CS486 - Assignment2

Zhaobo Ding

February 17, 2020

1. (1) Money is under Box 1.
   Proof: Since one and only one box has money under it, we enumerate all cases for placing money.
   Case 1: If money is under Box 1, then labels on Box 2 is true, while those on Box 1 and 3 are false. This case meets the requirement that one and only one of the labels is true.
   Case 2: If money is under Box 2, then labels on Box 1 and 3 are true, while that on Box 2 is false. This case does not meet the requirement that one and only one of the labels is true.
   Case 3: If money is under Box 3, then labels on Box 1 and 2 are true, while that on Box 3 is false. This case does not meet the requirement that one and only one of the labels is true.
   Hence, only Case 1 is correct, implying that money is under Box 1.

   (2) Let A, B, C be the binary variable representing respectively whether money is under Box 1, 2 and 3. In the other words, we define:
   A: Money is under Box 1.
   B: Money is under Box 2.
   C: Money is under Box 3.

   (3) The labels on boxes respectively gives $\neg A$, $\neg B$ and B.
   Only one box has money under it gives

   $$(A \wedge (\neg B) \wedge (\neg C)) \vee ((\neg A) \wedge B \wedge (\neg C)) \vee ((\neg A) \wedge (\neg B) \wedge C)$$

   One and only one of the labels is true gives

   $$((\neg A) \wedge (\neg(\neg B)) \wedge (\neg B)) \vee ((\neg(\neg A)) \wedge (\neg B) \wedge (\neg B)) \vee ((\neg(\neg A)) \wedge (\neg(\neg B)) \wedge B)$$

   Therefore, the CNF is
   $((A \wedge (\neg B) \wedge (\neg C)) \vee ((\neg A) \wedge B \wedge (\neg C)) \vee ((\neg A) \wedge (\neg B) \wedge C)) \wedge$
   $(((\neg A) \wedge (\neg(\neg B)) \wedge (\neg B)) \vee ((\neg(\neg A)) \wedge (\neg B) \wedge (\neg B)) \vee ((\neg(\neg A)) \wedge (\neg(\neg B)) \wedge B))$

   (4) We will show the proof by resolute the CNF.
   Applying Double-negation Law, we will get

   $$((A\wedge(\neg B)\wedge(\neg C))\vee((\neg A)\wedge B\wedge(\neg C))\vee((\neg A)\wedge(\neg B)\wedge C))\wedge(((\neg A)\wedge B\wedge(\neg B))\vee(A\wedge(\neg B)\wedge(\neg B))\vee(A\wedge B\wedge B))$$

   Applying Idempotent Laws, we will get

   $$((A\wedge(\neg B)\wedge(\neg C))\vee((\neg A)\wedge B\wedge(\neg C))\vee((\neg A)\wedge(\neg B)\wedge C))\wedge(((\neg A)\wedge B\wedge(\neg B))\vee(A\wedge(\neg B))\vee(A\wedge B))$$

Applying Contradiction Law, we will get

$$((A \wedge (\neg B) \wedge (\neg C)) \vee ((\neg A) \wedge B \wedge (\neg C)) \vee ((\neg A) \wedge (\neg B) \wedge C)) \wedge (false \vee (A \wedge (\neg B)) \vee (A \wedge B))$$

By property of "OR", we have

$$((A \wedge (\neg B) \wedge (\neg C)) \vee ((\neg A) \wedge B \wedge (\neg C)) \vee ((\neg A) \wedge (\neg B) \wedge C)) \wedge ((A \wedge (\neg B)) \vee (A \wedge B))$$

Applying Distributive Law, we will get

$$((A \wedge (\neg B) \wedge (\neg C)) \vee ((\neg A) \wedge B \wedge (\neg C)) \vee ((\neg A) \wedge (\neg B) \wedge C)) \wedge (A \wedge ((\neg B) \vee B))$$

