# A2

2022年6月12日 星期日 下午5:10

a) The money is under box 2, and the box 1 and 3 are empty. In this case, the labels on box 1 and 2 are false and the label on box 3 is true, so only one of these labels is true. If the money is under box 1, the there are two labels are true (labels on box 2 and 3), which is a contradiction. If the money is under box 3, then the labels on box 1 and 2 are true, which, again, is a contradiction.

b). $B_1$ : the money is under box 1
$B_2$ : the money is under box 2
$B_3$ : the money is under box 3
$L_1$ : The label on box 1 is true
$L_2$ : The label on box 2 is true
$L_3$ : The label on box 3 is true

K: The label truth matches the fact:

$$((L_1 \land B_3) \lor (\neg L_1 \land \neg B_3)) \land ((L_2 \land \neg B_2) \lor (\neg L_2 \land \neg\neg B_2)) \land ((L_3 \land \neg B_3) \lor (\neg L_3 \land \neg\neg B_3))$$

OL: one and only one label is true.

$$(L_1 \lor L_2 \lor L_3) \land (L_1 \to (\neg L_2 \land \neg L_3)) \land (L_2 \to (\neg L_1 \land \neg L_3)) \land (L_3 \to (\neg L_1 \land \neg L_2))$$

U: under one of the boxes is a pile of money.

: $B_1 \lor B_2 \lor B_3$

O: Only one box has money under it, the other two have nothing

$$(B_1 \to (\neg B_2 \land \neg B_3)) \land (B_2 \to (\neg B_1 \land \neg B_3)) \land (B_3 \to (\neg B_1 \land \neg B_2))$$

c) O: $B_1 \to (\neg B_2 \land \neg B_3) \land B_2 \to (\neg B_1 \land \neg B_3) \land B_3 \to (\neg B_1 \land \neg B_2)$

$$\leftrightarrow (\neg B_1 \lor (\neg B_2 \land \neg B_3)) \land (\neg B_2 \lor (\neg B_1 \land \neg B_3)) \land (\neg B_3 \lor (\neg B_1 \land \neg B_2))$$

$$\leftrightarrow (\neg B_1 \lor \neg B_2) \land (\neg B_1 \lor \neg B_3) \land (\neg B_2 \lor \neg B_1) \land (\neg B_2 \lor \neg B_3) \land (\neg B_3 \lor \neg B_1) \land (\neg B_3 \lor \neg B_2)$$

$$\leftrightarrow (\neg B_1 \lor \neg B_2) \land (\neg B_2 \lor \neg B_3) \land (\neg B_3 \lor \neg B_1)$$

OL: $(L_1 \lor L_2 \lor L_3) \land (L_1 \to (\neg L_2 \land \neg L_3)) \land (L_2 \to (\neg L_1 \land \neg L_3)) \land (L_3 \to (\neg L_1 \land \neg L_2))$

$$\leftrightarrow (L_1 \lor L_2 \lor L_3) \land (\neg L_1 \lor (\neg L_2 \land \neg L_3)) \land (\neg L_2 \lor (\neg L_1 \land \neg L_3)) \land (\neg L_3 \lor (\neg L_1 \land \neg L_2))$$

$$\leftrightarrow (L_1 \lor L_2 \lor L_3) \land (\neg L_1 \lor \neg L_2) \land (\neg L_1 \lor \neg L_3) \land (\neg L_2 \lor \neg L_1) \land (\neg L_2 \lor \neg L_3) \land (\neg L_3 \lor \neg L_1) \land (\neg L_3 \lor \neg L_2)$$

$$\leftrightarrow (L_1 \lor L_2 \lor L_3) \land (\neg L_1 \lor \neg L_2) \land (\neg L_2 \lor \neg L_3) \land (\neg L_3 \lor \neg L_1)$$

