Contents

1 Basic Test Results

```
Tue Jan 12 16:08:12 IST 2021
Tue Jan 12 16:08:12 IST 2021
Archive:  /tmp/bodek.KRNToq/intro2cs1/ex11/tsviel/final/submission
  inflating: src/ex11.py
  inflating: src/__MACOSX/._ex11.py
10 passed tests out of 10 in test set named 'presubmit'.
result_code    presubmit    10    1
16 passed tests out of 16 in test set named 'diagnose1'.
result_code    diagnose1    16    1
16 passed tests out of 16 in test set named 'diagnose2'.
result_code    diagnose2    16    1
16 passed tests out of 16 in test set named 'diagnose3'.
result_code    diagnose3    16    1
16 passed tests out of 16 in test set named 'diagnose4'.
result_code    diagnose4    16    1
4 passed tests out of 4 in test set named 'calcsuccess'.
result_code    calcsuccess    4    1
4 passed tests out of 4 in test set named 'allillnesses'.
result_code    allillnesses    4    1
--> BEGIN TEST INFORMATION
Test name: pathsillness_11
Module tested: ex11
Function call: paths_to_illness('take-coat')
Expected return value: ([[False, True], [True]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
--> END TEST INFORMATION
**********************************************************************
********************     There is a problem:
********************     The test named 'pathsillness_11' failed.
**********************************************************************
Wrong result, input: ['take-coat']:
expected: ([[False, True], [True]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
actual:   ([], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
result_code    pathsillness_11    wrong    1
--> BEGIN TEST INFORMATION
Test name: pathsillness_12
Module tested: ex11
Function call: paths_to_illness('no-coat')
Expected return value: ([[False, False]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
--> END TEST INFORMATION
**********************************************************************
********************     There is a problem:
********************     The test named 'pathsillness_12' failed.
**********************************************************************
Wrong result, input: ['no-coat']:
expected: ([[False, False]], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
actual:   ([], ('raining', ('take-coat', ('cold', ('take-coat', 'no-coat')))))
result_code    pathsillness_12    wrong    1
9 passed tests out of 11 in test set named 'pathsillness'.
result_code    pathsillness    9    1
3 passed tests out of 3 in test set named 'buildtree'.
result_code    buildtree    3    1
10 passed tests out of 10 in test set named 'optimaltree'.
result_code    optimaltree    10    1
--> BEGIN TEST INFORMATION
Test name: minimize_f0
Module tested: ex11
Function call: minimize(False)
Expected return value: (None, ('b', ('d', None)))
--> END TEST INFORMATION
```

```
**********************************************************************
********************     There is a problem:
********************     The test named 'minimize_f0' failed.
**********************************************************************
Wrong result, input: [False]:
expected: (None, ('b', ('d', None)))
actual:   (None, ('a', (('b', ('d', None)), ('b', ('d', None)))))
result_code    minimize_f0    wrong    1
--> BEGIN TEST INFORMATION
Test name: minimize_f1
Module tested: ex11
Function call: minimize(False)
Expected return value: (None, ('b', ('d', 'e')))
--> END TEST INFORMATION
**********************************************************************
********************     There is a problem:
********************     The test named 'minimize_f1' failed.
**********************************************************************
Wrong result, input: [False]:
expected: (None, ('b', ('d', 'e')))
actual:   (None, ('a', (('b', ('d', 'e')), ('b', ('d', 'e')))))
result_code    minimize_f1    wrong    1
--> BEGIN TEST INFORMATION
Test name: minimize_f4
Module tested: ex11
Function call: minimize(False)
Expected return value: (None, ('a', (('b', ('d', None)), None)))
--> END TEST INFORMATION
**********************************************************************
********************     There is a problem:
********************     The test named 'minimize_f4' failed.
**********************************************************************
Wrong result, input: [False]:
expected: (None, ('a', (('b', ('d', None)), None)))
actual:   (None, ('a', (('b', ('d', None)), ('b', (None, None)))))
result_code    minimize_f4    wrong    1
--> BEGIN TEST INFORMATION
Test name: minimize_f5
Module tested: ex11
Function call: minimize(False)
Expected return value: (None, ('a', (('b', ('d', None)), 'd')))
