CSC108: Assignment 3: Medical Treatment Planning

Deadline: Thursday 4 April 2019 by 4:00pm

There is no re-submission for Assignment 3. Before submitting your assignment, run the checker and fix any errors (e.g, `SyntaxError`s) that prevent your code from running.

Introduction

In this assignment, you will write a program to recommend medical treatments based on the clinical attributes of a patient. You will be using the invasive Breast Carcinoma (BRCA) dataset from The Cancer Genome (TCGA) Atlas. In the provided files for this assignment, there is a filtered version of data from TCGA Data Portal that is based on data generated by the TCGA Research Network.

We will be predicting a treatment using nearest neighbour search. The algorithm will predict a treatment (or treatments) based on the treatments given to patients with the most "similar" patient profiles.

Goals of this assignment

- Continue to practice learning about a problem domain in order to write code about that domain.
- Continue to use the Function Design Recipe to plan, implement, and test functions.
- Design and write function bodies, with focus on using files, type dictionary, and dictionary methods.
- Practice good programming style.
- Design and write unittest test suites to test your functions.

How to not be interviewed for an academic offence

"The truth is that on every campus, a large proportion of the reported cases of academic dishonesty come from introductory computer science courses, and the reason is totally obvious: we use automated tools to detect plagiarism"

Professor Ed Lazowska, former chair of computer science and engineering at the University of Washington, Why Computer Science Students Cheat.

We are pretty good at detecting cheating. We hate finding it, though, so we have written this short guide on how to avoid being interviewed by us for an academic offence.

A typical penalty for a first offence is a **zero on the assignment**. The case will also be entered into the **UofT academic offence database**. (If you get caught a second time ever as an undergrad, the penalties are much, much more severe.)

Click to read our tips for how not to be interviewed for an academic offence.

Starter Code

Please download the Assignment 3 Files and extract the zip archive. In the sections below we explain the starter code and describe the functions that you will add to the starter files.

Data Files

Four different data files have been prepared for you to work with in this assignment.

- `medical_data.tsv` contains clinical information about more than one thousand patients.
- `medical_data_small.tsv` contains information about sixteen patients.
- `medical_data_three.tsv` contains information about three patients.
- `new_patients.tsv` contains information about ten patients without a treatment plan. A goal of the assignment is to write code that may be used to suggest treatment plan(s) for these patients.

The data is formatted as *tab separated values* (TSV). That means that the information in each line in the file is separated by a tab character (`'\t'`). The first line in the file contains the names of the patient attributes. Each subsequent line contains information for one patient. The first piece of information in the line is a patient id. The rest of the line contains the patient's values for each of the attributes. Some values may have the special value `NA`, which stands for "Not Available".

For example, the first two lines in `medical_data.tsv` are the following strings:

```
'Patient_ID\tAge\tGender\tTumor_Size\tNearby_Cancer_Lymphnodes\tCancer_Spread\tHistological_Type\tLymph_Nodes\tTreatment\n'
'tcga.5l.aat0\t42\tfemale\tt2\tn0\tm0\th_t_1\t0\tplan_1\n'
```

Expanding the tab characters into whitespace and lining up the columns gives this tabular format:

```
Patient_ID      Age   Gender   Tumor_Size   Nearby_Cancer_Lymphnodes   Cancer_Spread   Histological_Type   Lymph_Nodes   Treatment
tcga.51.aat0    42    female   t2           n0                         m0              h_t_1               0             plan_1
```

Required Functions: treatment_functions.py

The file `treatment_functions.py` contains the headers for the functions you need to write for this assignment. You should follow the Function Design Recipe to implement each function. You are encouraged to create helper functions in this file that are called by the required functions.

Required Testcases: test_missing_values.py

The file `test_missing_values.py` contains the start to a unittest testfile for the function `test_missing_values.py`. You will add appropriate unittests for this function to this file.

Constants: constants.py

The file `constants.py` contains some constants that you must use in your program. Make sure you do **not** change them!

Name	Purpose
`NA`	The special value that represents "Not available".
`TREATMENT`	The name of the attribute that contains the treatment information for a patient.
`PATIENT_ID_INDEX`	The position of the patient identifier in the input data files.

Constants used in Assignment 3, defined in `constants.py`

You should design helper functions!

Any time that you find that your code has become too complicated, try to identify a small task that could be solved by a separate function. Ask yourself the question ``Are there a few lines of code that are trying to solve a single, simple task?'' If the answer to that question is ``yes'', try isolating those lines in a helper function. Doing so will help you to write correct code more quickly. One place to start is to try to understand the structure of the data in the files. For example, writing separate function(s) to process different parts of the input file is one way to simplify the task of processing the whole file.

