2022

# BBM 103 Assignment 2 Report

HACETTEPE UNIVERSITY – DEPARTMENT OF COMPUTER ENGINEERING

BARIŞ YILDIZ – 2210356107 – 24.11.2022

# Assignment 2 Report

## Table of Contents

# Analysis

We are given an input file named *"doctors_aid_inputs.txt"* that contains some lines that have certain commands in them. Our goal is to read this file line-by-line, determine the command of each line, and output proper results for each line in order. Each output will then be written in a text file called *"doctors_aid_outputs.txt"*. Every possible format of input lines and what the program should do when encountering them is given below:

**Create Command**: *create* [patient name], [diagnosis accuracy], [disease name] [disease incidence], [treatment name], [treatment risk]

This line will tell the program to store the values of *patient name*, *diagnosis accuracy*, *disease incidence*, *treatment name*, *treatment risk* of a patient to a list, then add that list to a multi-dimensional list called "patients" that is going to act as a database. We are going to refer to "patients" as the database of our program from now on. Then, the program is going to output "Patient is recorded."

However, if the database already contains information about this patient, the program is only going to output "Patient cannot be recorded due to duplication." and is not going to add anything into the database.

**Remove Command:** *remove* [patient name]

This line will tell the program to remove all information provided about a patient in the create command from the database and output "Patient is removed." If the information about this patient was not present in the database in the first place, output will be "Patient cannot be removed due to absence" and the program will not perform any removing operation.

**Probability Command:** *probability* [patient name]

This line will tell the program to calculate the probability of a patient having the disease which he/she has been diagnosed with, which is provided in the "create" line. This calculation is going to be made using the values of diagnosis accuracy and disease incidence of the patient given in the "create" line and by using the following formula:

**Cases** = numerator of incidence
**Total population** = denominator of incidence
**Accuracy =** diagnosis accuracy information about the patient

$$probability = \frac{cases\ \times accuracy}{cases\ \times accuracy + (total\ population - cases) \times (100 - accuracy)}$$

The output will be "Patient has a probability of … of having … However, if information about this patient doesn't exist in the database, the output will be "Probability for patient cannot be calculated due to absence." and the program will not attempt to make any calculations.

**List Command:** *list*

This line will tell the program to create a table containing contents of the database, like the table below, and output it.

| Patient Name | Diagnosis Accuracy | Disease Name | Disease Incidence | Treatment Name | Treatment Risk |
|---|---|---|---|---|---|
| Hayriye | 99.90% | Breast Cancer | 50/100000 | Surgery | 40% |
| Deniz | 99.99% | Lung Cancer | 40/100000 | Radiotherapy | 50% |
| Toprak | 98.00% | Prostate Cancer | 21/100000 | Hormonotherapy | 20% |
| Hypatia | 99.75% | Stomach Cancer | 15/100000 | Immunotherapy | 4% |
| Pakiz | 99.97% | Colon Cancer | 14/100000 | Targeted Therapy | 20% |

**Recommendation Command:** *recommendation* [patient name]

This line will make the program compare the probability of the patient having the disease to the risk value of the suggested treatment. The output will either recommend the patient to have the treatment, or not recommend it depending on which of the treatment risk and probability of having the disease is greater. These values will never be given as equal values, so we don't have to take this condition into account. Also, if information about this patient doesn't exist in the database, the output will be "Recommendation for cannot be calculated due to absence." and the program will not do any comparisons.

# Design

## *Solution of The Problem*

- **First things first**, we should define the database, a multi-dimensional list called "patients", where we are going to store all information about every patient as a list.
- **Secondly**, the program needs to access each input line located in *"doctors_aid_inputs.txt"*.
- **Thirdly**, the program needs to iterate over each input line. In each iteration, the program needs to find the command of the line and call a function associated with that command. The function that has been called needs to return a proper output. At the end of each iteration, the returnee of the function needs to be written in the *"doctors_aid_outputs.txt"* file.