--> END TEST INFORMATION
**********************************************************************
********************     There is a problem:
********************     The test named 'minimize_f5' failed.
**********************************************************************
Wrong result, input: [False]:
expected: (None, ('a', (('b', ('d', None)), 'd')))
actual:   (None, ('a', (('b', ('d', None)), ('b', ('d', 'd')))))
result_code    minimize_f5    wrong    1
--> BEGIN TEST INFORMATION
Test name: minimize_t0
Module tested: ex11
Function call: minimize(True)
Expected return value: (None, 'd')
--> END TEST INFORMATION
**********************************************************************
********************     There is a problem:
********************     The test named 'minimize_t0' failed.
**********************************************************************
return value could not be expanded
result_code    minimize_t0    recovertree    1
--> BEGIN TEST INFORMATION
Test name: minimize_t1
Module tested: ex11
Function call: minimize(True)
Expected return value: (None, ('b', ('d', 'e')))
--> END TEST INFORMATION
```

```
**********************************************************************
********************     There is a problem:
********************     The test named 'minimize_t1' failed.
**********************************************************************
Wrong result, input: [True]:
expected: (None, ('b', ('d', 'e')))
actual:   (None, ('a', (('b', ('d', 'e')), ('b', ('d', 'e')))))
result_code    minimize_t1    wrong    1
--> BEGIN TEST INFORMATION
Test name: minimize_t3
Module tested: ex11
Function call: minimize(True)
Expected return value: (None, 'd')
--> END TEST INFORMATION
**********************************************************************
********************     There is a problem:
********************     The test named 'minimize_t3' failed.
**********************************************************************
return value could not be expanded
result_code    minimize_t3    recovertree    1
--> BEGIN TEST INFORMATION
Test name: minimize_t4
Module tested: ex11
Function call: minimize(True)
Expected return value: (None, 'd')
--> END TEST INFORMATION
**********************************************************************
********************     There is a problem:
********************     The test named 'minimize_t4' failed.
**********************************************************************
return value could not be expanded
result_code    minimize_t4    recovertree    1
--> BEGIN TEST INFORMATION
Test name: minimize_t5
Module tested: ex11
Function call: minimize(True)
Expected return value: (None, 'd')
--> END TEST INFORMATION
**********************************************************************
********************     There is a problem:
********************     The test named 'minimize_t5' failed.
**********************************************************************
return value could not be expanded
result_code    minimize_t5    recovertree    1
3 passed tests out of 12 in test set named 'minimize'.
result_code    minimize    3    1
TESTING COMPLETED
```

2 ex11.py

```python
#################################################################
# FILE : ex11.py
# WRITER : TSVIEL ZAIKMAN , tsviel , 208241133
# EXERCISE : intro2cs1 ex11 2021
# DESCRIPTION: Traverse on medical records tree Graphs
#################################################################
from collections import Counter
from itertools import combinations

EMPTY = 0  # LENGTH IS 0
MAX = 0  # MAXIMAL VALUE
TYPE_ERROR_MESSAGE = "Either your symptom or record is invalid" \
                     "- Raised TypeError"
VALUE_ERROR_MESSAGE = "A Value Error has been raised"


class Node:
    def __init__(self, data, positive_child=None, negative_child=None):
        self.data = data
        self.positive_child = positive_child
        self.negative_child = negative_child


class Record:
    def __init__(self, illness, symptoms):
        self.illness = illness
        self.symptoms = symptoms


def parse_data(filepath):
    with open(filepath) as data_file:
        records = []
        for line in data_file:
            words = line.strip().split()
            records.append(Record(words[0], words[1:]))
        return records


def is_leaf(tree_node):
    """Return True if node is leaf, false if not None"""
    if tree_node.positive_child is None and tree_node.negative_child is None:
        return True
    return False


def empty(tree_node):
    """Return True if node is None, false if not None"""
    if tree_node.data is None:
        return True
    else:
        return False


class Diagnoser:
    def __init__(self, root: Node):
        self.root = root

    def diagnose(self, symptoms):
        return self._diagnose_helper(symptoms, self.root)

    def _diagnose_helper(self, symptoms, tree_root):
        """
        :param symptoms: a list of symptoms (strings)
