CMPUT361 Winter 2024 Assignment 3 – Dot-product scoring

© Denilson Barbosa

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 and/or NLTK.
  - 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.
- 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.
- **DON'TS**: Do not do any of the following. Violations will incur an automatic grade of zero and/or further sanctions:
  - Forget to upload your GitHub repository on eClass.
  - Forget to commit your latest code to the `main` branch of the repository before the deadline.
  - Share code, test collections, or answers.
  - Ask for code, test collections, or answers from others.
  - Use any non-standard library except for NLTK.

- Upload *binary* files or large text files to your GitHub repository.
- Add, remove, or rename any folder in your repository.
- Modify the signature of any of the functions provided.
- Move the functions provided to other files.
- Add any python files except for one called `utils.py` where you can write helper functions that you need and use in multiple places.

Learning Objectives

This assignment is intended for you to become familiar with:

1. Reading a test collection (CISI) an representing it with an inverted index.
2. Scoring keyword queries using the dot-product of document and query vectors.
3. Designing and writing test cases.

Overview

Your repository has the folders below. Each folder has a specific purpose.

```
--- collections  --> 'raw' collection files
 |- code         --> code for bulding the index and answering queri
 |- processed    --> output of your programs in 'code'
 |- tests        --> programs to test your 'code'
```

Collections

Each collection has exactly 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_simplified`. Any collection you create must have three files and be placed in the same folder.

Code for answering queries

You will work with the following starter `code`:

```
build_index.py  --> writes the inverted index
query.py         --> answers queries using the index
preprocessing.py --> tokenization and normalization functions
```

Building the index

`build_index.py` takes one **command line** argument: the **name** of the collection (i.e., not an OS path). It must read the corresponding `.ALL` file inside the `collections` folder, build an in-memory data structure to hold the contents of the file, from which to build the index, which must be written to the `processed` folder.

Example usage:

```bash
% python3 ./code/build_index.py CISI_simplified
```

You are free to represent the inverted index in memory and in files inside the `processed` folder in any way you want, as long as your solution is not worse (in terms of asymptotic cost) to the solutions discussed in class. You must document your solution index in the `README.md` file inside `processed`.

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

Handling Exceptions:

Your program must throw an exception if the input file has an error **OR** if the corresponding output file already exists **OR** if there are no files inside `collections` corresponding to the collection name.

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

Answering queries

`query.py` must take four command line arguments (in this order): a collection name, a document scoring specification, the number of max answers (k), and a keyword query. The scoring specification applies to document vectors only. Weights in the query vectors must be computed using the `nnn` scheme. Your program must support the following schemes:

- Term frequency: `n`, `l`
- Document frequency: `n`, `t`
- Normalization: `n`, `c`

Example usage:

```bash
% python3 ./code/query.py CISI_simplified ltn 10 keyword
```

`query.py` **must not** read anything inside `./collections/`; instead, it must read the inverted index file inside `./processed/`.

The result of the query must be computed following the process and algorithms explained in class. Your code must use a min heap to keep only up to `k` answers in memory at any time.

Output: Your program must print a list of pairs `docID:score` in a single line, separated by a single `\t` character, corresponding to the `k` documents with highest score. The list must be sorted by decreasing score. Ties are broken by ID (smaller first). Scores must be rounded to three decimal places.

Handling Exceptions:

Your program must throw an exception for invalid input: invalid collection name or invalid query expression. An error message must be printed to `STDOUT` and the program must exit returning an error code to the OS. You do not need to check for errors while reading the expression, though. Errors can be detected when the expression is being rewritten or being executed.

Tokenization and normalization

You **must** use the tokenization and normalization steps as provided in `preprocessing.py`. Tokenization is done by removing punctuation and splitting the text into words by the NLTK library. Normalization consists of case folding and stemming using NLTK's Porter Stemmer.

Test collection

You must create a test collection called `good` and write the corresponding files to `./collections/` . It should have 5 documents with 5-20 words each, and 10 queries with their answers. Use only the `.I` and `.W` fields.

Everything in `good` (documents, queries, answers) should be as specified, meaning that all tests using that collection should be successful. To get full marks, you must have queries with a single relevant document, queries with two relevant documents, and queries in which all documents are relevant.

In short, you should be able to fully test your scoring algorithm using your `good` collection before using the provided answers in `CISI_simplified` .

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

Test cases

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

You must create the following tests:

1. `test1.py` : (1) calls `build_index.py` on your `good` collection; (2) reads the corresponding inverted index file(s) in `processed` ; (3) compares the inverted index read to another one, **hard-coded** inside `test1.py` , that you created by hand for your `good` collection. Your hard-coded index must be correct and the index read from the files must contain the same information.
2. `test2.py` : (1) calls `query.py` on your `good` collection once for each of the ten queries you created by hand; (2) checks that your program returns the correct answer for each query. You must test all possible scoring schemes for documents.
3. `test3.py` : (1) calls `query.py` five times, each with a hard-coded invalid input (randomly chosen OOV terms, missing collection, wrong scoring scheme,

…); (2) checks that your program returns an error in each case.

4. `test4.py` : (1) calls `query.py` on `CISI_simplified` 10 times, each on a different query; (2) checks the answer provided contains at least one document returned by your program. Choose the scoring scheme appropriately. `k` should be 50.

Documentation Requirements

Write all documentation in the files specified here (i.e., do not add or modify any other files for documentation purposes). Documentation written in the incorrect file will not be graded.

`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.

`README.md` inside `processed`

Describe your data structure(s) used and how the dictionary and the postings are represented. Explain if they reside in memory or in a file while `query.py` runs.

Grading and rubric

Weights:

- Code: 30%:
  - `build_index.py` : 10%
  - `query.py` : 20%
- Test collection: 30%
- Test programs: 40% (evenly distributed).

Grade	Each Read/print program	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	1- contains the right number of documents, queries, answers. 2- has the expected kinds of queries as specified 3- does not reuse material from `CISI` 4- is not similar to another collection by other	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

Grade	Each Read/print program	Test Collection	Each Test Program
	instructions are correct 7- is well documented	student(s) 5- has the correct answers to the queries provided.	
75	At least 5 of the points above	4 of the points above	3 of the points above
50	A least 4 of the points above	3 of the points above	2 of the points above
25	At least 2 of the points above	2 of the points above	1 of the points above
0	Missing **OR** none of the points above **OR** code uses unauthorized library.	Missing **OR** just one of the points above **OR** answers to queries are incorrect	Missing **OR** none of the points above

PREVIOUS
Assignment 2

NEXT
Assignment 4