HACETTEPE UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

BBM 103 ASSIGNMENT 2 REPORT

Ahmet İkbal Kayapınar

2210356043

23/11/2022

Analysis

A simple CDSS program is required in the assignment. It is desired that the CDSS program can store various data about each patient, perform various operations on it, and give an output after each operation.

The program will store patients' name, disease, diagnostic accuracy, incidence of the disease in the community, possible treatment and treatment risk data.

The program will detect the commands in the entered data and run the correct function.

A new patient can be added with a single command or an existing patient can be removed with another command.

The program will be able to calculate the probability that patients are actually sick.

Using the probability it has calculated, the program will be able to advise on whether or not the patient should be treated.

The program will be able to show the information of current patients in a list.

The program will give an output after each operation.

Design

A main multi dimensinal list called patient_list is required to keep the data of the patients.

Functions named *create(), remove(), probability(), recommendation(), list(), read(), save()* will be written.

The *create()* function will take the list containing the patient's information as an argument and add the patient to the list.

The patient's name will be sent as an argument to the *remove(), probability(), recommendation()* functions. The *remove()* function will delete the patient from the list, the *probability()* function will calculate the probability that patient is really sick, and the *recommendation()* function will suggest a treatment.

The *list()* function will add the current patient list in tabular form to the output file.

The *read()* function will take the filename as an argument and read that file and save it in a variable.

The *save()* function will take the filename and output as arguments and save the output to the selected file.

The program will add the output of each command to a variable named output so that all outputs can be printed without any problems.

Since each line contains commands and information, first of all, it will be ensured that each line is processed sequentially with the help of a loop.

For each line in the loop, the spaces in the line will be deleted and a list named *l* will be created and the information will be separated and written to this list with the command.

Conditional statements will check the first element of the list, the command, to find which function should be used.

In case of using the create command, the program will check if the patient is in the list and if it is, it will use the function and add its output. If there is no patient it will add the error message to the output.

If the remove, probability or recommendation commands are used, the program will set a default error output, if it can find the patient, it will run the function and replace this output with the result, if it cannot find the patient, it will add the default output.

If the list command is used, it will print the current list.

The save function used at the end of the code will complete the program by printing the output to the selected file.

Solutions

Step 1: Functions

```
def create(patient):
    patient_list.append(patient)
```

Function that used to add new patients into the program. As an argument, it takes the patient's data as a list and adds it to the end of the patient list.

```
def remove(patient):
    patient_list.remove(patient)
```

Function that used to delete the specified patient in the patient list. As an argument, it takes the patient's data as a list and deletes the patient from the patient list.

```
def list():
    global output
    output += "Patient\tDiagnosis\tDisease\t\t\tDisease\t\tTreatment\t\tTreatment\n"
    output += "Name\tAccuracy\tName\t\t\tIncidence\tName\t\t\tRisk\n" + "-" * 73 + "\n"
    for patient in patient_list:
        output += "{}{}{:.2f}%\t\t{}{}{}\t{}{}{:g}%\n".format(patient[0],
        "\t" * (2 - len(patient[0]) // 4),float(patient[1]) * 100,
        patient[2],"\t" * (4 - len(patient[2]) // 4),patient[3],patient[4],
```

*"\t" * (4 - len(patient[4]) // 4),float(patient[5]) * 100,)*

Function that used to tabulate the patient list and add it to the output. It uses the *output* variable in the global frame. Firstly, it adds the fixed lines of the table to the output. Then it uses a for loop to add each patient in the patient list to the output. In order to convert the 2nd order diagnosis accuracy and 6th order treatment risk to percentile, it first converts it to a float number with the *float()* function and then multiplies it by 100. The number of tabs to be used must be determined correctly so that the width of the column containing the patient's name, disease name, treatment name, which may have variable lengths, is the same in each row. In order to achieve this, it subtracts integer part of quarter of the variable's length from the number of tabs as wide as the width, and adds the number of tabs found after the variable so that the width is equal.

```
def probability(patient):
    incidence = eval(patient[3])
    accuracy = float(patient[1])
    true_positives = incidence * accuracy
    false_positives = (1 - incidence) * (1 - accuracy)
    prob = round(true_positives / (true_positives + false_positives),4)
    return prob
```