        :param tree_root (object)
        :return: name of illness located on leaf
        """
        if is_leaf(tree_root):
            return tree_root.data
        if tree_root.positive_child is not None and tree_root.data in symptoms:
            return self._diagnose_helper(symptoms, tree_root.positive_child)
        if tree_root.negative_child is not None:
            return self._diagnose_helper(symptoms, tree_root.negative_child)

    def calculate_success_rate(self, records):
        """
        :param self: root of choice tree
        :param records: list of records objects
        :return: the ratio between amount of success on records to amount of
        records in total
        """
        records_length = len(records)
        try:
            if records_length == EMPTY:
                raise ValueError
            success = 0  # Counter
            for record in records:
                illness, symptoms = record.illness, record.symptoms
                if self.diagnose(symptoms) == record.illness:
                    success += 1
            return success / records_length
        except ValueError:
            return "Your records list is empty. " + VALUE_ERROR_MESSAGE

    def all_illnesses(self):
        """returns all illnesses"""
        res = self.__sort_all_illnesses_list(
            self.__all_illnesses(self.root, []))
        return res

    def __all_illnesses(self, tree_node, illnesses_list):
        """recursive helper to all_illnesses"""
        if tree_node.positive_child is not None:
            self.__all_illnesses(tree_node.positive_child, illnesses_list)
        if tree_node.negative_child is not None:
            self.__all_illnesses(tree_node.negative_child, illnesses_list)
        if is_leaf(tree_node) and not empty(tree_node):
            # if Leaf and not None
            illnesses_list.append(tree_node.data)
        return illnesses_list

    def __sort_all_illnesses_list(self, illnesses_to_sort):
        """Sort by occurrences and remove duplicates from illnesses list
        using dictionary's properties of orderd keys"""
        if len(illnesses_to_sort) <= 1:
            return illnesses_to_sort

        sorted_ilnesses = sorted(illnesses_to_sort,
                                 key=illnesses_to_sort.count,
                                 reverse=True)
        sorted_and_unique_illnesses_lst = list(dict.fromkeys(sorted_ilnesses))
        return sorted_and_unique_illnesses_lst

    def paths_to_illness(self, illness):
        """Return all path to illness (list of bools)"""
        return self.__paths_to_illness(self.root, illness)

    def __paths_to_illness(self, node, illness):
        """Recursive Helper of path to illness """
        if is_leaf(node):
            # Stop condition - We hit leaf
            if illness is node.data:  # If we reach our illness
                return [[]]
            return []  # If its not our illness
            # Run recursion on the left branch

        negative_path = self.__paths_to_illness(node.negative_child, illness)
        # Run recursion on the right branch
        positive_path = self.__paths_to_illness(node.positive_child, illness)
        paths = []  # Create new list for routes
        for route in negative_path:  # append left direction
            paths.append([False] + route)
        for route in positive_path:  # Append right direction
            paths.append([True] + route)
        return paths  # Return all routes of tree

    def remove_half_nodes(self, tree_node):
        """Remove all half nodes from tree"""
        if tree_node is None:
            return None
        if tree_node.positive_child is not None \
                and tree_node.negative_child is not None:
            tree_node = self.__remove_child_duplications(tree_node)
        tree_node.negative_child = self.remove_half_nodes(
            tree_node.negative_child)  # Recur to left tree
        tree_node.positive_child = self.remove_half_nodes(
            tree_node.positive_child)  # Recur to right tree
        if is_leaf(tree_node):  # We hit leaf
            return tree_node
        if tree_node.negative_child is None:
            next_root = tree_node.positive_child
            prev_root, tree_node = tree_node, None
            del prev_root
            return next_root
        if tree_node.positive_child is None:
            next_root = tree_node.negative_child
            prev_root, tree_node = tree_node, None
            del prev_root
            return next_root
        return tree_node

    def __remove_child_duplications(self, node):
        """
        :param node: a give node object which has 2 children
        :return: a node only with the positive child
        """
        if node.positive_child.data == node.negative_child.data:
            node = node.positive_child
        return node

    def __delete_leaves(self, tree_node, val):
        """
        :param tree_node: a given tree object