Part of the everyday experience of programming is figuring out what code to put into a helper function, what information the helper function needs to do its job, where the information comes from and how it is stored, and the resulting parameter types. You'll get better at this task with practice.

Data Types

In addition to the constants described above, the starter file `constants.py` contains constants for the data structures you will use to store data in your program.

Name	Purpose
`NAME_TO_VALUE`	Information about a patient is stored in a `NAME_TO_VALUE` dictionary. It is a dictionary that maps an attribute name to its value. For example, in the starter data files, we have attributes named `Age`, `Gender`, `Tumor_Size`, `Nearby_Cancer_Lymphnodes`, `Cancer_Spread`, `Histological_Type`, `Lymph_Nodes`, and `Treatment`. An example `NAME_TO_VALUE` dictionary is `{'Age': '42', 'Gender': 'female', 'Tumor_Size': 't2', 'Nearby_Cancer_Lymphnodes': 'n0', 'Cancer_Spread': 'm0', 'Histological_Type': 'h_t_1', 'Lymph_Nodes': '0', 'Treatment': 'plan_1'}` Both attribute names and attribute values are stored as `str`s.
`ID_TO_ATTRIBUTES`	Information about a group of patients is stored in an `ID_TO_ATTRIBUTES` dictionary. It is a dictionary that maps patient IDs to a corresponding dictionary that stores that patient's data. For example, the information from the starter data file `medical_data_three.tsv`, in which we have three patients, with IDs `'tcga.51.aat0'`, `'tcga.aq.a54o'`, and `'tcga.aq.a7u7'`, can be stored as the following `ID_TO_ATTRIBUTES` dictionary: `{'tcga.51.aat0': {'Age': '42', 'Gender': 'female', 'Tumor_Size': 't2', 'Nearby_Cancer_Lymphnodes': 'n0', 'Cancer_Spread': 'm0', 'Histological_Type': 'h_t_1', 'Lymph_Nodes': '0', 'Treatment': 'plan_1'},`

	'tcga.aq.a54o': {'Age': '51', 'Gender': 'male', 'Tumor_Size': 't2', 'Nearby_Cancer_Lymphnodes': 'n0', 'Cancer_Spread': 'm0', 'Histological_Type': 'h_t_2', 'Lymph_Nodes': '0', 'Treatment': 'plan_2'}, 'tcga.aq.a7u7' {'Age': '55', 'Gender': 'female', 'Tumor_Size': 't2', 'Nearby_Cancer_Lymphnodes': 'n2a', 'Cancer_Spread': 'm0', 'Histological_Type': 'h_t_1', 'Lymph_Nodes': '4', 'Treatment': 'plan_4'} } Patient IDs, attribute names, and attribute values are stored as strs.
VALUE_TO_IDS	The answer to a request like "Categorise all patients according to the values of the specified attribute" is stored in a VALUES_TO_IDS dictionary. For example, if the attribute of interest is named 'Gender' and our patient data is as above, then we get the following VALUE_TO_IDS dictionary: {'female': ['tcga.51.aat0', 'tcga.aq.a7u7'], 'male': ['tcga.aq.a54o']} Attribute values and patient IDs are stored as strs.
ID_TO_SIMILARITY	The measure of how "similar" the patients in our data are to another patient is stored in an ID_TO_SIMILARITY dictionary. The rules for computing similarity between two patients are given in the section Computing the Similarity below. For example, if a patient's data is {'Age': '50', 'Gender': 'female', 'Tumor_Size': 't2', 'Nearby_Cancer_Lymphnodes': 'n0', 'Cancer_Spread': 'm0', 'Histological_Type': 'h_t_1', 'Lymph_Nodes': '5'} then the ID_TO_SIMILARITY dictionary of this patient with the patients in our data above is: {'tcga.51.aat0': 5.28, 'tcga.aq.a54o': 3.67, 'tcga.aq.a7u7': 4.67} Patient IDs are stored as strs and similarities are stored as floats. **Make sure to do the calculation yourself to fully understand how the similarity is computed!**

Data structures used in Assignment 3, defined in constants.py

Computing the Similarity

Here is how we compute the similarity between two patients. The total similarity score is the sum of the similarity scores for each of the patient's attributes. Each attribute similarity score is determined as follows:

1. If either patient has an attribute value of NA (not available), the similarity score for the attribute is 0.5.
2. If the two attribute values are numeric, the similarity score for the attribute is
   1 / ((the absolute difference of the values) + 1).
3. Otherwise, the similarity score for the attribute is 0.0 if the two patient attribute values are different or 1.0 if the two patient attribute values are the same.

