# Contents

# 1 Basic Test Results

```
Starting tests...
Thu Jan  7 18:24:32 IST 2021
49b30658baf15cb0e34cfbd62d551700538d2562  -


Archive:  /tmp/bodek.eeZu0e/intro2cs1/ex11/itaygil135/presubmission/submission
  inflating: src/ex11.py


Running presubmit code tests...
10 passed tests out of 10 in test set named 'ex11'.
result_code    ex11    10    1
Done running presubmit code tests

Finished running the presubmit tests

Additional notes:

The presubmit tests check only for the existence of the correct function names.
Make sure to thoroughly test your code.

```

# 2 ex11.py

```
1  ######################################################################
2  # FILE : ex11.py
3  # WRITER : itai kahana, itaygil135 , 316385962
4  # EXERCISE : intro2cs2 ex11 2020
5  # DESCRIPTION: This program build and optimize decisions trees
6  # STUDENTS I DISCUSSED THE EXERCISE WITH: no one.
7  # WEB PAGES I USED: nothing.
8  # NOTES: ...
9  ######################################################################
10 import itertools
11
12
13 class Node:
14     """
15     This class implement the nodes action at the tree
16     """
17     def __init__(self, data, positive_child=None, negative_child=None):
18         self.data = data
19         self.positive_child = positive_child
20         self.negative_child = negative_child
21
22     def set_most_common_illness(self):
23         """
24         this method scan all leaves, for each leaf - set the most common
25         illness from its possible illness
26         :return: None
27         """
28         if self.positive_child:
29             self.positive_child.set_most_common_illness()
30             self.negative_child.set_most_common_illness()
31         else:
32             if len(self.data) == 0:
33                 self.data = None
34             else:
35                 leaf_all_illness = {}
36                 for item in self.data:
37                     if item in leaf_all_illness:
38                         leaf_all_illness[item] = leaf_all_illness[item] + 1
39                     else:
40                         leaf_all_illness[item] = 1
41                 self.data = []
42                 illness_dict_to_sorted_list(leaf_all_illness, self.data)
43                 self.data = self.data[0]
44
45     def diagnose_sick(self, symptoms):
46         """
47         This method diagnose a illness based on a decision tree and list
48         of symptoms
49         :param symptoms: list of symptoms
50         :return: the diagnosed illness
51         """
52         if not self.positive_child:
53             return self.data
54         if self.data in symptoms:
55             return self.positive_child.diagnose_sick(symptoms)
56         else:
57             return self.negative_child.diagnose_sick(symptoms)
58
59     def build_tree_helper(self, symptoms, i):
```

```
        """
        this method build the tree in a recursive way and in pre-order
        :param symptoms: list of symptoms
        :param i: index of the symptom that should be added to the tree
        :return: None
        """
        if i < len(symptoms):
            self.positive_child = Node(symptoms[i])
            self.negative_child = Node(symptoms[i])
            self.positive_child.build_tree_helper(symptoms, i+1)
            self.negative_child.build_tree_helper(symptoms, i+1)
        else:
            self.positive_child = Node([])
            self.negative_child = Node([])

    def minimize_tree(self, remove_empty):
        """
        this method minimize a given decision tree by removing nodes that
        have no impact on the tree making decision. the removed nodes may be:
        1) nodes that their two children are equal
        2) nodes that one of their children does not diagnose any illness.
        :param remove_empty: a boolean parameter. in case of False, only
        condition #1 above will be applied. in case of True, condition #2
        will be applied as well.