## *Prototypes of Functions and Subroutines*

### Accessing Input Lines

#### ***read_file()*** **Function**

*read_file()*

**Description:** This function opens *"doctors_aid_inputs.txt"* in read mode, reads it and stores all contents in it in a variable. This variable is then going to be used for iterating over each line. It also creates the *"doctors_aid_outputs.txt"* file.

**Parameters:** No parameters

**Returns:** a string variable that stores all contents in *"doctors_aid_inputs.txt"*.

**Algorithm**

1) Open the *"doctors_aid_inputs.txt"* file in read mode.
2) Store all contents of the file to a variable called *contents*.
3) Close *"doctors_aid_inputs.txt"*.
4) Create a file called *"doctors_aid_outputs.txt"* where the outputs are going to be stored.
5) Return *contents*.

## Iterating Over Input Lines

A global for loop is used for this task. The command and most importantly, the patient's name in each input line is fetched and stored into global variables. The global variable that stores patient's name is very important as its value is going to be used by most functions.

### Algorithm of The Tor Loop

```
1) Set the value of *lines* to the returnee of *read_file()*
2) For every line in *lines*:
3)         Find the command in the line and assign it to a variable called *command*
4)         Find the patient's name in the line and assign it to a variable called *patient_name*
5)         Set the value of *result* to the returnee of the function associated with *command*.
6)         Write the result to *"doctors_aid_outputs.txt"*.
```

Lines 3, 4, 5 and 6 of this algorithm are explained below.

## Finding the Command and the Patient Name in the Lines

In order to call the right function for the right command, we need to associate commands with functions. A dictionary named *function_dict* is used for this. This dictionary maps commands to functions. (line 5 in the for loop algorithm)

*function_dict* = {**keys:** input line commands, **values:** names of related functions}

Therefore, *function_dict[command]* returns the related function.

### Algorithm for Finding The Command And The Patient's Name

1) Split the line by whitespaces and add the elements to a list.
2) Set the value of *command* to the first element of this list.
3) Set the value of *patient_name* to the second element of this list if a second element exists. Else, set it to empty string. (for "list" command, a second element does not exist.)

## Functions Related with the Commands

The program contains some functions that are set as values in the *function_dict* mentioned above. There are also some other functions that are used by these functions that do tasks that are needed to be done several times.

***fetch_patient_info(info)* Function**

*fetch_patient_info(info)*

**Description:** This function returns the diagnosis accuracy, disease name, incidence, treatment name or treatment risk of the patient. It is important to note that this function, like many in this program, accesses the patient's name from the global scope and uses it.

**Parameters:**

- *info*(string): the strings "accuracy","disease","incidence","treatment_name" or "treatment_risk".

**Returns:** an information about that patient as a string, based on the value of "info".

*e.g. fetch_patient_info*("disease") returns the name of the disease the patient has been diagnosed with.

**Data Structures:**

- a dictionary named *info_dict* that maps every possible value of info to 1, 2, 3, 4, 5 respectively.
- "patients" (the database) as the list to loop through.

*info_dict* = {**keys:** every possible info, **values:** indexes}

The values are actually indexes where these information are stored inside the elements of the database. This dictionary is used for readability because we are going to call this function several times inside other functions.

*fetch_patient_info("accuracy")* is more readable than *fetch_patient_info(1)*.

### Algorithm

1) Declare *info_dict* that maps info values to indexes.
2) Set the value of *index_of_info* to *info_dict[info]*
3) Iterate over the elements of the database until you have found the information list whose first element is our patient.
4) Return *[index_of_info]'th* element of that list.

## *patient_is_recorded(patient)* Function

*patient_is_recorded(patient):*

This function returns True if the information about patient is present in the database. Otherwise, it returns False.

**Parameters:**

- *patient*(string): the name of the patient.