For example, if the two patients have the following data:

```
{'Age': '42', 'Gender': 'female', 'Tumor_Size': 't2', 'Nearby_Cancer_Lymphnodes': 'n0',
'Cancer_Spread': 'm0', 'Histological_Type': 'NA', 'Lymph_Nodes': '0'}

{'Age': '51', 'Gender': 'male', 'Tumor_Size': 't2','Nearby_Cancer_Lymphnodes': 'n0',
'Cancer_Spread': 'm0', 'Histological_Type': 'h_t_2', 'Lymph_Nodes': '0'},
```

then the similarity measure of these two patients is (rounded to 2 decimal places after the decimal point):

```
(1/(abs(51-42)+1)) + 0 + 1 + 1 + 1 + 0.5 + (1/(abs(0-0)+1)) = 4.60
```

Your Task

Required Functions

This section contains a table of detailed descriptions of the functions you must complete. These are the functions that we will be testing. However, you should follow the approach we've been using on large problems

recently and write additional helper functions to break down tasks and make your functions smaller and more concise. Each helper function must have a clear purpose. Each helper function must have a complete docstring produced by following the Function Design Recipe. You should test your helper functions to make sure they work.

Function name: (Parameter types) -> Return type	**Full Description (paraphrase to get a proper docstring description)**
`read_patients_dataset: (TextIO) -> ID_TO_ATTRIBUTES`	The parameter refers to a tab-separated values file that is open for reading. The data is in the Data Format described at the top of the Starter Code section. This function should read all of the data from the given file and return it in a `ID_TO_ATTRIBUTES`. See the Data Types table above for an explanation of the types. (Use your browser's back button to return to here.) NOTE: since this function takes a file open for reading as its argument, the function does **not** need to open the file, and must **not** close it. HINT: since the data in the file is in a prescribed format, use the structure of the file to guide the design of your code. Write helper function(s) to simplify your code.
`build_value_to_ids: (ID_TO_ATTRIBUTES, str) -> VALUE_TO_IDS`	The first parameter is an `ID_TO_ATTRIBUTES` that contains information about patients and their attributes. The second parameter is an attribute name for which information is desired. This function is to return a `VALUE_TO_IDS` dictionary. See the Data Types table above for an explanation of the types.
`patients_with_missing_values: (ID_TO_ATTRIBUTES, str) -> List[str]`	The first parameter is an `ID_TO_ATTRIBUTES` that contains information about patients and their attributes. The second parameter is the name of an attribute. This function should return a list of patient IDs of all patients who have the value `NA` (not available) for the given attribute name. See the Data Types table above for an explanation of the types.
`similarity_score: (NAME_TO_VALUE, NAME_TO_VALUE) -> float`	The parameters are the attribute names and values of two patients. The function should return the similarity score between these patients. The attribute `Treatment` should not be included in the calculation of the similarity measure. The return value should be rounded to 2 decimal places. See the Data Types table above for an explanation of the types. See the Computing the Similarities section above for the rules of computing the similarity measure.
`patient_similarities: (ID_TO_ATTRIBUTES, NAME_TO_VALUE) -> ID_TO_SIMILARITY`	The first parameter is an `ID_TO_ATTRIBUTES` that contains information about patients and their attributes. The second parameter is a `NAME_TO_VALUE` that contains the data for another, new, patient. The function should calculate the similarities between the given patient and every patient in the `ID_TO_ATTRIBUTES`. The return value is a `ID_TO_SIMILARITY` that maps each patient ID from the input `ID_TO_ATTRIBUTES` to the computed similarity measure between that patient and the patient with data from the given `NAME_TO_VALUE`. See the Data Types table above for an explanation of the types. See the Computing the Similarities section above for the rules of computing the similarity measure.
`patients_by_similarity: (ID_TO_ATTRIBUTES, NAME_TO_VALUE) -> List[str]`	The two parameters are exactly as in `patient_similarities`. The function should return a list of all patient IDs from the given `ID_TO_ATTRIBUTES`, sorted according to the these patients' similarities with the patient data in the given `NAME_TO_VALUE`. The output list should be sorted in descending order by similarity score. In the case of a tie, the patient IDs are sorted in alphabetical order. For example, if `ID_TO_ATTRIBUTES` and `NAME_TO_VALUE` are the dictionaries from the example in the Data Types table, then the returned list should be: `['tcga.5l.aat0', 'tcga.aq.a7u7', 'tcga.aq.a54o']`
`treatment_recommendations: (ID_TO_ATTRIBUTES, NAME_TO_VALUE)`	The two parameters are exactly as in `patient_similarities`. The function should return a list of **unique** values for the attribute named `TREATMENT`, in the following order. The