K: $((L_1 \land B_3) \lor (\neg L_1 \land \neg B_3)) \land ((L_2 \land \neg B_2) \lor (\neg L_2 \land \neg\neg B_2)) \land ((L_3 \land \neg B_3) \lor (\neg L_3 \land \neg\neg B_3))$

$$\leftrightarrow ((L_1 \land B_3) \lor (\neg L_1 \land \neg B_3)) \land ((L_2 \land \neg B_2) \lor (\neg L_2 \land B_2)) \land ((L_3 \land \neg B_3) \lor (\neg L_3 \land B_3))$$

$$\leftrightarrow (L_1 \lor \neg L_1) \land (B_3 \lor \neg L_1) \land (L_1 \lor \neg B_3) \land (B_3 \lor \neg B_3) \land (L_2 \lor \neg L_2) \land (L_2 \lor B_2) \land (\neg B_2 \lor \neg L_2) \land (\neg B_2 \lor B_2) \land (L_3 \lor \neg L_3) \land (L_3 \lor B_3) \land (\neg B_3 \lor \neg L_3) \land (\neg B_3 \lor B_3)$$

$$\leftrightarrow True \land (B_3 \lor \neg L_1) \land (L_1 \lor \neg B_3) \land True \land True \land (L_2 \lor B_2) \land (\neg B_2 \lor \neg L_2) \land True \land True \land (L_3 \lor B_3) \land (\neg B_3 \lor \neg L_3) \land True$$

$$\leftrightarrow (B_3 \lor \neg L_1) \land (L_1 \lor \neg B_3) \land (L_2 \lor B_2) \land (\neg B_2 \lor \neg L_2) \land (L_3 \lor B_3) \land (\neg B_3 \lor \neg L_3)$$

So: $U \wedge O \wedge OLAK$

$\Leftrightarrow (B_1 \vee B_2 \vee B_3) \wedge (B_1 \to (\neg B_2 \wedge \neg B_3)) \wedge (B_2 \to (\neg B_1 \wedge \neg B_3)) \wedge (B_3 \to (\neg B_1 \wedge \neg B_2))$
$\wedge (L_1 \vee L_2 \vee L_3) \wedge (L_1 \to (\neg L_2 \wedge \neg L_3)) \wedge (L_2 \to (\neg L_1 \wedge \neg L_3)) \wedge (L_3 \to (\neg L_1 \wedge \neg L_2))$
$\wedge ((L_1 \wedge B_3) \vee (\neg L_1 \wedge \neg B_3)) \wedge ((L_2 \wedge \neg B_2) \vee (\neg L_2 \wedge \neg\neg B_2)) \wedge ((L_3 \wedge \neg B_3) \vee (\neg L_3 \wedge \neg\neg B_3))$

$\Leftrightarrow (B_1 \vee B_2 \vee B_3) \wedge (\neg B_1 \vee \neg B_2) \wedge (\neg B_2 \vee \neg B_3) \wedge (\neg B_3 \vee \neg B_1) \wedge (L_1 \vee L_2 \vee L_3) \wedge (\neg L_1 \vee \neg L_2)$
$\wedge (\neg L_2 \vee \neg L_3) \wedge (\neg L_3 \vee \neg L_1) \wedge (B_3 \vee \neg L_1) \wedge (L_1 \vee \neg B_3) \wedge (L_2 \vee B_2) \wedge (\neg B_2 \vee \neg L_2)$
$\wedge (L_3 \vee B_3) \wedge (\neg B_3 \vee \neg L_3)$

$\Leftrightarrow$: $\{\{B_1, B_2, B_3\}, \{\neg B_1, \neg B_2\}, \{\neg B_2, \neg B_3\}, \{\neg B_3, \neg B_1\}, \{L_1, L_2, L_3\}, \{\neg L_1, \neg L_2\}, \{\neg L_2, \neg L_3\},$
$\{\neg L_3, \neg L_1\}, \{B_3, \neg L_1\}, \{L_1, \neg B_3\}, \{L_2, B_2\}, \{\neg B_2, \neg L_2\}, \{L_3, B_3\}, \{\neg B_3, \neg L_3\}\}$

d) To prove: $U \wedge O \wedge OLAK \to B_2$

We show a contradiction. $U \wedge O \wedge OLAK \wedge \neg B_2 \models \bot$

$\{\{B_1, B_2, B_3\}, \{\neg B_1, \neg B_2\}, \{\neg B_2, \neg B_3\}, \{\neg B_3, \neg B_1\}, \{L_1, L_2, L_3\}, \{\neg L_1, \neg L_2\}, \{\neg L_2, \neg L_3\},$
$\{\neg L_3, \neg L_1\}, \{B_3, \neg L_1\}, \{L_1, \neg B_3\}, \{L_2, B_2\}, \{\neg B_2, \neg L_2\}, \{L_3, B_3\}, \{\neg B_3, \neg L_3\}, \{\neg B_2\}\}$

