CMPUT361 Winter 2024 Assignment 1 – Reading Test Collections

January 2024

Instructions

- All code must be written in Python3 and run in the Computing Science instructional lab machines using **only** the Python standard libraries, NLTK, or Spacy.
  - No other libraries are allowed. If your code uses them, an automatic grade of zero will be assessed.
- It is **your responsibility** to provide clear instructions for the TAs to execute your code **from the command line**.
- This programming assignment can be completed individually, in pairs, or in groups of three, under the **Consultation** model of collaboration as per the Computing Science Department Course Policies.
- Do not upload *binary* files or large text files of any kind to your GitHub repository.
- All code and/or data *must be on the GitHub repository created by following the instructions on eClass*, following the folder structure for the assignment and the configuration file for automated testing. **DO NOT MODIFY** the folder structure.
- Remember to *submit the URL of your repository through eClass* **before the deadline**.

Learning Objectives

This assignment is intended for you to become familiar with:

1. Reading a test collection (CISI) that will be used for testing IR algorithms.
2. Design and write test cases.

Overview

Your repository has four folders as below. Each folder has a specific purpose.

```
--- collections  --> 'raw' collection files
 |- code         --> code for processing collections
 |- processed    --> output of your programs in 'code'
 |- tests        --> programs to test your 'code'
```

Collections

Each collection has a name (e.g., `CISI`) and is composed of three files, all with the same name and a different extension.:

- The documents are stored in a file with extension `.ALL`
- The queries are stored in a file with extension `.QRY`
- The query answers are stored in a file with extension `.REL`

Your repository comes with one test collection, called `CISI`. You will create test collections which you place in this folder and use in your test cases.

Code for processing collections

There are six *stub* programs inside `code`:

```
read_corpus.py     --> reads a .ALL file into memory
read_queries.py    --> reads a .QRY file into memory
read_answers.py    --> reads a .REL file into memory
print_document.py  --> prints a single document to STDOUT
print_query.py     --> prints a single query to STDOUT
print_answer.py    --> prints the answer to a query to STDOUT
```

You must keep to the provided code structure.

Do not share code. Do not reuse code from other courses. Do not move any of the provided functions to other files.

Read programs

All `read_XYZ.py` programs take one **command line** argument: the **name** of the collection (i.e., not a full path). They must read the corresponding file inside the `collections` folder, build an in-memory data structure to hold the contents of the file, and write that data structure to the `processed` folder.

All `read_XYZ.py` programs will be executed from the root folder of the repository as in this example:

bash

```
% python3 ./code/read_corpus.py CISI
```

You can choose any Python data structure to implement your read programs. Dictionaries are probably the easiest.

All `read_XYZ.py` programs **must** write a file to the `processed` folder, with the contents of the data structure you created. The name of the file must be the same as the collection (e.g., `CISI`). You are free to choose the extension as long as you do not reuse the ones inside the `collections` folder (e.g., `documents`, `queries`, `answers`).

Do not hard-code full paths in your programs. You should hard-code only relative paths `./collections/` and `./processed/`.

Handling Exceptions:

All `read_XYZ.py` programs must throw an exception if the input file has an error (e.g., a missing required field or an extra field – see the repo for file specifications) **OR** if the corresponding output file already exists **OR** if there are no files inside `./collections/` corresponding to the collection name provided.

If an exception is thrown, no partial output should be written to any file in `./processed/`. If the program does not encounter any errors, it should print `SUCCESS` to `STDOUT`.

Print programs

All `print_XYZ.py` programs take two **command line** arguments. The first is always the collection name. The second will be an ID: `print_document.py` takes a document ID while the other two take a query ID:

- `print_document.py` should print to `STDOUT` (only) the original text in the collection corresponding to the document ID provided.
- `print_query.py` should print to `STDOUT` (only) the original text of the query corresponding to the ID provided.

- `print_answer.py` should print to `STDOUT` the IDs of the documents (one per line) that are relevant for the query corresponding to the query ID provided.

All `print_XYZ.py` programs **must not** read the files inside `./collections/`; instead, they must read the files inside `./processed/`.