Applying Excluded middle Law, we will get

$$((A \wedge (\neg B) \wedge (\neg C)) \vee ((\neg A) \wedge B \wedge (\neg C)) \vee ((\neg A) \wedge (\neg B) \wedge C)) \wedge A$$

Applying Distributive Laws and then Associative Laws, we will get

$$(A \wedge A \wedge (\neg B) \wedge (\neg C)) \vee (A \wedge (\neg A) \wedge B \wedge (\neg C)) \vee (A \wedge (\neg A) \wedge (\neg B) \wedge C))$$

Applying Idempotent Laws and Contradiction Law again, we will get

$$(A \wedge (\neg B) \wedge (\neg C)) \vee (false \wedge B \wedge (\neg C)) \vee (false \wedge (\neg B) \wedge C))$$

By property of "AND", we have

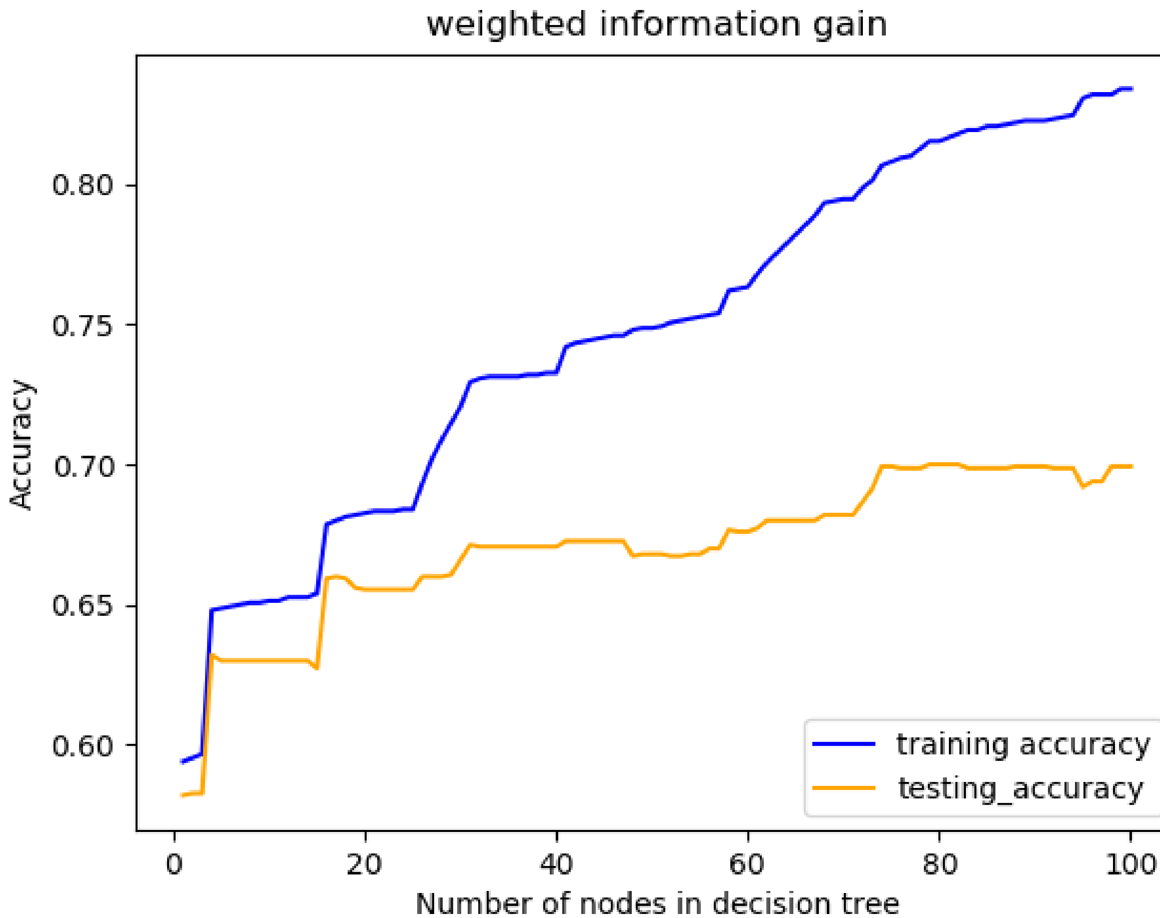
$$(A \wedge (\neg B) \wedge (\neg C)) \vee false \vee false$$