$\Leftrightarrow \{\{B_1, B_2, B_3\}, \{\neg B_1, \neg B_2\}, \{\neg B_2, \neg B_3\}, \{\neg B_3, \neg B_1\}, \{L_1, L_2, L_3\}, \{\neg L_1, \neg L_2\}, \{\neg L_2, \neg L_3\},$
$\{\neg L_3, \neg L_1\}, \{B_3, \neg L_1\}, \{L_1, \neg B_3\}, \{L_2, B_2\}, \{\neg B_2, \neg L_2\}, \{L_3, B_3\}, \{\neg B_3, \neg L_3\}, \{\neg B_2\},$
$\{L_2\}\}$

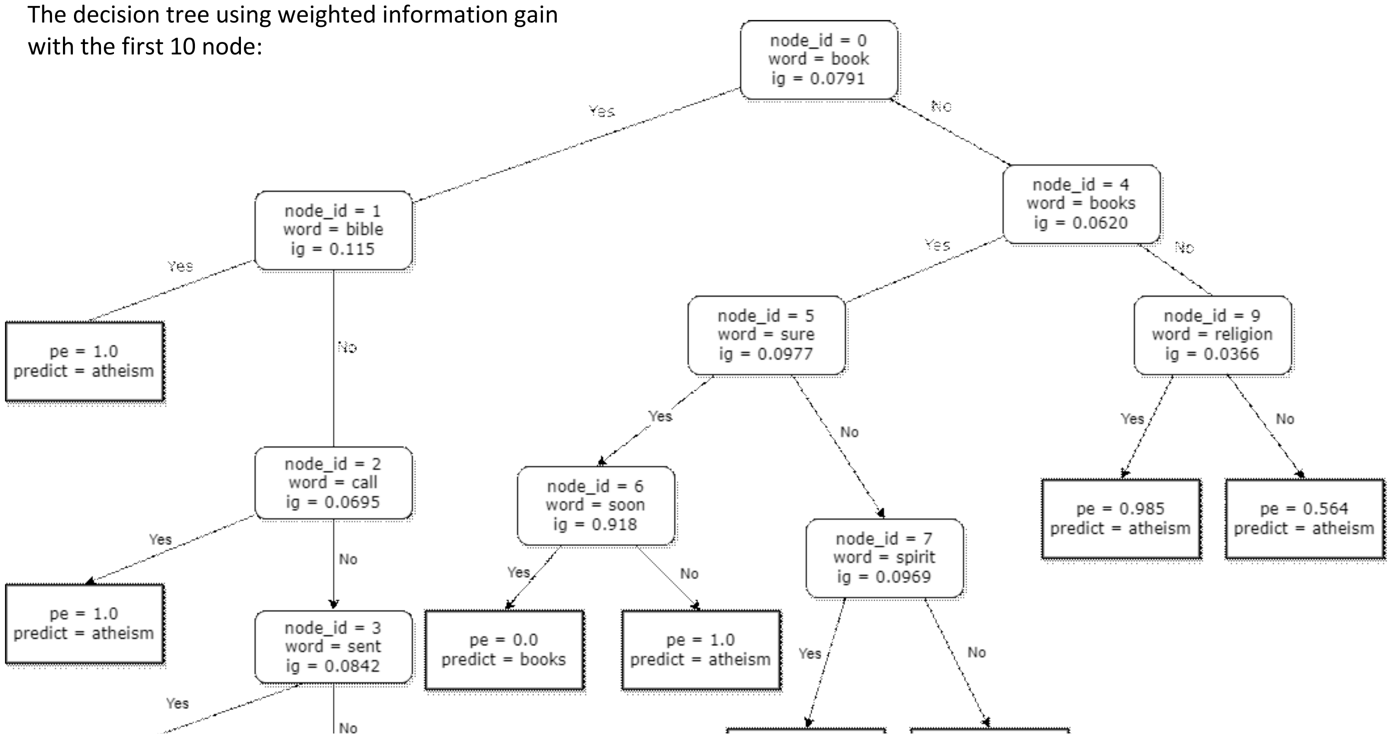
$\Leftrightarrow \{\{B_1, B_2, B_3\}, \{\neg B_1, \neg B_2\}, \{\neg B_2, \neg B_3\}, \{\neg B_3, \neg B_1\}, \{L_1, L_2, L_3\}, \{\neg L_1, \neg L_2\}, \{\neg L_2, \neg L_3\},$
$\{\neg L_3, \neg L_1\}, \{B_3, \neg L_1\}, \{L_1, \neg B_3\}, \{L_2, B_2\}, \{\neg B_2, \neg L_2\}, \{L_3, B_3\}, \{\neg B_3, \neg L_3\}, \{\neg B_2\},$
$\{L_2\}, \{\neg L_1\}, \{\neg L_3\}\}$