        :param val: a value we wish to remove
        :return: remove the leaves with the given value
        """
        if tree_node is None:
            return
        tree_node.negative_child = self.__delete_leaves(
            tree_node.negative_child, val)
        tree_node.positive_child = self.__delete_leaves(
            tree_node.positive_child, val)

        if tree_node.data == val and is_leaf(tree_node):
            return
        return tree_node

    def minimize(self, remove_empty=False):
        """The function minimize the tree and removes unnesseary path"""
        if remove_empty:
            self.__delete_leaves(self.root, None)
        self.remove_half_nodes(self.root)
        return


def _build_tree_helper(records):
    """helper of build_tree for counting"""
    if len(records) is EMPTY:
        return None
    illnesses = []
    for i in range(len(records)):
        illnesses.append(records[i].illness)
    counts = Counter(illnesses)
    #  return sorted list by prevalence
    return sorted(counts, key=counts.get, reverse=True)[0]


def _build_tree(records, symptoms):
    """
    recursive function to build a tree
    :param records: list of record objects
    :param symptoms: list of strings representing symptoms of illness
    :return: Diagnoser Object for the tree we build
    """
    if len(symptoms) == EMPTY:
        return Node(_build_tree_helper(records))

    first_symptom, rest_symptoms = symptoms[0], symptoms[1:]

    positive_path = _build_tree([record for record in records if
                                 first_symptom
                                 in record.symptoms], rest_symptoms)
    negative_path = _build_tree([record for record in records if
                                 first_symptom
                                 not in record.symptoms], rest_symptoms)

    final_tree = Node(first_symptom, positive_path, negative_path)

    return final_tree


def build_tree(records, symptoms):
    """
    :param records: a list of records(objects)
    :param symptoms: a list of symptoms(strings)
    :return: Diagnoser object based on the built_tree
    """
    try:
        symptoms_are_strings = [type(item) == str for item in symptoms]
        records_are_record = [type(record) == Record for record in records]
        if not all(symptoms_are_strings) and all(records_are_record):
            # Logic of statement is based on de-morgan law
            raise TypeError
        return Diagnoser(_build_tree(records, symptoms))
    except TypeError:
        return TYPE_ERROR_MESSAGE


def _optimal_tree(records, symptoms, depth):
    """
    Returns the optimal tree for different subsets of symptoms with size of
    depth. The function iterates on all of the subsets of symptoms in size
    and depth for each time we create a tree Node.
    The function gets the rates for success, update the maximal value(tuple)
    :param records: list of Records
    :param symptoms: list of symptoms
    :param depth: the size of the subset of symptoms
    :return: an optimal tree node object
    """
    max_val = None, MAX
    combs = combinations(symptoms, depth)  # all combinations of symptoms
    # and depth using itertools lib
    for comb in combs:
        diagnostic = build_tree(records, comb)  # Build subset tree of combs
        #  Checks the success rates for records
        success_rates = diagnostic.calculate_success_rate(records)
        if success_rates >= max_val[1]:
            # case the success rates of this tree are higher than the
            # maximal_value
            max_val = diagnostic.root, success_rates  # update it.

    if type(max_val[0]) is not Node:
        return Node("")  # Case we fail to produce node

    return max_val[0]


def optimal_tree(records, symptoms, depth):
    """
    :param records: a list of objects of Record type
    :param symptoms: a list of strings representing symptoms
    :param depth: the depth of the tree required
    :return: a Diagnostic based on the optimal tree
    """
    try:
        optimal_tree_node = _optimal_tree(records, symptoms, depth)
        depth_property = (len(symptoms) >= depth >= 0)
        contains_duplicates = any(
            symptoms.count(element) > 1 for element in symptoms)

        if not depth_property or contains_duplicates:
            raise ValueError

        symptoms_are_strings = [type(item) == str for item in symptoms]
        records_are_record = [type(record) == Record for record in records]

        if not all(symptoms_are_strings) and all(records_are_record):
            raise TypeError

        return Diagnoser(optimal_tree_node)
    except ValueError:
        return "Either The symptoms doesn't meet the depth " \
                             "property or it contains duplicates. " + \
                             VALUE_ERROR_MESSAGE
    except TypeError:
        return TYPE_ERROR_MESSAGE


if __name__ == "__main__":
    pass
```