By property of "OR", we have

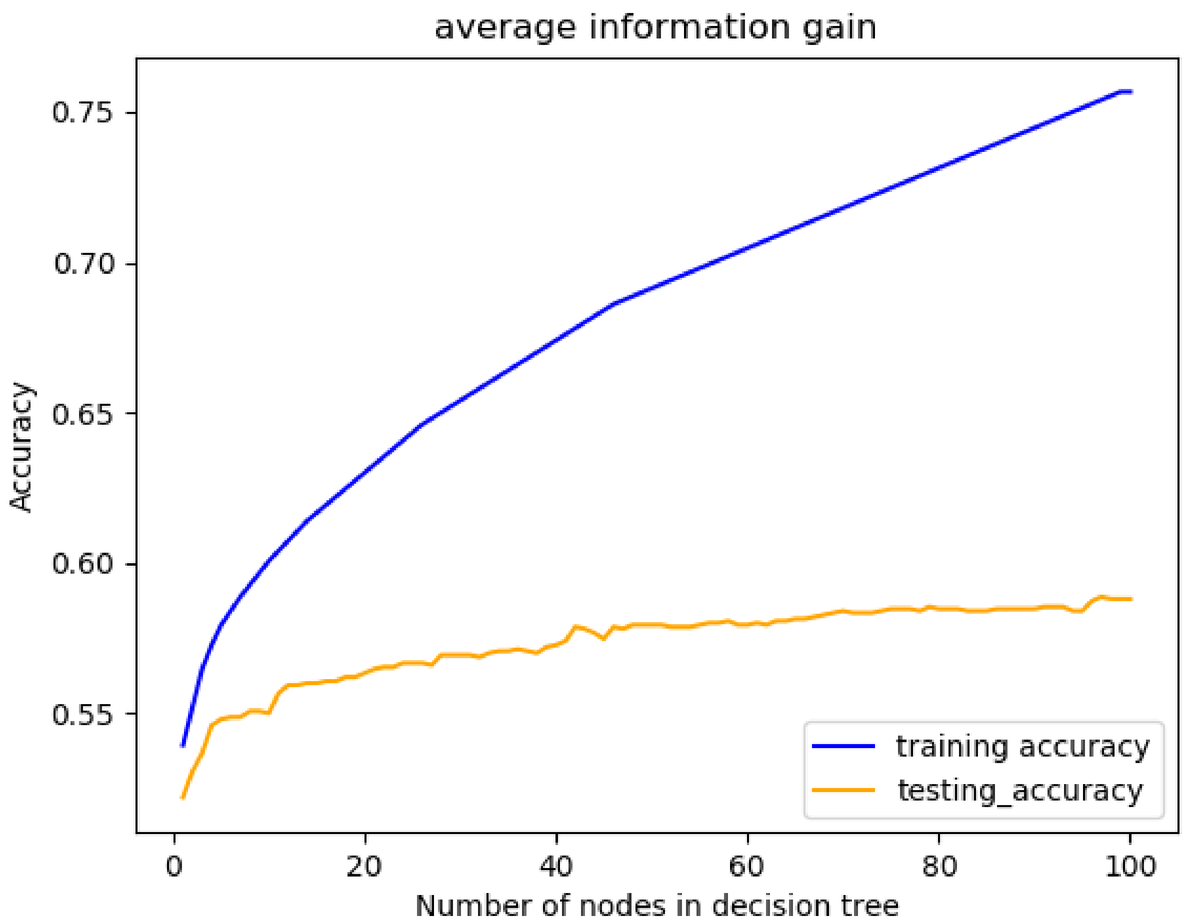
$$A \wedge (\neg B) \wedge (\neg C)$$

Therefore, to meet the requirements in the condition, we must have $A = true, B = C = false$, implying that money is under Box 1.