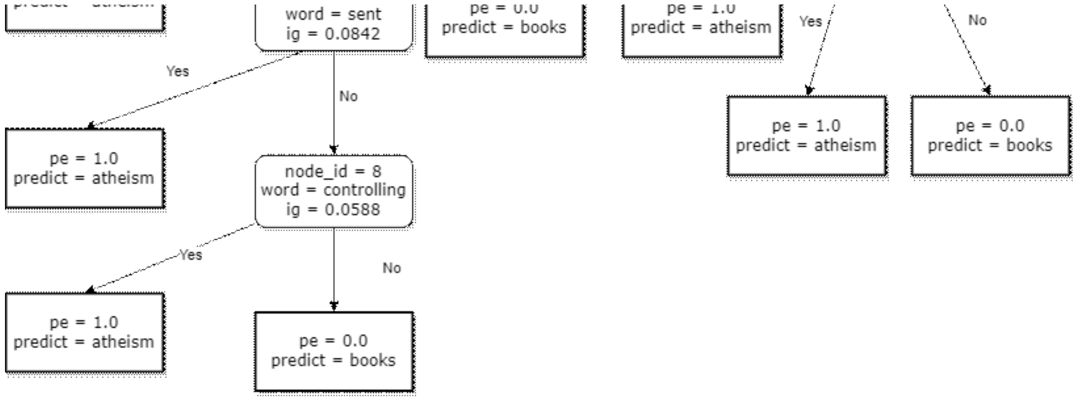
$\Leftrightarrow \{\{B_1, B_2, B_3\}, \{\neg B_1, \neg B_2\}, \{\neg B_2, \neg B_3\}, \{\neg B_3, \neg B_1\}, \{L_1, L_2, L_3\}, \{\neg L_1, \neg L_2\}, \{\neg L_2, \neg L_3\},$
$\{\neg L_3, \neg L_1\}, \{B_3, \neg L_1\}, \{L_1, \neg B_3\}, \{L_2, B_2\}, \{\neg B_2, \neg L_2\}, \{L_3, B_3\}, \{\neg B_3, \neg L_3\}, \{\neg B_2\},$
$\{L_2\}, \{\neg L_1\}, \{\neg L_3\}, \{\neg B_3\}, \{B_3\}\}$