        :return: None
        """
        if self.positive_child.positive_child:
            self.positive_child.minimize_tree(remove_empty)
        if self.negative_child.negative_child:
            self.negative_child.minimize_tree(remove_empty)
        else:
            if self.positive_child.data == self.negative_child.data:
                self.data = self.positive_child.data
                self.positive_child = None
                self.negative_child = None
            elif remove_empty:
                if not self.positive_child.data:
                    self.data = self.negative_child.data
                    self.positive_child = None
                    self.negative_child = None
                elif not self.negative_child.data:
                    self.data = self.positive_child.data
                    self.positive_child = None
                    self.negative_child = None

    def create_all_illness_dict(self, illness_dictionary):
        """
        This method scan a decision tree and create a dictionary of all
        illness appear at the tree. the dictionary keys are the illness name
        and the values are the number each illness appear at the tree
        :param illness_dictionary: the dictionary to build
        :return: None
        """
        if self.positive_child:
            self.positive_child.create_all_illness_dict(illness_dictionary)
            self.negative_child.create_all_illness_dict(illness_dictionary)
        else:
            if self.data:
                if self.data in illness_dictionary:
                    illness_dictionary[self.data] = \
                        illness_dictionary[self.data] + 1
                else:
                    illness_dictionary[self.data] = 1

    def paths_to_illness_helper(self, all_paths, moving_lst, illness):
        """
        this method find all possible path to a given illness
        :param all_paths: nested list. each internal list is a path to the
```

```
        given illness
        :param moving_lst: a temporary list used to find the paths
        :param illness: the illness to find the paths to
        :return: the nested list contains all the paths
        """
        if self.positive_child:
            moving_lst.append(True)
            self.positive_child.paths_to_illness_helper(all_paths, moving_lst, illness)
            moving_lst.append(False)
            self.negative_child.paths_to_illness_helper(all_paths, moving_lst, illness)
        else:
            if self.data == illness:
                all_paths.append(moving_lst[:])
        if len(moving_lst) > 0:
            moving_lst.pop()
        return all_paths


class Record:
    """
    this class reflect the records off diagnosed cases
    """
    def __init__(self, illness, symptoms):
        self.illness = illness
        self.symptoms = symptoms


class Diagnoser:
    """
    this class implement the diagnoster methods. the diagnoster is a decision
    tree that diagnose an illness based on a given symptoms
    """
    def __init__(self, root: Node):
        self.root = root

    def diagnose(self, symptoms):
        """
        This method diagnose an illness based on a given symptoms
        :param symptoms: list of symptoms to diagnose based on them
        :return: the diagnosed illness
        """
        return self.root.diagnose_sick(symptoms)

    def calculate_success_rate(self, records):
        """
        this method calculate the success rate of a decision tree based on
        list of diagnosed cases
        :param records: list of diagnosed cases
        :return: the success rate of the tree
        """
        if len(records) == 0:
            raise ValueError('there is no symptoms')
        passed = 0
        for rec in records:
            if self.diagnose(rec.symptoms) == rec.illness:
                passed = passed + 1
        return passed/len(records)

    def all_illnesses(self):
        """
        This method return a list of all illness appears in the
        tree. the list is sorted from the most common illness to less common
        one
        :return:
        """
        illness_dict = dict()
        self.root.create_all_illness_dict(illness_dict)
        sorted_list = []
```

```
196         for i in range(len(illness_dict)):
197             illness_dict_to_sorted_list(illness_dict, sorted_list)
198         return sorted_list
199 
200     def paths_to_illness(self, illness):
201         """
202         This method scan a tree and return a list of all possible paths to a
203         given illness
204         :param illness: the illness to find the paths to.
205         :return: list of all paths to the given illness
206         """
207         all_paths = []
208         moving_lst = []
209         return self.root.paths_to_illness_helper(all_paths,
210                                                  moving_lst,
211                                                  illness)
212 
213     def minimize(self, remove_empty=False):
214         """
215         this method minimize a given decision tree by removing nodes that
216         have no impact on the tree making decision. the removed nodes may be:
217         1) nodes that their two children are equal
218         2) nodes that one of their children does not diagnose any illness.
219         Note that this method actually send the tree root to the Node class to
220         actually implement the minimize logic
221         :param remove_empty: a boolean parameter. in case of False, only
222         condition #1 above will be applied. in case of True, condition #2
223         will be applied as well.
