CMPUT361 Winter 2024 Assignment 2 – Boolean Queries

© 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, 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) an representing it as a term-document incidence matrix.
2. Parsing and evaluating boolean queries.
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 matrix and answering bool
 |- 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_bool`, with the documents of the `CISI` collection used in other assignments but Boolean queries instead. You will create a test collection which you will place in this folder and use in your test cases.

Code for answering Boolean queries

You will work with the following starter `code` :

```
build_matrix.py  --> writes a term-document incidence matrix for th
query.py         --> answers queries using the matrix
precedence.py    --> functions to "wrap" sub-expression in paranthe
preprocessing.py --> tokenization and normalization functions
```

You must keep to the provided structure. Do not move any of the functions to other files.

Do not share code. Do not reuse code from other courses.

Building the matrix

`build_matrix.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 matrix, which must be written to the `processed` folder.

Example usage:

```bash
% python3 ./code/build_matrix.py CISI_bool
```

Complete the code provided to build the matrix and use the data structures in that code.

The matrix must look like the following example:

```Python
matrix={
    "_M_" : 4,
    "t1" : [1, 0, 0, 0],
    "t2" : [1, 1, 1, 1],
    "t3" : [0, 1, 1, 1],
    "t4" : [0, 1, 0, 1],
    "t5" : [1, 0, 0, 1]
}
```

`matrix['_M_']` contains the number of documents in the collection, which is also the length of all vectors. Other entries in the matrix are the term vectors. The matrix must be a proper JSON representation of the matrix, written to a file whose name is the same as the collection and whose extension is `.matrix.json` .

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 two command line arguments (in this order): a collection name and a query expression. The syntax for query expressions is the same discussed in class. Tokens in an expression **must be separated by blanks**, otherwise the expression is considered invalid.

For example, `:not: ( t1 :and: :not: ( t2 ) )` is valid, while `:not:(t1 :and: :not:(t2))` isn't.

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

The result of the query must be computed following the process and algorithms explained in class. First, you must rewrite the expression by adding parenthesis "around" expressions with `:not:` and with `:and:` . Next, you must read the resulting expressions left-to-right, using a stack, and solving expressions at the top of the stack when encountering a `)` .

The functions for rewriting expressions are inside `precedence.py` . The function `fix_precedence()` is already called in the right place – in `tokenize_and_answer()` inside `query.py` . All you need to do is to implement the helper functions inside `precedence.py` .

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.

We will look at your matrix to check that you have tokenized and normalized the text correctly, in which case your code should produce the exact same matrix as the instructors' code.

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.

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 term, queries with conjunctions, disjunctions, queries using negation, and queries using parenthesis. At least two queries must use all operators and parenthesis. At least two queries must be the complement of one another.

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

You **cannot** reuse documents or queries from `CISI`. Do not share test collections, queries, or test code.

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_matrix.py` on your `good` collection; (2) reads the corresponding `good.matrix.json` file using Python's built in JSON library; (3) compares the matrix read to another matrix **hard-coded** inside `test1.py` with a matrix that you create by hand out of your own `good` collection. The matrices must be identical.
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.
3. `test3.py` : (1) calls `query.py` on your `good` collection five times, each with a hard-coded invalid query (term not in vocabulary, misplaced parenthesis, missing blanks between operators, invalid `:and:` expression, and invalid `:or:` expression); (2) checks that your program returns an error in each case.
4. `test4.py` : (1) calls `query.py` on `CISI_bool` collection once for each query in `CISI_bool.QRY` ; (2) checks the answer provided in `CISI_bool.REL` is identical to what your program finds.

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: 30%:
  - `build_matrix.py` : 5%
  - `precedence.py` : 15%
  - `query.py` : 10%
- 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	1- contains the right number of documents, queries, answers. 2- has the expected kinds of queries as specified 3- does not reuse	1- does not take command-line arguments 2- is documented as specified on repository 3- indicates whether the

Grade	Each Read/print program	Test Collection	Each Test Program
	test cases 5- runs on CS lab machines 6- execution instructions are correct 7- is well documented	material from `CISI` 4- is not similar to another collection by other student(s) 5- has the correct answers to the queries provided.	test should be a PASS or a FAIL 4- is correct and readable
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 1

NEXT >
Assignment 3