$\Leftrightarrow \bot$

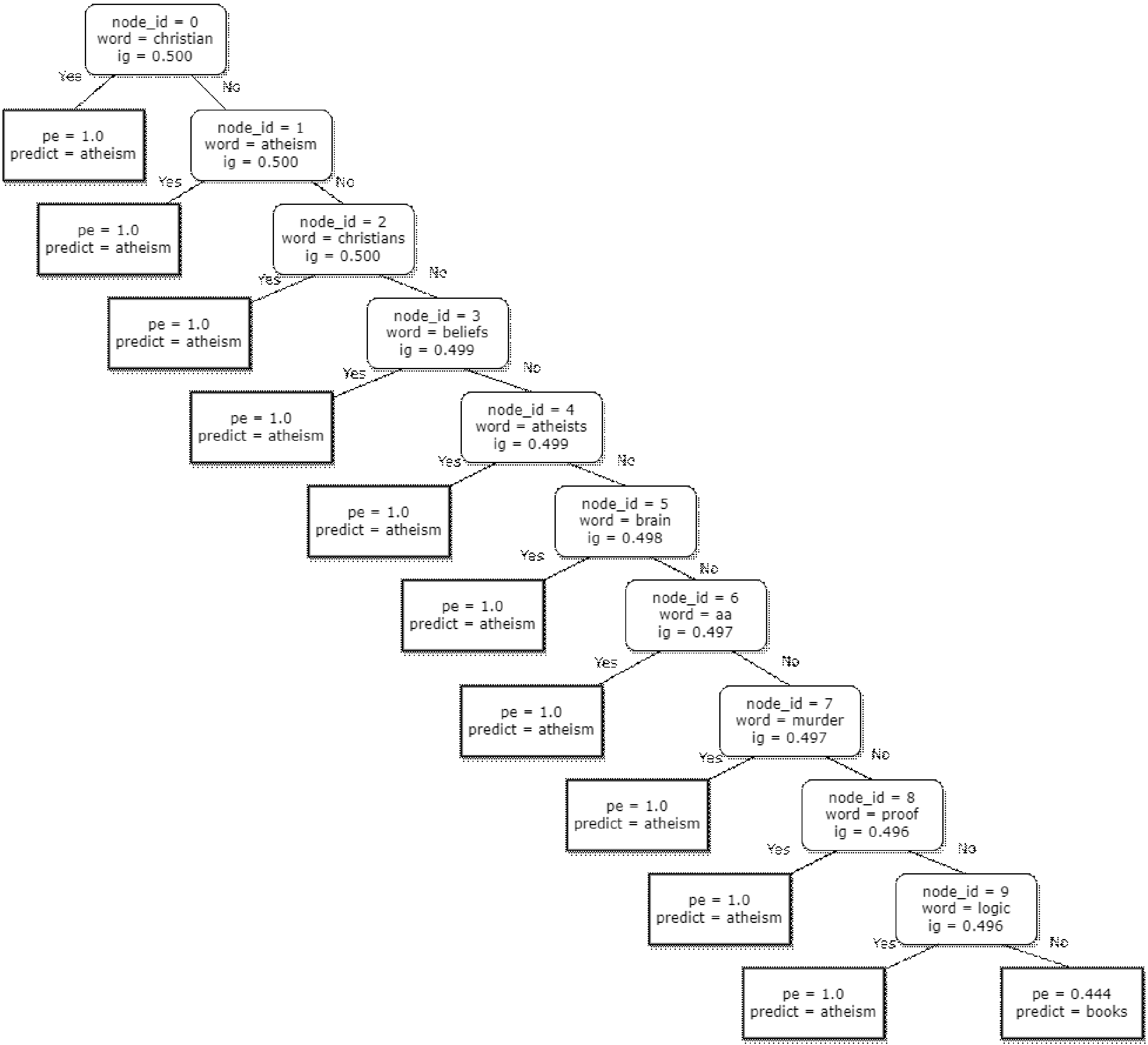
2. The code is at the end of this file.

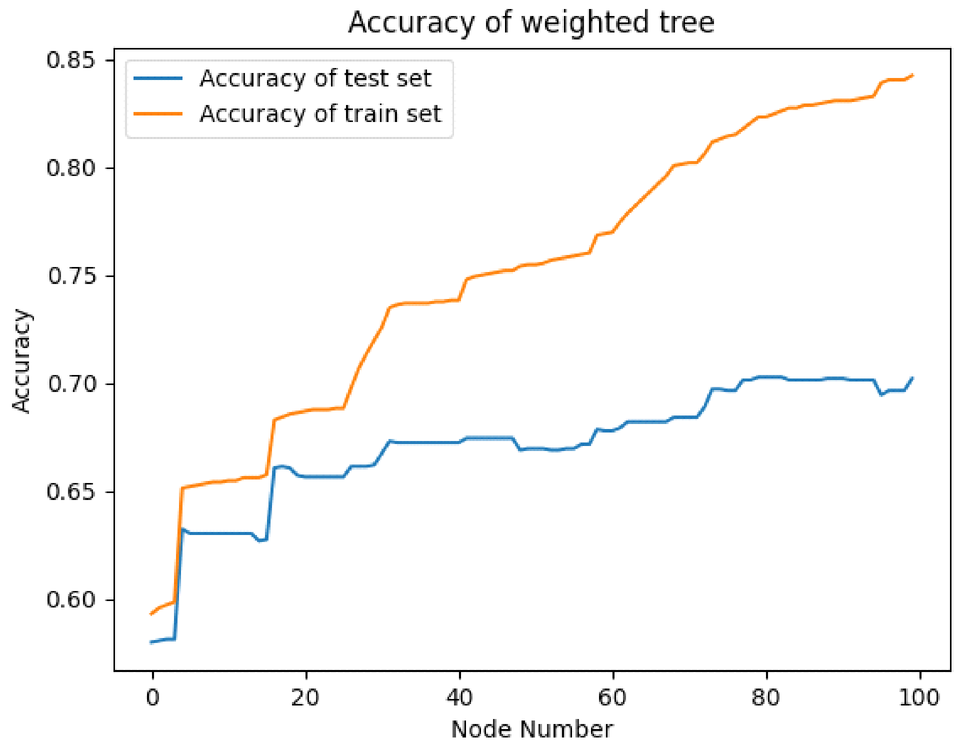
The decision tree using weighted information gain with the first 10 node:

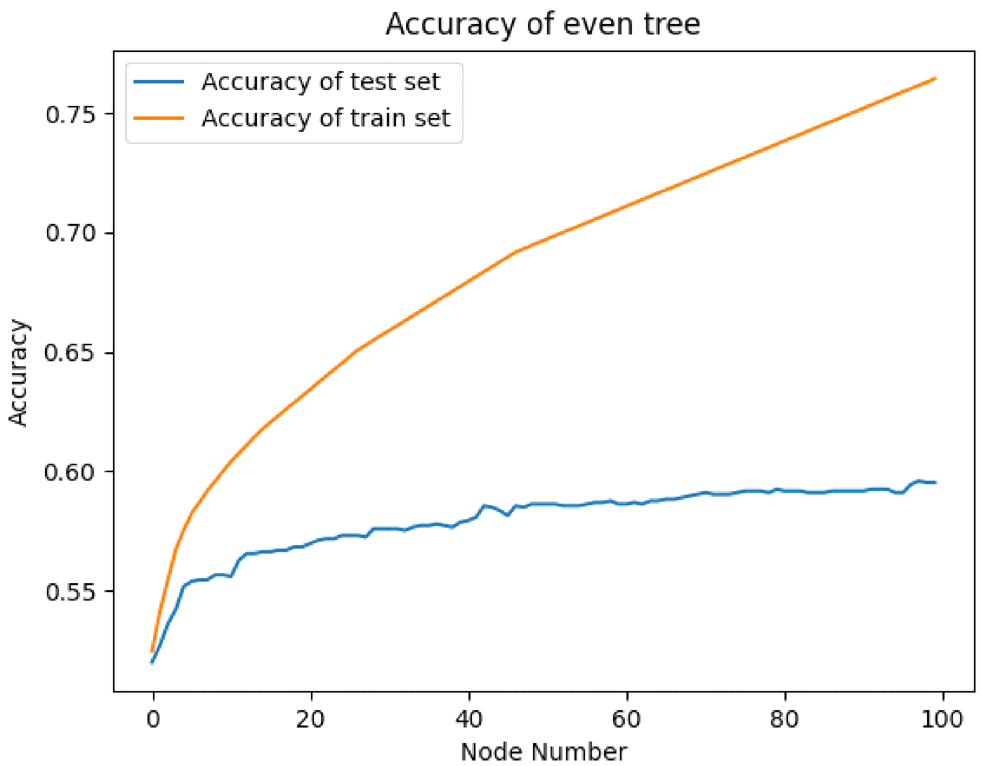





The decision tree using average information gain with the first 10 node:






```
from collections import defaultdict
from heapq import heappop, heappush
import sys
import numpy as np
from matplotlib import pyplot as plt

class DecisionTreeNode:
    def __init__(self, estimate, feature, info_gain, doc_set) -> None:
        self.estimate = estimate
        self.pos_child = None
        self.neg_child = None
```

```python
        self.feature = feature
        # Store negative information gain to maintain the property of heap since python heap is a min heap
        self.neg_information_gain = info_gain
        self.doc_set = doc_set
        self.id = -1

    # For print purpose
    def __str__(self, words_mapping=None, level=0, contain="root", prev_word=-1):
        if not words_mapping:
            ret = "|" + "\t"*level + contain + ": "  + str(prev_word) + ". current word: " + str(self.feature) + \
            " ig: " + str(-self.neg_information_gain) + " id: " + str(self.id) + "\n"
        else:
            ret = "|" + "\t"*level + contain + ": "  + words_mapping[prev_word-1] + ". current word: " + \
            words_mapping[self.feature-1] + " ig: " + str(-self.neg_information_gain) + " id: " + str(self.id) + "\n"

        if self.pos_child and self.neg_child:
            ret += self.pos_child.__str__(words_mapping, level+1, "contain", self.feature)
            ret += self.neg_child.__str__(words_mapping, level+1, "not contain", self.feature)
        else:
            ret += "|" + "\t"*(level + 1) + "PE: " + str(self.estimate) + "\n"

        return ret
    def __repr__(self):
        return '<tree node representation>'
    # For compare purpose, required by the min heap
    def _is_valid_operand(self, other):
            return hasattr(other, "neg_information_gain")
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return (self.neg_information_gain < other.neg_information_gain)


# Estimate the possiblity of label equalling Y.
def pointEstimate(Y: int, E: set, labels: list):
    N = len(E)
    if N == 0:
        return -1
    counter = len([doc_id for doc_id in E if labels[doc_id - 1]== Y])
    return counter / N


def findBestFeatureAndValue(E: set, words: list, data: dict, labels: list, weighted: bool = True):
    best_word = -1
    best_info_gain = -1
    def calculateInfo(_E: set):
        N = len(_E)
        # I({}) = 1, the entropy of an empty set is one.
        if N == 0:
            return 1

        # Count the number of instances with label one
        P_one = pointEstimate(1, _E, labels)
        P_two = 1 - P_one

        # 0 * log(0) = 0, but it is not handled by the library
        part_one = P_one * np.log2(P_one) if P_one > 0 else 0
        part_two = P_two * np.log2(P_two) if P_two > 0 else 0
        I_E = - part_one - part_two
        return I_E

    def calculateSubInfo(E_pos, E_neg, weighted = True):
        N_pos = len(E_pos)
        N_neg = len(E_neg)
        N = N_pos + N_neg
        I_E_pos = calculateInfo(E_pos)
        I_E_neg = calculateInfo(E_neg)
        # Evenly count the information of the two sub splits