Function that used to calculate the probability that the patient is actually sick. It takes the incidence data in the patient list and completes the division with the *eval()* function. It also converts the diagnostic accuracy data in the patient list from string to float, making it ready for operation. In order to find the false positive rate in the entire population, it is necessary to divide the true positives by all people who test positive. Those who test positive are also true positives and the sum of those who test positive even though they are negative. The function multiplies the incidence in the population by the accuracy of the test to find true positives. To find false positives, it multiplies the proportion of the population that is actually negative by the margin of error of the test. Rounds the result to 4 digits after the decimal point and returns the result of the function.

```
def recommendation(patient):
    global recom_output
    if(probability(patient)>float(patient[5])):
        recom_output = "System suggests {} to have the treatment.\n".format(patient[0])
    else:
        recom_output = "System suggests {} NOT to have the treatment.\n".format(patient[0])
```

Function that adds a recommendation on whether to treat the patient or not to the output. It uses the variable in the global frame to manipulate the output of the recommendation command named *recom_output*. It compares the probability calculated in the *probability()*

function with the patient's treatment risk. If the probability is greater, it adds the suggested treatment to the output file. It adds that it is not recommended in the opposite case.

```
def read(file_name):
    reading_file_name = file_name
    reading_file_path = os.path.join(current_dir_path, reading_file_name)
    with open(reading_file_path, "r") as f:
        data = f.readlines()
    return data
```

A function that reads the data in the selected file and saves it to a variable named *data* and returns the variable. Determines the file to be read using the *os* library. With *open(reading_file_path, "r")*, it opens the file for reading and closes it after the process is finished. Using the *readlines()* method, it reads the contents of the file and adds it to the variable named *data*. It returns *data*.

```
def save(file_name, output):
    writing_file_name = file_name
    writing_file_path = os.path.join(current_dir_path, writing_file_name)
    with open(writing_file_path,"w") as f:
        f.writelines(output)
```

Function that saves the output to the selected file. Determines the file to be saved using the *os* library. With *open(writing_file_path,"w")* it opens the file for overwriting and closes it after the process is finished. It saves the output to the file using the *writelines()* method.

Step 2: Converting Data to Lists

```
for line in data:
    new_line = line.strip()
    l = new_line.split(" ",1)
```

The *data* variable is a long string of many lines. To create a list where commands and data are separate elements, first it puts it in a loop where it separates each line. It deletes the space under each line for the next line with the *strip()* method. In order to create a list, it uses the space between the command and the name after it and splits it into a two-dimensional list named *l* using the *split()* method.

Step 3: Conditional Statements

Each if condition causes a different command written to the input file to run. Each if looks at the first element containing the command in the *l* list and if it matches, it executes the code inside. If not, this continues until it find the matching if condition.

```
if(l[0]=="create"):
    new_patient = l[1].split(", ")
    if(new_patient not in patient_list):
        create(new_patient)
        output += "Patient {} is recorded.\n".format(new_patient[0])
    else:
        output += "Patient {} cannot be recorded due to duplication.\n".format(
new_patient[0])
```

It is the if condition of the create command. Since there is more than one data in the data section only in the inputs entered with the create command, a list needs to be created. In order to do this, the information of the new patient to be added is transferred to the nested list with the *split()* method. If condition checks that the newly added patient is not already listed. If it doesn't exist, it adds it to the list using the *create()* function and adds its output to the *output* variable. If the patient is already in the list, he adds the statement indicating that the operation could not be performed to the *output* variable.

```
elif(l[0]=="remove"):
    removes_output = "Patient {} cannot be removed due to absence.\n".format(l[1])
    for patient in patient_list:
        if(l[1]==patient[0]):
            remove(patient)
            removes_output = "Patient {} is removed.\n".format(l[1])
            break
    output += removes_output
```