`-> List[str]:`	first value should be the treatment for the patient from `ID_TO_ATTRIBUTES` that has the greatest similarity with the patient in `NAME_TO_VALUE`. The second value should be the treatment for the patient with the second greatest similarity, and so on. If some patient has the value `NA` for the attribute name `TREATMENT`, this treatment is not included in the list of recommendations. Treatments should not be repeated in the returned list. For example, if `ID_TO_ATTRIBUTES` and `NAME_TO_VALUE` are the dictionaries from the example in the Data Types table, then the returned list should be: `['plan_1', 'plan_4', 'plan_2']`

`make_treatment_plans(ID_TO_ATTRIBUTES, ID_TO_ATTRIBUTES) -> None`	The first parameter is an `ID_TO_ATTRIBUTES` that contains information about patients and their attributes. The second parameter is an `ID_TO_ATTRIBUTES` that contains information for newly admitted patients, and in which the values for the attribute `TREATMENT` are `NA`. The function should *modify* the second dictionary by replacing the values for the `TREATMENT` attribute with the first recommended treatment, as computed by the function `treatment_recommendations`.
	
	For example, if `ID_TO_ATTRIBUTES` is the dictionary from the example in the Data Types table, and the second dictionary is:
	
	`{'newid': {'Age': '50', 'Gender': 'female', 'Tumor_Size': 't2', 'Nearby_Cancer_Lymphnodes': 'n0', 'Cancer_Spread': 'm0', 'Histological_Type': 'h_t_1', 'Lymph_Nodes': '5', 'Treatment': 'NA'}}`
	
	then the function should modify the second dictionary to:
	
	`{'newid': {'Age': '50', 'Gender': 'female', 'Tumor_Size': 't2', 'Nearby_Cancer_Lymphnodes': 'n0', 'Cancer_Spread': 'm0', 'Histological_Type': 'h_t_1', 'Lymph_Nodes': '5', 'Treatment': 'plan_1'}}`

List of functions to implement for Assignment 3

Required Testing

Implement unittests for the function `patients_with_missing_values`. These tests should be implemented in the file `test_missing_values.py`. The starter file is provided with the starter code.

Make sure your tests are exhaustive. We will grade this part of the assignment by running your tester on faulty implementations of the function `patients_with_missing_values` and seeing how many bugs your tester detects.

What Not to Do

- Do **not** call `print`, `input`, or `open`, except within the `if __name__ == '__main__'` block.
- Do **not** modify or add to the import statements provided in the starter code.
- Do **not** add any code outside of a function definition or the `if __name__ == '__main__'` block.
- Do **not** use any global variables (other than constants).
- Do **not** mutate objects unless specified.

Marking

These are the aspects of your work that will be marked for Assignment 3:

- **Correctness (70%):** Your functions should perform as specified. Correctness, as measured by our tests, will count for the largest single portion of your marks. Once your assignment is submitted, we will run additional tests, not provided in the checker. Passing the checker **does not** mean that your code will earn full marks for correctness.
- **Testing (10%):** We will run the `unittests` that you submit on a series of flawed (incorrect) implementations we have written. Your testing mark will depend on how many of the flawed implementations your unittests catch, whether they successfully pass a working (correct) implementation, and whether your test files contain redundant (unnecessary) tests.
- **Coding style (20%):**
  - Make sure that you follow Python style guidelines that we have introduced and the Python coding conventions that we have been using throughout the semester. Although we don't provide an exhaustive list of style rules, the checker tests for style are complete, so if your code passes the checker, then it will earn full marks for coding style with two exceptions: docstrings and use of helper

functions may be evaluated separately. For each occurrence of a PyTA error, a 1 mark (out of 20) deduction will be applied. For example, if a C0301 (line-too-long) error occurs 3 times, then 3 marks will be deducted.

- Your program should be broken down into functions, both to avoid repetitive code and to make the program easier to read. If a function body is more than about 20 lines long, introduce helper functions to do some of the work — even if they will only be called once.
- All functions, including helper functions, should have complete docstrings including preconditions when you think they are necessary.
- Also, your variable names and names of your helper functions should be meaningful. Your code should be as simple and clear as possible.

What to Hand In

The very last thing you do before submitting should be to run the checker program one last time.

Otherwise, you could make a small error in your final changes before submitting that causes your code to receive zero for correctness.

Submit `treatment_functions.py` and `test_missing_values.py` on MarkUs by following the instructions on the syllabus. Remember that spelling of filenames, including case, counts: your file must be named exactly as above.