**Returns:** True or False

**Data structures:** Only the value of database is used.

### Algorithm

1) Iterate over each list in the database
2) If, at any point list[0] is equal to the patient, return True
3) if no match is found after iteration, return False

## *create_patient()* Function

*create_patient():*

This is a function that is associated with the "create" command and can be accessed by using *function_dict["create"]*. It attempts to append the data about the patient provided in the "create" line into the database.

**Parameters:** No parameters, accesses the name of the patient from the global scope.

**Returns:**

- *"Patient [patient name] cannot be recorded due to duplication."* if the database already contains data about patient.
- *"Patient [patient name] is recorded."* otherwise.

**Data Structures:** database, which is used as the target list.

### Algorithm

1) If the database already has patient's data, return the duplication message.
2) Else, gather all information that is separated by commas in the "create" line into a list and append that list to the database. Finally, return the success message.

***calculate_probability()* Function**

*calculate_probability()*

This function returns the probability of a patient having the disease by using the formula below.

$$probability = \frac{cases \times accuracy}{cases \times accuracy + (total\ population - cases) \times (100 - accuracy)}$$

**Parameters:** No parameters.

**Returns:** The probability value as a float

## Algorithm

1) Split the returnee of *fetch_patient_info("incidence")* by "/" and set the first element to *cases,* the second element to *total_population*.
2) Set the value of *accuracy* to the float version of *fetch_patient_info("accuracy")* (the returnee is originally a string)
3) Multiply *accuracy* by 100.
4) Set the value of *result* to *(cases * accuracy / cases * accuracy + (total population - cases)* (100-accuracy))*
5) Return *result*

***probability()* Function**

*probability()*

This is a function that is associated with the "probability" command and can be accessed by using *function_dict["probability"]*. This function returns a string denoting the probability of the patient having the disease.

**Parameters:** No parameters, accesses patient's name globally.

**Returns:**

- "Patient [patient name] has a probability of [probability] of having [disease name] if the patient's information exists in the database.
- "Probability for [patient name] cannot be calculated due to absence." otherwise

## Algorithm

```
1) If the patient's data is not present in the database, return the absence message.
2) Else:
3)     Calculate the probability by calling calculate_probability() and store the returnee
       into a variable called probability.
4) Return the appropriate message and plug in the value of probability in it.
```

## ***recommend()*** **Function**

*recommend()*

This is a function that is associated with the "recommendation" command and can be accessed by using *function_dict["recommendation"].* It returns a string denoting the recommendation for a patient. It decides on the recommendation by comparing the probability of having the disease and the treatment risk.

**Parameters:** No parameters, accesses patient's name globally

**Returns:**

- "System suggests [patient name] to have the treatment" if treatment risk is lower.
- "System suggests [patient name] NOT to have the treatment" if treatment risk is higher
- "Recommendation for [patient name] cannot be calculated due to absence." if patient's information is not present in the database.

### Algorithm:

```
1) If patient's data is not present in the database, return the absence message.
2) Else:
3)      If treatment risk > probability, return the "suggest against" message.
4)      Else if treatment risk < probability, return the "suggestion" message.
```

## ***remove_patient()*** **Function**

*remove_patient()*

This is a function that is associated with the "remove" command and can be accessed by using *function_dict["remove"].* It removes a patient's information from the database.

**Parameters:** No parameters, accesses patient's name globally

**Returns:**

- "Patient [patient name] is removed." if patient's information existed.
- "Patient [patient name] cannot be removed due to absence." otherwise

### Algorithm:

```
1) If patient's data is present in the database:
2)      Iterate over each list in the database until you find the patient's list.
3)      Remove that list from the database.
4)      Return the removal message.
5) Else:
6)      Return the absence message.
```

***list_patients()* Function**

*list_patients():*

This is a function that is associated with the "list" command and can be accessed by using function_dict["list"]. It tabulates each patient's information in the database.

**Parameters:** No parameters

**Returns:** A table as a string

**Data Structures:** A dictionary *column_lengths* for storing lengths of columns in the table

*column_lengths* = {**keys:** indexes 0, 1, 2, 3, 4, **values:** lengths 8, 12, 16, 12, 16}

Values indicate the maximum number of characters each column can store. Keys are column indexes. Columns are indexed from 0 to 4.