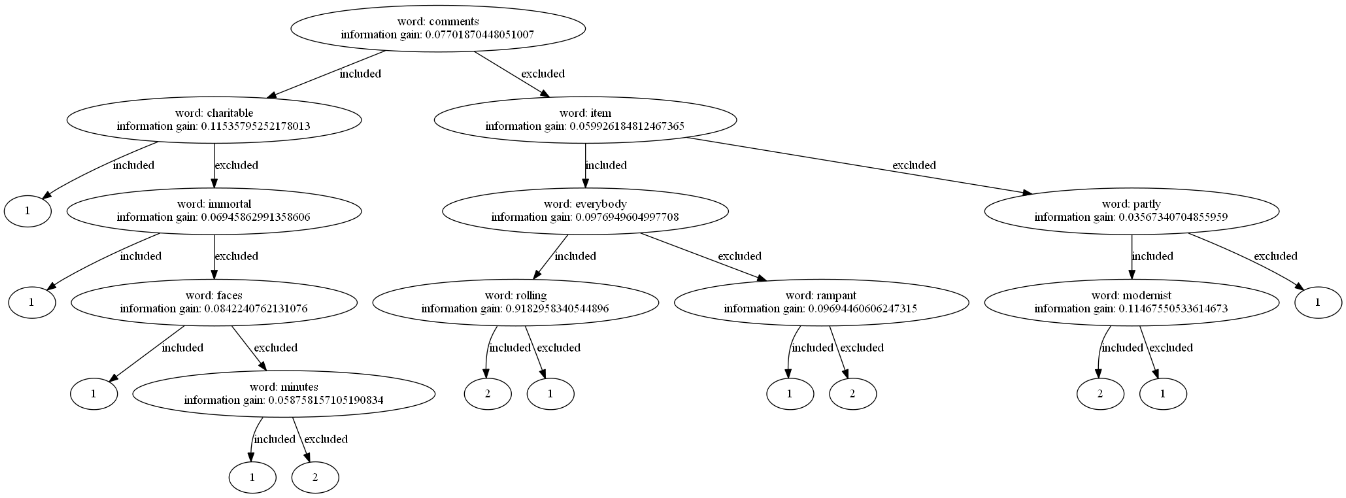
2. Accuracy of decision tree trained based on Weighted Information Gain method:



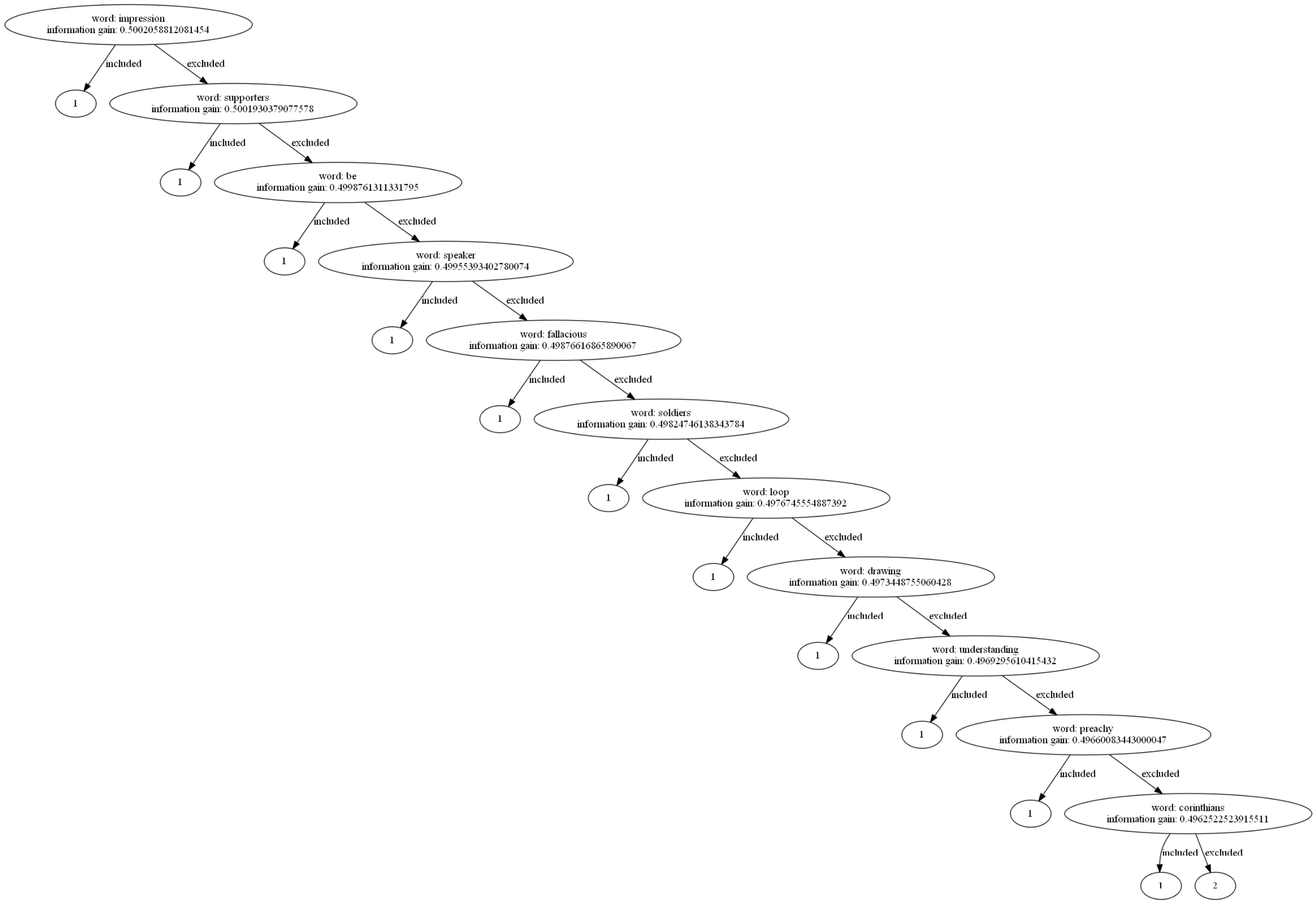
Accuracy of decision tree trained based on Average Information Gain method:



Structure of decision tree trained based on Weighted Information Gain method:



Structure of decision tree trained based on Average Information Gain method:



Following are the files of code. Running 'assignment.py' with Python 3.8 will generate the four images we attached above.

```
# config.py
train_data_path = './dataset/trainData.txt'
train_label_path = './dataset/trainLabel.txt'
test_data_path = './dataset/testData.txt'
test_label_path = './dataset/testLabel.txt'
words_path = './dataset/words.txt'
decision_tree_path = './decision_tree.pkl'
decision_tree_picture_path = './decision_tree.png'

AIM_TREE_SIZE = 100

AVERAGE_INFORMATION_GAIN = 1
WEIGHTED_INFORMATION_GAIN = 2
FEATURE_SELECTION_MECHANISM = WEIGHTED_INFORMATION_GAIN
```

```
# max_priority_queue.py
from functools import total_ordering

@total_ordering
```

```python
class QueueNode:
    def __init__(self, gain, root, split_word):
        self.gain = gain
        self.root = root
        self.split_word = split_word

    def __eq__(self, other):
        return self.gain == other.gain

    def __lt__(self, other):
        return self.gain > other.gain
```

```python
# data_io.py
import pickle
from A2.configs import *
from anytree import Node
from anytree.exporter import DotExporter

class Document:
    def __init__(self, doc_id=0, label=None):
        self.id = doc_id
        self.label = label
        self.wordList = set()

    def __contains__(self, word):
        return word in self.wordList

    def add_word(self, word):
        self.wordList.add(word)

def load_train_data():
    return load_data(train_data_path, train_label_path)

def load_test_data():
    return load_data(test_data_path, test_label_path)

def load_data(data_path, label_path):
    docs = []
    with open(label_path, 'r', encoding='utf-8') as file:
        for index, line in enumerate(file):
            doc_id = index + 1
            label = int(line.strip())
            docs.append(Document(doc_id, label))

    with open(data_path, 'r', encoding='utf-8') as file:
        for line in file.readlines():
            [doc_id, word_id] = list(map(int, line.strip().split(' ')))
            docs[doc_id - 1].add_word(word_id)
```

```
    return docs

def load_words(filename=words_path):
    word_map = dict()

    with open(filename, 'r', encoding='utf-8') as file:
        for index, word in enumerate(file):
            word_map[index] = word.strip()

    return word_map

def store_tree(tree, tree_path=decision_tree_path):
    with open(tree_path, 'wb') as file:
        pickle.dump(tree, file, pickle.HIGHEST_PROTOCOL)

def load_tree(tree_path=decision_tree_path):
    with open(tree_path, 'rb') as file:
        tree = pickle.load(file)

    return tree

def build_tree(root, tree_size, parent=None, edge=None):
    def node_information(node):
        return "word: " + node.word + "\n" + \
               "information gain: " + str(node.information_gain) + "\n"

    if root is None:
        return

    if root.leaf or root.order > tree_size:
        return Node(id(root),
                    parent=parent,
                    edge=edge,
                    display_name=root.pred)
    else:
        node = Node(id(root),
                    parent=parent,
                    edge=edge,
                    display_name=node_information(root))
        build_tree(root.included, tree_size, node, "included")
        build_tree(root.excluded, tree_size, node, "excluded")
        return node

def render(tree, tree_size, filename=decision_tree_picture_path):
    tree_to_render = build_tree(tree.root, tree_size)
    DotExporter(tree_to_render,
                nodeattrfunc=lambda node: 'label="{}"'.format(node.display_name),
```

```
                edgeattrfunc=lambda p, c: 'label="{}"'.format(c.edge)).to_picture(filename)
```

```python
# decision_tree.py
import math
from collections import Counter
from .configs import *
from .max_priority_queue import QueueNode
from queue import PriorityQueue

def entropy(p):
    return -p * math.log2(p)

def information_entropy(doc_labels):
    stat = Counter(doc_labels)
    doc_length = len(doc_labels)
    if doc_length:
        return sum([entropy(freq / doc_length) for freq in stat.values()])
    else:
        return 1

def information_gain(docs, word, method=FEATURE_SELECTION_MECHANISM):
    e = [doc.label for doc in docs]
    e1 = [doc.label for doc in docs if word in doc]
    e2 = [doc.label for doc in docs if word not in doc]
    ie = information_entropy(e)
    ie1 = information_entropy(e1)
    ie2 = information_entropy(e2)

    if method is AVERAGE_INFORMATION_GAIN:
        return ie - (ie1 + ie2) / 2
    elif method is WEIGHTED_INFORMATION_GAIN:
        return ie - (len(e1) * ie1 + len(e2) * ie2) / len(docs)

class DecisionTreeNode:
    def __init__(self, docs):
        self.word_id = None
        self.word = None