224         :param remove_empty:
225         :return:
226         """
227         if self.root.positive_child:
228             self.root.minimize_tree(remove_empty)
229 
230 
231 
232 # module functions
233 def parse_data(filepath):
234     with open(filepath) as data_file:
235         records = []
236         for line in data_file:
237             words = line.strip().split()
238             records.append(Record(words[0], words[1:]))
239         return records
240 
241 
242 def illness_dict_to_sorted_list(dictionary, sorted_list):
243     """
244     this function scan a given dictionary, and create a sorted list of all
245     dictionary keys, sorted by their values
246     :param dictionary: dictionary to be soreted
247     :param sorted_list: an empty list that is updated to store the dictionary
248     keys soreted by their values
249     :return: None
250     """
251     max_value = 0
252     for key in dictionary:
253         if max_value < dictionary[key]:
254             max_value = dictionary[key]
255     for key in dictionary:
256         if dictionary[key] == max_value:
257             sorted_list.append(key)
258             dictionary.pop(key)
259             break
260 
261 
262 def set_all_possible_illness(rec, root):
263     """
```

```
    set all possible illness to each leaf at the tree
    :param rec: list of illness cases, including list of symptoms and
    diagnosed illness
    :param root: root of the decision tree
    :return:
    """
    if type(root.data) != list:
        if root.data in rec.symptoms:
            set_all_possible_illness(rec, root.positive_child)
        else:
            set_all_possible_illness(rec, root.negative_child)
    else:
        root.data.append(rec.illness)


def build_tree(records, symptoms):
    """
    This function build a decision tree based on list of illness cases and
    list of symptoms
    :param records: list of illness cases, including list of symptoms and
    diagnosed illness
    :param symptoms: full list of symptoms to build the tree from
    :return: Diagnoster object that include the root of the built tree
    """
    for item in records:
        if type(item) != Record:
            raise TypeError("there is an item or more inside records which is not type Record")

    if len(symptoms) == 0:
        root = Node(None)
        diagnoser = Diagnoser(root)
        return diagnoser

    for item in symptoms:
        if type(item) != str:
            raise TypeError("there is an item or more inside symptoms which is not type string")
    # build tree based on the symptoms. update the nodes data only
    root = Node(symptoms[0])
    root.build_tree_helper(symptoms, 1)
    # set all possible illness to each leaf at the tree
    for rec in records:
        set_all_possible_illness(rec, root)
    # for each leaf - set the most common illness from its possible illness
    root.set_most_common_illness()
    diagnoser = Diagnoser(root)
    return diagnoser


def optimal_tree(records, symptoms, depth):
    """
    This function create sub-lists of symptoms, at the size depth, build trees
    for each sub-list and return the tree with the best rate.
    :param records: list of illness cases, including list of symptoms and
    diagnosed illness
    :param symptoms: full list of symptoms to build the tree from
    :param depth: the size of the sub-list
    :return: the oprtimal tree
    """
    validate_input(records, symptoms, depth)
    symptomps_combinations = list()
    for comb in itertools.combinations(symptoms, depth):
        symptomps_combinations.append(comb)

    best_diagnoser = Diagnoser(Node(None))
    if len(records) > 0:
        for comb in symptomps_combinations:
            best_rate = best_diagnoser.calculate_success_rate(records)
            current_tree = build_tree(records, comb)
```

```
            if best_rate < current_tree.calculate_success_rate(records):
                best_diagnoser = current_tree
    return best_diagnoser

def validate_input(records, symptoms, depth):
    if depth < 0 or depth > len(symptoms):
        raise ValueError("invalid depth")
    symptoms_set = set()
    for item in symptoms:
        symptoms_set.add(item)
    if len(symptoms) > len(symptoms_set):
        raise ValueError("double symptom")
    for item in symptoms:
        if type(item) != str:
            raise TypeError("there is an item or more inside symptoms which is not type string")
    for item in records:
        if type(item) != Record:
            raise TypeError("there is an item or more inside records which is not type Record")

if __name__ == "__main__":

    # Manually build a simple tree.
    #                cough
    #          Yes /       \ No
    #        fever           healthy
    #   Yes /     \ No
    # covid-19   cold

    '''flu_leaf = Node("covid-19", None, None)
    cold_leaf = Node("cold", None, None)
    inner_vertex = Node("fever", flu_leaf, cold_leaf)
    healthy_leaf = Node("healthy", None, None)
    root = Node("cough", inner_vertex, healthy_leaf)

    diagnoser = Diagnoser(root)

    # Simple test
    diagnosis = diagnoser.diagnose(["cough"])
    if diagnosis == "cold":
        print("Test passed")
    else:
        print("Test failed. Should have printed cold, printed: ", diagnosis)'''

    rec1 = Record("covid-19", ["fever", "cough"])
    rec4 = Record("covid-19", ["fever", "cough"])
    rec2 = Record("no", ["fever", "cough"])
    rec3 = Record("no", [])
    record_list = [rec3, rec2, rec1, rec4,8]

    # record_list = parse_data("./Data/tiny_data.txt")

    # rate = diagnoser.calculate_success_rate(record_list)

    # diagnoser.all_illnesses()
    symptoms_list = ['koko', 'fever ', 'fatigue', 'headache'
        , 'nausea', 'cough',
                'sore_throat', 'muscle_ache', 'congestion', 'irritability',
                'rigidity']
    '''
    diagnoser = build_tree(record_list, symptoms)

    illness_list = diagnoser.all_illnesses()
    for illness in illness_list:
         print(illness,"  ", diagnoser.paths_to_illness(illness))
    '''

    optimal_diagnoser = optimal_tree(record_list, symptoms_list, 2)
    print(type(optimal_diagnoser))
```

```
400         optimal_diagnoser.minimize(True)
```