*column_lengths[0]* means the length of column 0 which is 8.

| **Patient Name (Column 0)** | **Diagnosis Accuracy (Column 1)** | **Disease Name (Col. 2)** | **Disease Incidence (Col. 3)** | **Treatment Name (Col. 4)** | **Treatment Risk (Col. 5)** |
|---|---|---|---|---|---|
| Hayriye | 99.90% | Breast Cancer | 50/100000 | Surgery | 40% |
| Deniz | 99.99% | Lung Cancer | 40/100000 | Radiotherapy | 50% |
| Toprak | 98.00% | Prostate Cancer | 21/100000 | Hormonotherapy | 20% |
| Hypatia | 99.75% | Stomach Cancer | 15/100000 | Immunotherapy | 4% |
| Pakiz | 99.97% | Colon Cancer | 14/100000 | Targeted Therapy | 20% |

## Algorithm:

```
1) Set the value of table to empty string
2) For each list in the database:
3)     For each data except the last in list:
4)         Determine which column this data will go into by looking at its index in the list and determine that column's length with the help of the dictionary column_lengths.
5)         Determine the number of tab characters needed to fit to data into the column by establishing a connection between the character count of data and the length of column
6)         Add the data and the tabs to table.
7)     Add the last data with a newline character to table.
8) Combine table with the column names and --------- separation.
9) Return table
```

## Writing The Returnee to the Output File

After calling the proper function for our line command and getting a returnee, we must write that returnee to the output file and finish executing one line of input.

***write_file(content)*** **Function**

*write_file(content)*

This function opens the *"doctors_aid_outputs.txt"* file in append mode and appends *content* to that file.

**Parameters:** *content*(string): the content that is going to be written

**Returns:** No return value. (None)

**Algorithm:**

1) Open the file
2) Write the content to it.
3) Close the file

# Programmer’s Catalogue

## *The Main Loop*

```
patients = []  # this is the multidimensional list that
is going to act as a database for patient data.

# this dictionary is used to associate commands in the
doctors_aid_inputs.txt file to functions in this
program.
function_dict = {"create": create_patient,
                 "probability": probability,
                 "list": list_patients,
                 "remove": remove_patient,
                 "recommendation": recommend, }

lines = read_file()          # stores the contents of
doctors_aid_inputs.txt file
```

Before entering the main for loop, we declare the database “patients” and declare a dictionary that maps commands found in lines to functions. We call the *read_file()* function and store all contents in the input file to the *lines* variable.

```
def read_file():
# this function reads and returns all contents in the
doctors_aid_inputs.txt file
    open("doctors_aid_outputs.txt", "w", encoding='utf-
8')    # this line is for removing any existing contents
in the output file.
    inputfile = open("doctors_aid_inputs.txt", "r",
encoding='utf-8')
    contents = inputfile.read()
    inputfile.close()
    return contents
```

This is the *read_file()* function that reads the *“doctors_aid_outputs.txt”* file and returns the contents of it. It also creates or overwrites *“doctors_aid_outputs.txt”* file. The function only sees the file named *“doctors_aid_inputs.txt”* and only outputs to a file named *“doctors_aid_outputs.txt”*. Flexibility for working with different named files was not implemented because of the scope of the assignment and for readability purposes.

```
for line in lines.split("\n"):           # goes through
every line in doctors_aid_inputs file
    line_data = line.split()
    command = line_data[0]
    patient_name = line_data[1] if command != "list"
else ""     # "patient_name" global variable stores
patient's name and gets used by multiple functions.
    output_line = str(function_dict[command]()) + "\n"
# output_line, calls the related function and stores
the returnee.
    write_file(output_line)      # lastly, the output is
written to the doctors_aid_outputs file.
```

This is the main loop of the program which iterates over each line of *lines* that stores every line in the input file, finds the command and the patient's name in the line and stores them to global variables, calls the right function using *function_dict[command]* and stores its returnee and lastly, writes that returnee to the output file.

```
def write_file(content):
# writes outputs to doctors_aid_outputs.txt file
    file = open("doctors_aid_outputs.txt", "a",
encoding='utf-8')
    file.write(content)
    file.close()
```

This is the *write_line()* function that writes the parameter *content* to the output file. It opens the file in append mode as not to overwrite other output lines that has already been recorded.