Handling Exceptions:

All programs must throw an exception for invalid input: invalid collection name or invalid ID (document or query). An error message must be printed to `STDOUT` and the program must exit returning an error code to the OS.

Test cases

You must create at least two test collections and write test code inside the `./test/` folder to check for errors in the collection files.

Do not share test collections or test code.

Creating test collections

You must create at least two test collections: `good` and `bad` and write the corresponding files to `./collections/`:

- `good` should have 5 documents with 5-20 words each, and 5 queries (with their answers).
- `bad` should have 3 documents with 5-20 words each, and 5 queries (with their answers).

Everything in `good` (documents, queries, answers) should be as specified, meaning that all tests using that collection should succeed. `bad`, on the other

hand, should have at least one document, one query, and one entry in the answers file that is not according to the specifications and cause the corresponding test cases to fail.

If you prefer, you can have multiple `bad` collections, each with just one incorrect file and two correct ones. Name them `bad1`, `bad2` and `bad3`. They will be considered as a single collection for grading.

You **cannot** reuse documents or queries from `CISI`.

Creating test cases

You are provided three example test cases for the `print_XYZ.py` programs inside `./test/`. These are rather oversimplified (for example, they hard-code the expected answers and they test a single case), but should serve as a starting point.

You are **strongly encouraged** to write as many test cases as you can and no less than two test cases for each `read_XYZ.py` program: one that is meant to be successful and another that is meant to encounter an error. Your test cases should only use `good` and `bad`, and not `CISI`.

None of the test programs should take any command line arguments.

Documentation Requirements

Write all documentation in the files specified here (i.e., do not add or modify any other files for documentation purposes).

`README.md` at the root of the repo

Include the following:

- CCID(s) and name(s) of students who worked on the assignment.
- Instructions for using your programs from the command line on an instructional machine in the CS department, including all parameters.
  - TAs and instructors will copy and paste your instructions and test them.
- A list of all sources (e.g., StackOverflow) consulted to answer the assignment.

`README.md` **inside** `./tests/`

Include the following, for each test case:

- Name of python program.
- Purpose of the test (one sentence is fine).
- Sample output of the program.

Follow the formatting instructions in the example provided.

Grading and rubric

Weights:

- Code (read and print programs): 30% (5% each)
- Test collections: 20% (evenly distributed)
- Test programs: 50% (evenly distributed). You must write at least the following five test cases:
  - Three test cases that use the `bad` collection and FAIL – each must test a different feature and use a different file.
  - Two test cases that use the `good` collection and PASS – each must use a different feature and use a different file.
  - If you provide more than five test cases, we will grade all of them and use the best five scores.

20% bonus points for providing three test cases that generalize the ones provided by testing with multiple (e.g., 20) randomly-chosen entries (e.g., multiple documents). These should be named `bonus_1.py`, `bonus_2.py` and `bonus3.py`. Each will be graded according to the criteria below.

Below are the grading thresholds for **each** read/print program, test collection, and test program:

Grade	Each Read/print program	Each Test Collection	Each Test Program
100	1- follows structure provided 2- produces correct output 3- passes all TA-designed test cases 4- passes all student-provided test cases 5- runs on CS lab machines 6- execution instructions are correct 7- is well documented	1- contains the right number of entries (documents, queries, ...) 2- has the expected number of valid and invalid entries 3- does not reuse material from `CISI` 4- is not similar to another collection by other student(s)	1- does not take command-line arguments 2- is documented as specified on repository 3- indicates whether the test should be a PASS or a FAIL 4- is correct and readable
75	At least 5 of the points above	3 of the points above	3 of the points above
50	A least 4 of the points above	2 of the points above	2 of the points above
25	At least 2 of the points above	1 of the points above	1 of the points above

Grade	Each Read/print program	Each Test Collection	Each Test Program
0	Missing **OR** does not meet any point above **OR** code uses unauthorized library(ies).	Missing **OR** none of the points above	Missing **OR** none of the points above

NEXT
Assignment 2