        self.docs = docs
        self.included = None
        self.excluded = None
        self.order = None
        self.leaf = True
        self.information_gain = 0
        labels = [doc.label for doc in docs]
        self.pred = Counter(labels).most_common(1)[0][0]

    def split(self, word, order):
        in_list = [doc for doc in self.docs if word in doc]
```

```python
        out_list = [doc for doc in self.docs if word not in doc]
        self.included = DecisionTreeNode(in_list)
        self.excluded = DecisionTreeNode(out_list)

        self.order = order
        self.leaf = False
        del self.docs

        return self.included, self.excluded

    def best_split(self, words, method):
        split_results = {word: information_gain(self.docs, word, method) for word in words}
        chosen_word = max(split_results, key=split_results.get)
        self.word_id = chosen_word
        self.information_gain = split_results[chosen_word]
        return chosen_word, split_results[chosen_word]

class DecisionTree:
    def __init__(self, word_map, algo=WEIGHTED_INFORMATION_GAIN):
        self.root = None
        self.word_map = word_map
        self.word_set = set(word_map)
        self.size = 0
        self.algo = algo

    def train(self, docs, size=AIM_TREE_SIZE):
        self.root = DecisionTreeNode(docs)
        q = PriorityQueue()
        split_word, ig = self.root.best_split(self.word_set, self.algo)
        self.root.word = self.word_map[split_word]
        q.put(QueueNode(ig, self.root, split_word))
        self.size = 0

        while self.size < size:
            node = q.get()
            if node.root.leaf:
                if node.gain:
                    inc, exc = node.root.split(node.split_word, self.size)
                    spi, igi = inc.best_split(self.word_set, self.algo)
                    spe, ige = exc.best_split(self.word_set, self.algo)
                    q.put(QueueNode(igi, inc, spi))
                    q.put(QueueNode(ige, exc, spe))
                    inc.word = self.word_map[spi]
                    exc.word = self.word_map[spe]
                    self.size += 1
                else:
                    break
```

```python
        while not q.empty():
            node = q.get()
            del node.root.docs

    def predict(self, doc, size=0):
        node = self.root
        while not node.leaf and (size == 0 or node.order <= size):
            if node.word_id in doc:
                node = node.included
            else:
                node = node.excluded

        return node.pred
```

```python
# assignment.py
import matplotlib.pyplot as plt
from A2.decision_tree import DecisionTree
from A2.configs import AIM_TREE_SIZE, AVERAGE_INFORMATION_GAIN, WEIGHTED_INFORMATION_GAIN
from A2.data_io import load_train_data, load_test_data, load_words, render

def build_decision_tree(size, method):
    train_data = load_train_data()
    word_map = load_words()

    tree = DecisionTree(word_map, method)
    tree.train(train_data, size)

    return tree

def test_decision_tree(tree, size):
    def test(data):
        correct = incorrect = 0
        for doc in data:
            if tree.predict(doc, size) == doc.label:
                correct += 1
            else:
                incorrect += 1

        return correct / (correct + incorrect)

    test_data = load_test_data()
    train_data = load_train_data()

    return test(test_data), test(train_data)

def generate_assignment_files(method):
    suffix = 'average' if method == AVERAGE_INFORMATION_GAIN else 'weighted'
    TREE_SIZE_TO_DRAW = 10
```

```
    train_data = load_train_data()
    word_map = load_words()
    decision_tree = DecisionTree(word_map, method)
    decision_tree.train(train_data)

    test_result = []
    train_result = []

    for i in range(AIM_TREE_SIZE):
        tree_size = i + 1
        test_accuracy, train_accuracy = test_decision_tree(decision_tree, tree_size)
        test_result.append(test_accuracy)
        train_result.append(train_accuracy)

    filename = './decision_tree_' + suffix + '.png'
    render(decision_tree, TREE_SIZE_TO_DRAW, filename)

    size = list(range(1, AIM_TREE_SIZE + 1))
    plt.figure()
    plt.plot(size, train_result, label='training accuracy', color='blue')
    plt.plot(size, test_result, label='testing_accuracy', color='orange')
    plt.legend(loc='lower right')
    plt.xlabel('Number of nodes in decision tree')
    plt.ylabel('Accuracy')
    plt.title(suffix + ' information gain')
    plt.savefig('./accuracy_plot_' + suffix + '.png')


if __name__ == "__main__":
    generate_assignment_files(AVERAGE_INFORMATION_GAIN)
    generate_assignment_files(WEIGHTED_INFORMATION_GAIN)
```