## *Function_Dict Functions and Other Functions*

```
def patient_is_recorded(patient):    # returns True if
a patient's data exists in the "patients" list
(database) .
    return True if patient in [i[0] for i in patients]
else False
```

This is the function that checks whether a patient's data exists in the database. It does this by gathering the 0th elements, which are patients' names, of every data list in the database. This function is used by many other functions.

```
def create_patient():                                    # a
function for storing patient data to the database.
    info = line[line.index(" ")+1:].split(", ")      #
starts from the position of the first whitespace (in
order not to get the command) and splits the line by
commas to get every detail about a patient.
    patient = info[0]                                    #
the first element is our patient's name. the global
variable patient_name has a comma at the end of it, so
we have to declare another variable for patient's name.
    if patient_is_recorded(patient):
        return "Patient {} cannot be recorded due to
duplication.".format(patient)
    else:
        patients.append(info)
        return "Patient {} is
recorded.".format(patient)
```

This is the function called by *function_dict["create"]*. It gathers every detail about a patient that is given in the "create" line into a list and appends that list to the database. It also calls the *patient_is_recorded()* function to check if the patient's data exists in the database.

```
def fetch_patient_info(info):                          # a
function for finding a certain detail about the current
patient
    info_dict = {"accuracy": 1, "disease": 2, "inci-
dence": 3, "treatment_name": 4, "treatment_risk": 5}  #
a dictionary that maps the "detail" about a patient to
its index in the patient's data list.
    index_of_info = info_dict[info]
    for data_list in patients:
        if data_list[0] == patient_name:
            return data_list[index_of_info]
```

This is the function that returns a certain detail/information about a patient. e.g. *fetch_patient_info("accuracy")* returns the current patient's diagnosis accuracy information. It basically searches through the data lists in the database and looks at the element at index 0, which locates patient's name. When it finds the name of the current patient, it returns the element at the index where the wanted information is stored.

This function does not check if the patient's data actually exists in the database because this function is called after and if the patient's data is confirmed to exist in the database. Calling this function before verifying that the patient exists in the database will lead to the return of None, and no such usage is present in this program.

A dictionary called *info_dict* is also used in this function to determine the index of the detail that is wanted. It is mainly for readability and efficiency.

```
def calculate_probability():
# a function that returns the probability of the
patient having the disease
    incidence, accuracy =
fetch_patient_info("incidence"),
float(fetch_patient_info("accuracy"))*100
    cases, total_population =
int(incidence.split("/")[0]),
int(incidence.split("/")[1])
    probability_value = (cases * accuracy) / (cases *
accuracy + (total_population-cases) * (100-accuracy))
    return probability_value
```

This function returns the probability of the current patient having the disease. The returnee of this function is used by the functions *probability()* and *recommend()* which are going to be shown after this. *fetch_patient_info()* is used to gather the values of "incidence" and "accuracy" which are both required for the computation of the probability value. Incidence is split by the slash ("/") for the extraction of the numerator and the denominator, which are the number of cases and the total population respectively.

```
def probability():
# a function for outputting the probability of a
patient having the disease.
    if patient_is_recorded(patient_name):
        return "Patient {} has a probability of {} of
having {}.".format(patient_name,
("{:.2f}".format(calculate_probability()*100)).rstrip("
0").rstrip(".") + "%",
str.lower(fetch_patient_info("disease")))  # rstrip is
used to remove trailing zeros.
    else:
        return "Probability for {} cannot be calculated
due to absence.".format(patient_name)
```