It is the if condition of the remove command. It uses a default output named *removes_output*. It divides the patient list into its elements with a loop and compares each patient's name with the patient's name in the given input, with the if condition. If it finds the patient, it deletes the patient using the *remove()* function and replaces the default negative output with an output that indicates the patient has been deleted. If it cannot find the patient, the default negative output is added to the *output* variable.

```
elif(l[0]=="list"):
    list()
```

It is the if condition of the list command. It uses the *list()* function to add the list to the *output* variable.

```
elif(l[0]=="recommendation"):
    recom_output = "Recommendation for {} cannot be calculated due to absence.\n".format(l[1])
    for patient in patient_list:
      if(l[1]==patient[0]):
        recommendation(patient)
        break
    output += recom_output
```

It is the if condition of the recommendation command. It uses a default output called *recom_output*. It divides the patient list into its elements with a loop and compares each patient's name with the patient's name in the given input, with the if condition. If it finds the patient, it replaces the default negative output with an output relevant to the treatment recommendation for the patient, using the *recommendation()* function. If it cannot find the patient, the default negative output is added to the *output* variable.

```
elif(l[0]=="probability"):
    probs_output = "Probability for {} cannot be calculated due to absence.\n".format(l[1])
    for patient in patient_list:
      if(l[1]==patient[0]):
        probs_output = "Patient {} has a probability of {:g}% of having breast cancer.\n".format(l[1],probability(patient)*100)
        break
    output += probs_output
```

It is the if condition of the probability command. It uses a default output called *probs_output*. It divides the patient list into its elements with a loop and compares each patient's name with the patient's name in the given input, with the if condition. If it finds the patient, it replaces the output with an expression that represents the probability calculated by the *probability()* function. If it cannot find the patient, the default negative output is added to the *output* variable.

Programmer's Catalogue

It took my one hour to analyze and two hours to design. It took my six hours to implement the design. I developed my program step by step and implemented my design ideas in order to move on to the next after solving the problems of each. While implementing, the part of separating the commands and data and taking them to the functions correctly bothered me

more than I expected. I spent at least two hours testing, including testing while implanting. The fact that the tab equality my computer used was different from the one given in the assignment cost me extra time and caused me to do extra tests. It took my at least six hours to write the report.