        if not weighted:
            return (I_E_pos + I_E_neg) / 2
        # Calculat the information with weight
        return N_pos / N * I_E_pos + N_neg / N * I_E_neg
    I_E_zero = calculateInfo(E)
    for word_id in words:
        # Split on each word
        # find all docs that have this feature
        doc_contains_word = set([doc_id for doc_id in E if word_id in data[doc_id]])
        doc_not_contains_word = set([doc_id for doc_id in E if word_id not in data[doc_id]])
```

```
        sub_info = calculateSubInfo(doc_contains_word, doc_not_contains_word, weighted=weighted)
        info_gain = I_E_zero - sub_info
        # Update the feature and info_gain if we found a better feature
        if info_gain > best_info_gain:
            best_word = word_id
            best_info_gain = info_gain

    return best_word, best_info_gain


def train(train_dict, train_label, USE_WEIGHTED, TREE_SIZE):
    E = set(train_dict.keys())
    # Initialization of the first root node
    X, delta_I = findBestFeatureAndValue(E, words, train_dict, train_label, USE_WEIGHTED)
    start_node = DecisionTreeNode(pointEstimate(1, E, train_label), X, -delta_I, E)
    pq = [start_node]
    node_num = 0
    while node_num < TREE_SIZE and pq:
        cur_node = heappop(pq)
        cur_node.id = node_num
        # Contain Feature
        E_contain = set([doc_id for doc_id in cur_node.doc_set if cur_node.feature in train_dict[doc_id]])
        X_contain, delta_I_contain = findBestFeatureAndValue(E_contain, words, train_dict, train_label, USE_WEIGHTED)
        PE_pos = pointEstimate(1, E_contain, train_label)
        T_pos = DecisionTreeNode(PE_pos, X_contain, -delta_I_contain, E_contain)
        heappush(pq, T_pos)
        # Not Contain Feature
        E_not_contain = set([doc_id for doc_id in cur_node.doc_set if cur_node.feature not in train_dict[doc_id]])
        X_not_contain, delta_I_not_contain = findBestFeatureAndValue(E_not_contain, words, train_dict, train_label,
USE_WEIGHTED)
        PE_neg = pointEstimate(1, E_not_contain, train_label)
        T_neg = DecisionTreeNode(PE_neg, X_not_contain, -delta_I_not_contain, E_not_contain)
        heappush(pq, T_neg)
        # Append child
        cur_node.pos_child = T_pos
        cur_node.neg_child = T_neg
        # add node_num
        node_num += 1
    return start_node


def verification(decisionTree: DecisionTreeNode, test_dict, test_label, TREE_SIZE):
    # A list to store the accuracy. Index i means the accuracy when the tree only have i+1 nodes
    correctly_identified = [0] * TREE_SIZE
    N = len(test_dict.keys())
    for doc_id, words in test_dict.items():
        root = decisionTree
        # Predict
        while root.pos_child and root.neg_child:
            # Find which side of the tree it goes
            if root.feature in words:
                leaf = root.pos_child
            else:
                leaf = root.neg_child

            range_end = leaf.id if leaf.id != -1 else len(correctly_identified)
            if (leaf.estimate > 0.5 and test_label[doc_id-1] == 1) or (leaf.estimate <= 0.5 and test_label[doc_id-1]
== 2):
