Assignment 2

1.1)

The box is in the **first box.**
Proof by brute force:

1. If #1 was true, then box 2 shouldn't be empty according to #2, but the money shouldn't be in box 2 according to #3. #2 and #3 contradict themselves.
2. If #2 was true, then box 2 is empty. According to #1, box 1 should have the money, and according to the #3, box 2 doesn't have the money. This works out.

1.2)

Let's define A, B, C to represent whether the first, second or third box hold the money respectively.
A: The first box has the money
B: The second box has the money
C: The third box has the money

1.3)

Label 1 implies $\neg A$
Label 2 implies $\neg B$
Label 3 implies B

We also know that only one label is true:
$(\neg A \wedge \neg(\neg B) \wedge \neg(B)) \vee$ `// label 1 is true`
$(\neg(\neg A) \wedge \neg B \wedge \neg(B)) \vee$ `// label 2 is true`
$(\neg(\neg A) \wedge \neg(\neg B) \wedge B)$ `// label 3 is true`

We also know that only one box has the money:
$(A \wedge \neg B \wedge \neg C) \vee$ `// box 1 has the money`
$(\neg A \wedge B \wedge \neg C) \vee$ `// box 2 has the money`
$(\neg A \wedge \neg B \wedge C)$ `// box 3 has the money`

CNF Form:
$((\neg A \wedge \neg(\neg B) \wedge \neg(B)) \vee$
$(\neg(\neg A) \wedge \neg B \wedge \neg(B)) \vee$
$(\neg(\neg A) \wedge \neg(\neg B) \wedge B))$
$\wedge$
$((A \wedge \neg B \wedge \neg C) \vee$
$(\neg A \wedge B \wedge \neg C) \vee$
$(\neg A \wedge \neg B \wedge C))$
$\vdash A$

1.4)

Prove by contradiction

Reduce all $\neg\neg$

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

Reduce redundant $\wedge$ terms

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

Remove contradicting clauses

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

Demorgans law

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

Demorgans law again

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

Reduce contradictions and redundant terms

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

Break first parenthesis

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

A cannot be both true and false, so we have a contradiction. As a result, box 1 has the money.

2)