This is the function called by *function_dict["probability"]*. It knows whether a patient's data exists in the database by calling the *patient_is_recorded()* function. It calculates the probability by calling the *calculate_probability()* function. Instances of *rstrip()* are there to strip any trailing zeros because our probability values should not have any trailing zeros. Usage of *str.lower()* is to convert the disease name to lowercase letters for a more natural look on the output.

```python
def recommend():                         # this function decides
on whether the patient should take the treatment.
    if patient_is_recorded(patient_name):
        message = "NOT to" if
float(fetch_patient_info("treatment_risk")) >
calculate_probability() else "to"
        return "System suggests {} {} have the
treatment.".format(patient_name, message)
    else:
        return "Recommendation for {} cannot be
calculated due to absence.".format(patient_name)
```

This is the function called by *function_dict[“recommendation”].* It checks if the patient’s data exists by calling the *patient_is_recorded()* function. It compares the values of treatment risk and the returnee of *calculate_probability()*, which is the probability, and decides on an output string.

```python
def remove_patient():                         # this function
removes a certain patient’s data from the database.
    if patient_is_recorded(patient_name):
        for data_list in patients:
            if data_list[0] == patient_name:
                patients.remove(data_list)
                break
        return "Patient {} is
removed.".format(patient_name)
    else:
        return "Patient {} cannot be removed due to
absence.".format(patient_name)
```

This is the function called by function_dict[“remove”]. It checks if the patient’s data is present in the database by calling patient_is_recorded() and if so, iterates over each list in patients and removes the patient’s data upon finding it.

```
def list_patients():
    table =
"Patient\tDiagnosis\tDisease\t\t\tDisease\t\tTreatment\t\tTreatment\nName\tAccuracy
\tName\t\t\tIncidence\tName\t\t\tRisk\n--------------------------------------------
------------------------------\n"
    # the dictionary below maps the column indexes with their length. 0 being
Patient Name column which allows up to 8 characters, 4 being Treatment Name column
which allows up to 16 characters etc.
    column_lengths = {0: 8,
                      1: 12,
                      2: 16,
                      3: 12,
                      4: 16,
                      }

    for data_list in patients:            # this loop creates the table.
        for data_index in range(len(data_list)-1):
            column_length = column_lengths[data_index]
            data = "{:.2f}".format(float(data_list[1])*100) + "%" if data_index ==
1 else data_list[data_index]      # ternary statement prevents variable "data" from
being assigned to the accuracy values of 0.999, 0.975... instead of 99.90%,
99.75%...
            data_length = len(data)
            table += "{}\t".format(data) if data_length % 4 != 0 else data
# line 61 to 63 adds appropriate number of tabs.
            data_length = data_length if data_length % 4 == 0 else
((data_length//4)+1) * 4
            table += "\t" * int((column_length - data_length) / 4)
        table += "{}\n".format(str(float(data_list[5])*100).rstrip("0").rstrip(".")
+ "%")
    return table.rstrip("\n")
```

This is the function called by *function_dict["list"].* It first defines the *table* variable with the initial value below:

```
Patient Diagnosis Disease       Disease       Treatment     Treatment
Name    Accuracy  Name          Incidence     Name          Risk
--------------------------------------------------------------------
```

Then it creates a dictionary named *column_lengths* which maps indexes with lengths. The indexes are column indexes and the element at "n" index in every patient data list in the database will go to the column n.

e.g.  patient names are stored at index 0 of every data list, therefore they go to column 0 which is named "Patient Name".

In the for loop, the program iterates over each individual data in every data list in the database. In each iteration:

**Firstly**, it decides on the column length by using *column_lengths[data_index]*. E.g. if we are at the element at index 0 in a data list, *column_lengths[0]* will be the length of the column which is the length of column 0 which is 8.

**Secondly**, it takes the data it is currently iterating over and looks at its length. If the length is not a multiple of 4, it inserts a tab character at the end of it and saves the new length as the next multiple of 4. If the data's length is a multiple of 4 initially, no insertion of tab is made.