                # correctly predicted, update the identified counter accordingly
                for i in range(root.id, range_end):
                    # At this moment, we tree the leaf node as the leaf
                    # so the result is consistent for tree from size root.id + 1 to size range_end
                    correctly_identified[i] += 1
            root = leaf
    return [num / N for num in correctly_identified]


if __name__ == "__main__":
    # Parse the input parameter to get the data folder, using current working directory as default
    args = sys.argv[1:]
    data_folder = args[0] if len(args) > 0 else '.'
    if data_folder.endswith("/") or data_folder.endswith("\\"):
        data_folder = data_folder[:-1]
    # Load data
    train_data = np.loadtxt(f"{data_folder}/trainData.txt", delimiter=" ")
    train_label = np.loadtxt(f"{data_folder}/trainLabel.txt", delimiter=" ")
    train_label = [int(i) for i in train_label]
    test_data = np.loadtxt(f"{data_folder}/testData.txt", delimiter=" ")
    test_label = np.loadtxt(f"{data_folder}/testLabel.txt", delimiter=" ")
```

```python
test_label = [int(i) for i in test_label]
words = []              # The index of words
words_mapping = []      # The mapping of words for printing
with open(f"{data_folder}/words.txt") as f:
    words_mapping = [line.strip() for line in f]
with open(f"{data_folder}/words.txt") as f:
    words = range(1, 1 + len(f.readlines()))

# Parse the train data into a dict, key: doc_id, value: set of word_id for better performance
train_dict = defaultdict(set)
for (doc_id, word_id) in train_data:
    train_dict[int(doc_id)].add(int(word_id))
test_dict = defaultdict(set)
for (doc_id, word_id) in test_data:
    test_dict[int(doc_id)].add((int(word_id)))
TREE_SIZE = 100
print("Training weighted")
weighted_tree = train(train_dict, train_label, True, TREE_SIZE)

print("Predicting weighted")
accuracy_weighted_test = verification(weighted_tree, test_dict, test_label, TREE_SIZE)
accuracy_weighted_train = verification(weighted_tree, train_dict, train_label, TREE_SIZE)
# plot graph
plt.xlabel("Node Number")
plt.ylabel("Accuracy")
plt.plot(range(TREE_SIZE), accuracy_weighted_test, label="Accuracy of test set")
plt.plot(range(TREE_SIZE), accuracy_weighted_train, label="Accuracy of train set")
plt.legend(loc='best')
plt.title("Accuracy of weighted tree")
plt.savefig(f"Weighted_{TREE_SIZE}.png")
# plt.show()
plt.close()
print("Training even")
even_tree = train(train_dict, train_label, False, TREE_SIZE)
print("Predicting even")
accuracy_even_test = verification(even_tree, test_dict, test_label, TREE_SIZE)
accuracy_even_train = verification(even_tree, train_dict, train_label, TREE_SIZE)
# plot graph
plt.xlabel("Node Number")
plt.ylabel("Accuracy")
plt.plot(range(TREE_SIZE), accuracy_even_test, label="Accuracy of test set")
plt.plot(range(TREE_SIZE), accuracy_even_train, label="Accuracy of train set")
plt.legend(loc='best')
plt.title("Accuracy of even tree")
plt.savefig(f"Even_{TREE_SIZE}.png")
# plt.show()
plt.close()
# print(weighted_tree.__str__(words_mapping))
# print(even_tree.__str__(words_mapping))
```