All my code is here:

```
import os


current_dir_path = os.getcwd()


def create(patient):
    patient_list.append(patient)


def remove(patient):
    patient_list.remove(patient)


def list():
    global output
    output += "Patient\tDiagnosis\tDisease\t\t\tDisease\t\tTreatment\t\tTreatment\n"
    output += "Name\tAccuracy\tName\t\t\tIncidence\tName\t\t\tRisk\n" + "-" * 73 + "\n"
    for patient in patient_list:
        # It ensures that each line is equal width where there are variable-length inputs.
        # Since \t makes the line width a multiple of 4.
        # It divides the variable lengths by 4 to find the desired number of \t.
        output += "{}{}{:.2f}%\t\t{}{}{}\t{}{}{:g}%\n".format(patient[0],
        "\t" * (2 - len(patient[0]) // 4),float(patient[1]) * 100,
        patient[2],"\t" * (4 - len(patient[2]) // 4),patient[3],patient[4],
        "\t" * (4 - len(patient[4]) // 4),float(patient[5]) * 100,)


def probability(patient):
    incidence = eval(patient[3])
    accuracy = float(patient[1])
```

```
    true_positives = incidence * accuracy
    false_positives = (1 - incidence) * (1 - accuracy)
    prob = round(true_positives / (true_positives + false_positives),4)
    return prob

def recommendation(patient):
    global recom_output
    if(probability(patient)>float(patient[5])):
        recom_output = "System suggests {} to have the treatment.\n".format(patient[0])
    else:
        recom_output = "System suggests {} NOT to have the treatment.\n".format(patient[0])

# It opens, reads and saves all data of the chosen file's
def read(file_name):
    reading_file_name = file_name
    reading_file_path = os.path.join(current_dir_path, reading_file_name)
    with open(reading_file_path, "r") as f:
        data = f.readlines()
    return data

# It saves final output of the program to chosen file
def save(file_name, output):
    writing_file_name = file_name
    writing_file_path = os.path.join(current_dir_path, writing_file_name)
    with open(writing_file_path,"w") as f:
        f.writelines(output)

# Every command adds their output statement into the output variable
output = ""
```

```
data = read("doctors_aid_inputs.txt")

patient_list = []

for line in data:
    new_line = line.strip()    # Strip function erases every line's spaces
    # It separates the information and commands in each line and creates a list
    l = new_line.split(" ",1)
    # If conditions control which command the command in the list is
    if(l[0]=="create"):
        new_patient = l[1].split(", ") # It seperates patient's infos and creates another list
        if(new_patient not in patient_list):
            create(new_patient)
            output += "Patient {} is recorded.\n".format(new_patient[0])
        else:
            output += "Patient {} cannot be recorded due to duplication.\n".format(
new_patient[0])
    elif(l[0]=="remove"):
        # If it can't find name of the patient it uses the default negative statement
        removes_output = "Patient {} cannot be removed due to absence.\n".format(l[1])
        # It checks if the patient is on the list
        for patient in patient_list:
            if(l[1]==patient[0]):
                remove(patient)
                removes_output = "Patient {} is removed.\n".format(l[1])
                break
        output += removes_output
    elif(l[0]=="list"):
        list()
    elif(l[0]=="recommendation"):
```

```
        # If it can't find it uses the default negative statement
        recom_output = "Recommendation for {} cannot be calculated due to
absence.\n".format(l[1])
        # It checks if the patient is on the list
        for patient in patient_list:
            if(l[1]==patient[0]):
                recommendation(patient)
                break
        output += recom_output
    elif(l[0]=="probability"):
        # If it can't find it uses the default negative statement
        probs_output = "Probability for {} cannot be calculated due to absence.\n".format(l[1])
        # It checks if the patient is on the list
        for patient in patient_list:
            if(l[1]==patient[0]):
                probs_output = "Patient {} has a probability of {:g}% of having breast
cancer.\n".format(l[1],probability(patient)*100)
                break
        output += probs_output

save("doctors_aid_outputs.txt", output)
```

While writing my code, I took care to be flexible. It can be used for a similar management system with minor changes.

The *save()* and *read()* functions work depending on the file name, so file names or locations can change.

I have tried to write each if condition functions step by step and based on the abstraction principle. Therefore, it is possible to use the code for different purposes by changing only a part of it.

The most likely use case is its use as a simple database management system in different industries.

User Manuel

The program uses "*doctors_aid_inputs.txt*" file as input and "*doctors_aid_outputs.txt*" file as output file. Make sure your input and output files are in the same folder and with the correct name.

You will write the commands into the input file. Write each command on a separate line and in the order you want it to run.

To add a new patient, you need to write "*create [patient's name, diagnosis accuracy, disease name, disease incidence, treatment name, treatment risk]*".

To delete a patient, you need to write "*remove [patient's name]*".

To see the patient list, you need to write "*list*".

To see the probability that the patient is actually sick, you need to write "*probability [patient's name]*".

You need to write "*recommendation [patient's name]*" to get advice about the patient's treatment.

After filling the input file, you can run the program.

After changing the selected location in your console to the location of your file, you can run the program by typing *python3 Assignment2.py*.

The results will be in the output file.

Program Restrictions

If text is entered where numbers are required, the program will not work properly.

If the entered variables' length are longer than supported, errors may occur while generating the list.

If a number with a zero denominator is written to the disease incidence, the program will give an error.

Grading Table

Evaluation	Points	Evaluate Yourself / Guess Grading
Indented and Readable Codes	5	5
Using Meaningful Naming	5	5
Using Explanatory Comments	5	5
Efficiency (avoiding unnecessary actions)	5	5
Function Usage	25	25
Correctness	35	35
Report	20	20