**Lastly**, the program divides the column length by the new length of the data (initial data + tab) and adds that many tabs at the end of the data. The newly created string gets added to *table.*

**At the end of each data list iteration**, the last element of the data list is added manually with a \n character at the end of it. This value is also added to table.

**After all this**, the table is returned with a single \n character stripped from the end of it.

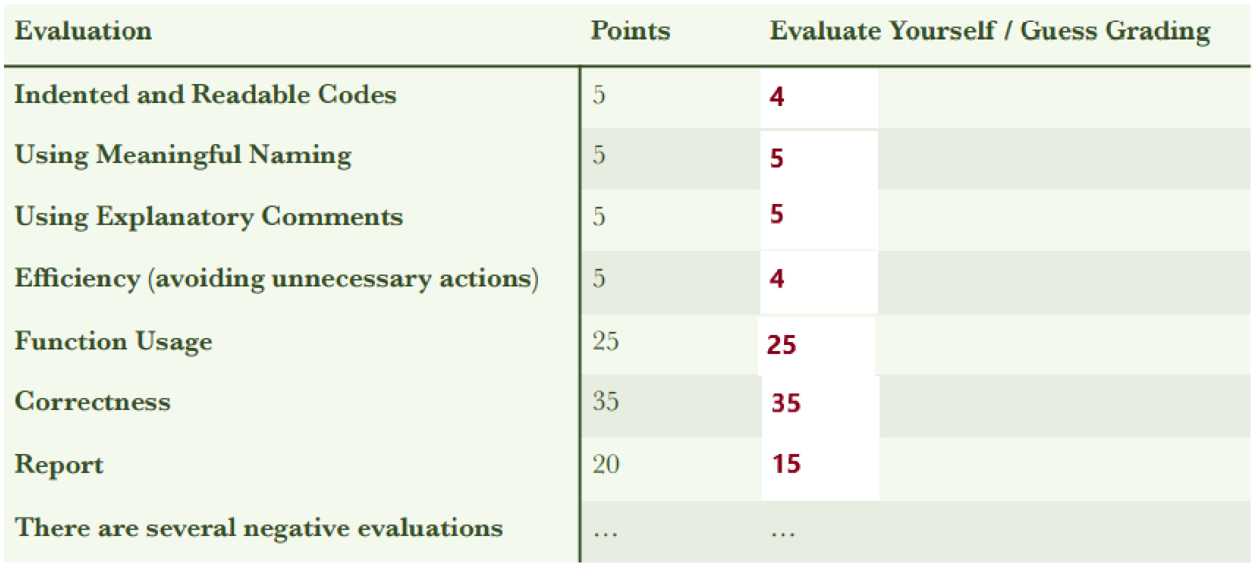
# User Catalogue

## *Program's Manual*

- Write your commands in a file called "*doctors_aid_inputs.txt*" and put that file in the same directory with the program.
- Go into the program and run it.
- The outputs are going to be written to a file called "*doctors_aid_outputs.txt*" in the same directory with the program.

## *Restrictions on The Program*

- The input lines must always be in the same format shown in the analysis section and no other commands should be used. Also, every line must include a command at the start of it.
- The treatment risk and the probability values cannot be equal. In case of this, the program will not be able to give a recommendation.
- Different patients with the same name cannot exist in the database at the same time.
- The command lines must be written to a .txt file named *"doctors_aid_inputs.txt"* and this file must be in the same directory with the program.



| Evaluation | Points | Evaluate Yourself / Guess Grading |
|---|---|---|
| Indented and Readable Codes | 5 | 4 |
| Using Meaningful Naming | 5 | 5 |
| Using Explanatory Comments | 5 | 5 |
| Efficiency (avoiding unnecessary actions) | 5 | 4 |
| Function Usage | 25 | 25 |
| Correctness | 35 | 35 |
| Report | 20 | 15 |
| There are several negative evaluations | … | … |