information from splitting it, in which case we will classify it as a leaf and return the Leaf object initialized with the data. If splitting the data is possible and algo grants us some information, we proceed to partition the database using the attribute we got from the previous step. Once we get the two branches (partitions), we proceed to repeat this process on each of them, recursively, to build the tree. The first call of the function returns the root node created with the data and pointing to its two branches.

## best\_split(data):

# Find best split trying each attribute

def best\_split(data):

    best\_gain = 0

    best\_attribute = None

    node\_uncertainty = gini(data)

    n\_cols = len(data[0]) - 1

    for column in range(n\_cols):

        values = set([value[column] for value in data])

        for value in values:

            attribute = Attribute(column, value)

            # Attemp to partition data

            best\_split\_rows, other\_rows = divide(data, attribute)

            # Check if it actually divided the data, else go next

            if len(best\_split\_rows) == 0 or len(other\_rows) == 0:

                continue

            # Calculate information gain with this split

            gain = information\_gain(best\_split\_rows, other\_rows, node\_uncertainty)

            # Using >= instead of > increases the ammount of correct answers

            if gain >= best\_gain:

                best\_gain, best\_attribute = gain, attribute

    return best\_gain, best\_attribute

One of the most important functions in this project. The way it works is simpler than it may look at first. We first initialize some auxiliary parameters: we set the gain as 0, in case we cannot find a split which gives us any information gain, and for the same reason we set the split attribute as None; we then calculate the impurity (uncertainty) of the node (to keep track of the accumulated uncertainty so far); and finally we get the number of columns we will have to iterate though.

After these settings, we start iterating through each column and get all the different values in that column (again, using the set structure type). For each of these values, we attempt to partition the data, get the information gained (using Gini in this project) and compare it with the latest value of information gained. If the current value is greater or equal (I got a higher score doing it this way), we store the gained information and the attribute as the best candidates for the partition and keep on iterating. Once we are finished, we will get the higher information gain and the attribute that got us that gain.

## divide(data, attribute):

# Divide the data based on the value of an attribute

def divide(data, attribute):

match\_rows, other\_rows = [], []

    for row in data:

        if attribute.compare(row):

            match\_rows.append(row)

        else:

            other\_rows.append(row)

    return match\_rows, other\_rows

Another one of the main functions in the tree building process. Given the data and an attribute (value and column) to split it, we check all the values for each row and classify them depending on whether they match the attribute or not, and then return the resulting lists.

## information\_gain(right\_rows, left\_rows, accumulated\_uncertainty):

# Calculate information gain using Gini Impurity (Gini Index)

def information\_gain(best\_split\_rows, other\_rows, node\_uncertainty):

    prob = float(len(best\_split\_rows)) / (len(other\_rows) + len(best\_split\_rows))

    return node\_uncertainty - prob \* gini(best\_split\_rows) - (1-prob) \* gini(other\_rows)

This function calculates the information gain of a given node using the Gini Index, taking into account the accumulated impurity so far (instead of assuming a value of 1 for each node). The formula is basically the accumulated uncertainty of the node minus the weighted uncertainty of each node (I found that this formula (using the previous uncertainty) was a bit more accurate).

## gini(data):

# Calculate gini impurity

def gini(data):

    values = count\_values(data)

    impurity = 1

    for value in values:

        probability = values[value] /float(len(data))

        impurity -= probability\*\*2

    return impurity

Implementation of the Gini Index formula.

## classify(data, node):

def classify(data, node):

    # Returns the key of the dictionary

    # (could return the whole dic, but we don't need the count anymore)

    if isinstance(node, Leaf):

        #return node.predictions

        return max(node.predictions, key=node.predictions.get)

    if node.attribute.compare(data):

        return classify(data, node.best\_partition\_branch)

    else:

        return classify(data, node.other\_branch)

Second part of the project. Once we are done building the decision tree with the training data, we proceed to classify the test data using our tree. This a recursive function that will return the value of the leaf object we get as a result. We start at the root node (named dt): if the data we are trying to classify matches the value of this node, we will get that value. Otherwise, we will send this data to each of the branches this node is referring to and attempt to classify it through them. The data will eventually reach a leaf and the result will be returned.

## main():

def main():

    # Read data and store labels

    training\_data\_file = sys.argv[1]

    test\_data\_file = sys.argv[2]

    result\_file = sys.argv[3]

    training\_data\_labels, training\_data = read\_file(training\_data\_file)

    test\_data\_labels, test\_data = read\_file(test\_data\_file)

    # Create tree using training data

    dt = build\_decision\_tree(training\_data)

    # Get predictions for each row

    results = []

    for row in test\_data:

        results.append(classify(row, dt))

    # Write output file

    write\_file(result\_file, training\_data\_labels, test\_data, results)

Core function of the program. Reads and prepares the data, creates the decision tree with the training data, classifies the test data and writes the results in a file.

# **INSTRUCTIONS FOR COMPILING**

The whole program is written in Python 3.x, in a single .py file, so no compilation is needed. It expects three arguments as input, in this order: training dataset, test dataset, output file name. So, going to the directory where the file is, the command for executing would be:

python dt.py [training dataset] [test dataset] [output file name]

ex:

python dt.py dt\_train.txt dt\_test.txt dt\_result.txt

# **ADDITIONAL SPECIFICATIONS**

All the files share the same directory, which might make it look a little messy, but made the execution a bit faster since I had to type less. Also, the dt\_test.exe, results and the answers are in this same directory, since in the instructions we were told to make sure they were in the same place, but it was not clear to me if we were supposed to save them in a separate directory or if it was okay to save them anywhere as long as they shared the same one.