Accuracy:
Average: 85.431%
Weighted: 81.188%

Code:

```
from collections import defaultdict
from Queue import PriorityQueue as queue
import math

AVERAGE_INFO_GAIN = 1
WEIGHTED_INFO_GAIN = 2
APPROACH = WEIGHTED_INFO_GAIN

RENDER = True


class Node:

    def __init__(self, word_id, docs, word_ids, is_terminal=False, contains=
None):
```

```
        self.word_id = word_id
        self.entropy = entropy(docs)
        self.children = {}
        self.docs = docs
        self.word_ids = word_ids
        if is_terminal:
            self.info_gain = 0.0
        else:
            self.info_gain = information_gain(self.docs, self.word_id)
        self.split = self.get_split()
        self.contains = contains
        self.is_terminal = is_terminal

    def add_child(self, node, val):
        self.children[val] = node

    def has_child(self, val):
        return val in self.children

    def child_count(self):
        return len(self.children)

    def is_end(self):
        # Checks if there are any empty
        labels = self.split
        if labels[True][1] + labels[True][2] == 0 or labels[False][1] + labels[False][2] == 0:
            return True
        return False

    def get_split(self):
        labels = {}
        labels[True] = defaultdict(int)
        labels[False] = defaultdict(int)

        for d in self.docs:
            if d.has_word(self.word_id):
                labels[True][d.label] += 1
```

```python
            else:
                labels[False][d.label] += 1

        return labels

    def format_split(self):
        labels = self.split
        return 'True-1: {} True-2: {} False-1: {} False-2: {}'.format(labels[True][1], labels[True][2], labels[False][1], labels[False][2])

    def __cmp__(self, node):
        if self.entropy < node.entropy:
            return -1
        elif self.entropy > node.entropy:
            return 1
        return 0

    def __str__(self):
        return str(self.word_id) + ' ' + str(self.info_gain) + ' ' + 'docs: ' + str(len(self.docs)) + ' ' + 'word_ids: ' + str(self.word_ids)

    def get_id(self):
        word_ids = map(str, self.word_ids)
        if self.is_terminal:
            return str(self.contains) + '-' + '-'.join(word_ids)
        return '-'.join(word_ids)


class Option:

    def __init__(self, n1, n2, word_id, contains):
        self.n1 = n1
        self.n2 = n2
        self.gain = information_gain(n2.docs, n2.word_id)
        self.contains = contains

    def __cmp__(self, option):
        if self.gain < option.gain:
```

```
            return 1
        elif self.gain > option.gain:
            return -1
        return 0

    def __str__(self):
        return str(self.n1.word_id) + ' ' + str(self.n2.word_id) + ' ' + str(self.gain) + ' ' + str(self.contains)


class Doc:

    def __init__(self, id):
        self.id = id
        self.label = None
        self.words = defaultdict(bool)

    def add_word(self, index):
        self.words[index] = True

    def set_label(self, label):
        self.label = label

    def has_word(self, word_id):
        return self.words[word_id]

    def get_label(self, node):
        has_word = self.has_word(node.word_id)
        if node.has_child(has_word):
            return self.get_label(node.children[has_word])
        if node.split[has_word][1] > node.split[has_word][2]:
            return 1
        else:
            return 2


def entropy(docs):
    count = defaultdict(int)
```

```
    for doc in docs:
        count[doc.label] += 1

    s = 0.0
    for i in count:
        p = 1.0 * count[i] / len(docs)
        s += -1.0 * p * math.log(p, 2)

    return s


def information_gain(docs, word_id):
    has_word = []
    missing_word = []
    for doc in docs:
        if doc.has_word(word_id):
            has_word.append(doc)
        else:
            missing_word.append(doc)

    has_word_entropy = entropy(has_word)
    missing_word_entropy = entropy(missing_word)

    if APPROACH == 2:
        return entropy(docs) - \
            (1.0 * len(has_word) / len(docs) * has_word_entropy +
             1.0 * len(missing_word) / len(docs) * missing_word_entropy)
    else:
        return entropy(docs) - (0.5 * has_word_entropy + 0.5 * missing_word_entropy)


def load_data(filename):
    docs = {}
    last_index = 0
    with open(filename, 'r') as f:
        for line in f.readlines():
            doc_id, word_id = line.rstrip().split('\t')
```

```python
            doc_id = int(doc_id)
            word_id = int(word_id)

            if doc_id not in docs:
                docs[doc_id] = Doc(doc_id)
                if doc_id - last_index > 1:
                    for i in range(last_index+1, doc_id):
                        docs[i] = Doc(i)

            docs[doc_id].add_word(word_id)
            last_index = doc_id

    return docs


def load_labels(docs, filename):
    with open(filename, 'r') as f:
        for i, line in enumerate(f.readlines()):
            docs[i].set_label(int(line.strip()))


def load_words(filename):
    words = []
    with open(filename, 'r') as f:
        for line in f.readlines():
            words.append(line.strip())

    return words


def render_tree(node, words):
    global RENDER
    if not RENDER:
        return
    from graphviz import Digraph
    dot = Digraph()
    register_node(dot, node, words)
```

```python
    dot.render('decision-tree.gv', view=True)


def register_node(dot, node, words):
    if node.is_terminal:
        label = 1
        if node.split[node.contains][2] > node.split[node.contains][1]:
            label = 2
        dot.node(node.get_id(), str(label))
    else:
        dot.node(node.get_id(), str(words[node.word_id]) + ', info gain: ' + str(node.info_gain)[0:6])

    if True in node.children:
        dot.edge(node.get_id(), node.children[True].get_id(), label='Contains')
        register_node(dot, node.children[True], words)

    if False in node.children:
        dot.edge(node.get_id(), node.children[False].get_id(), label='Not contains')
        register_node(dot, node.children[False], words)


def fill_single_childs(node):
    print(node.get_id())
    if node.has_child(True):
        fill_single_childs(node.children[True])
    if node.has_child(False):
        fill_single_childs(node.children[False])

    print(node.child_count())
    if node.child_count() == 2:
        return
    is_true = node.has_child(True)

    has_word = []
    missing_word = []
```

```python
    for d in node.docs:
        if d.has_word(node.word_id):
            has_word.append(d)
        else:
            missing_word.append(d)

    if node.child_count() == 1:
        if is_true:
            node.add_child(Node(node.word_id, missing_word, node.word_ids, True, False), False)
        else:
            node.add_child(Node(node.word_id, has_word, node.word_ids, True, True), True)
    else:
        node.add_child(Node(node.word_id, missing_word, node.word_ids, True, False), False)
        node.add_child(Node(node.word_id, has_word, node.word_ids, True, True), True)


def create_tree():
    docs = load_data('trainData.txt')
    docs = [docs[i] for i in docs]
    load_labels(docs, 'trainLabel.txt')
    words = load_words('words.txt')
    pq = queue()

    root = Node(None, docs, [])
    for i in range(1, len(words)+1):
        n = Node(i, docs, [i])
        pq.put(Option(root, n, n.word_id, True))

    start = pq.get()
    while not pq.empty():
        try:
            pq.get(False)
        except:
            continue
```

```
        pq.task_done()

    pq.put(start)
    node_count = 1

    while node_count < 200:
        o = pq.get()
        parent_parent_node = o.n1
        parent_node = o.n2
        if parent_parent_node.has_child(o.contains) or parent_node.is_end():
            continue

        print(str(parent_node), parent_node.format_split())
        print(str(o))
        parent_parent_node.add_child(parent_node, o.contains)
        node_count += 2

        has_word = []
        missing_word = []
        print('Parent node word id', parent_node.word_id)
        for d in parent_node.docs:
            if d.has_word(parent_node.word_id):
                has_word.append(d)
            else:
                missing_word.append(d)

        for word in range(1, len(words)+1):
            if word not in parent_node.word_ids:
                n1 = Node(word, has_word, parent_node.word_ids + [word])
                o1 = Option(parent_node, n1, word, True)
                pq.put(o1)

                n2 = Node(word, missing_word, parent_node.word_ids + [word])
                o2 = Option(parent_node, n2, word, False)
                pq.put(o2)

        print('')
```

```python
    fill_single_childs(root.children[True])

    render_tree(root.children[True], words)

    return root.children[True]


def load_test_label(filename):
    label = []
    with open(filename, 'r') as f:
        for i, line in enumerate(f.readlines()):
            label.append(int(line.strip()))
    return label


def test_tree(root):
    correct = 0
    data = load_data('testData.txt')
    label = load_test_label('testLabel.txt')
    for i in range(1, len(label)+1):
        predicted_label = data[i].get_label(root)
        if predicted_label == label[i-1]:
            correct += 1
    print('Accuracy: ' + str(correct * 1.0 / len(label)))

if __name__ == '__main__':
    tree = create_tree()
    test_tree(tree)
```