CISC 4610 Fall 2020 Homework 4: Text retrieval

Due 12/10/2020

Introduction

For this assignment, you will be processing the texts of all inaugural addresses of all US presidents. You will tokenize them, normalize the tokens, and build a vector space representation of the documents so they can be queried.

Starter code is provided in Python, but ***you can submit in Java or C++ as well***. You will complete several functions within the code or, for Java/C++, write the entire code yourself. You will submit your code and the output of your queries.

Introduction to the data

The data is stored in the "data/" directory, one file per speech, in plain text. The format of the transcription of the speeches is slightly inconsistent but you can ignore that.

Introduction to the code

The Python code is contained in the file "runIndexing.py". If you have all dependencies installed, you should be able to run the script as it is. If it works, it should print out (among others):

"vocabulary size: 0"

Tasks

The main() function of the starter code loads all documents and sends them to function placeholders in the correct order. Your task is to implement these functions according to the specifications laid out in comments in the code and described below.

The specifications refer to the Python datatypes "list" and "dictionary". Lists are variable size arrays and dictionaries are hash maps. If you choose to submit Java or C++, use equivalent datatypes in those languages.

All functions should be implemented with basic functionality, not using specialized text processing or linear algebra modules/packages (like nltk or numpy for Python). In Python, you should not import any more packages beyond what the starter code does. In Java/C++ you should not need anything beyond basic file processing, data types, and math (for logarithm). Please ask before using anything else.

Note that the starter code handles document vectors as row vectors rather than column vectors. That is, matrix[i] refers to a document, matrix[i][j] refers to the weight of term j in document i. This simplifies the code.

Since the code is broken up into many small functions, you can test these individually as you implement them. I recommend that you do that to verify that your code is working correctly. Your implementations do not have to check that the input has the correct format and may fail if that is not the case. Of course, your implementation should ensure that the functions return values in the correct format so this will not happen.

Implement functions in the code

1. Implement “tokenize()”. It expects a string containing the text of a document and returns a list of (non-empty) strings representing the tokens. All tokens are maximal sequences of alphabetical characters (a-z, A-Z), all other characters are treated as delimiters (where a token starts or ends). The beginning and end of the string also serve as delimiters.
2. In a comment within “tokenize()”, give 5 examples (made up by yourself) of different types of character sequences that will be handled inappropriately by this simple tokenization algorithm.
3. Implement “normalize()”. It expects a list of strings (tokens) and returns the same list all strings changed to lower case in place.
4. In a comment within “normalize()”, give 5 examples (made up by yourself) of token pairs that might be normalized and treated as the same (5 different types of differences between the tokens in the pairs) but are treated as distinct by this simple normalization algorithm.
5. Implement “getVocabulary()”. It expects a list of lists of strings and returns a sorted list of all distinct strings across all those input lists. (vocabulary should contain 9084 items)
6. Implement “getInverseVocabulary()”. It expects a list of (distinct) strings and returns a dictionary mapping from strings to indexes in the input list. That is, for any term in the input vocabulary, the output term2id should behave as follows: vocab[term2id[term]] = term. It is ok if it produces an error for terms that do not exist.
7. Implement “getTermFrequencies()”. It expects a list of terms (normalized tokens of a document) and the term2id mapping produced by “getInverseVocabulary()”. It produces a list of integers/floats representing the term frequencies for all terms, in the same order as in the vocabulary. That is, tfs[i] refers to the term for which term2id[term] = i, for all i. It should be able to handle terms in the input list that are not in the vocabulary (i.e., cannot be mapped by term2id), simply by not counting them at all.
8. Implement “getInverseDocumentFrequencies()”. It expects a 2D weight matrix (with documents in rows), determines all document frequencies (how many documents each term appears in) and then computes and returns the inverse document frequencies from them (see lecture 19 for the formula).
9. Implement “logTermFrequencies()”. It expects a list of term frequencies (integers/floats) and returns the same list after changing all of them to log term frequencies in place (see lecture 19).
10. Implement “getTfIdf()”. It expects two vectors (lists) of term frequencies and inverse document frequencies (floats) and returns their elementwise product in a new list.
11. Implement “normalizeVector()”. It expects a vector (list) of log term frequencies (floats) and returns the same list after dividing all of them (in place) by the L2 norm of the vector (the square root of the sum of the squares of all its elements).
12. Implement “dotProduct()”. Expects two vectors (lists of floats) of the same length and returns their dot product, i.e., the sum of the pairwise products of their corresponding elements.
13. Implement “runQuery()”. It expects a query string, an integer k, a 2D weight matrix, and the term2id mapping produced by “getInverseVocabulary()” and returns up to k documents with non-zero score, sorted in descending order of their score. It tokenizes and normalizes the query, determines its term frequencies and the log of those frequencies, normalizes that vector, and then computes the dot product between the normalized query vector and all document vectors (row of the given matrix). Then it returns the indices for the k documents with the highest scores (only non-zero).

Submission

Submit the following two files in a zip file named "<lastname>_4.zip" by 12/10/2020, 2:15pm:

1. Your completed version of "runIndexing.py" (or an equivalent file with Java/C++ code). Note that it should contain two comments in response to points 2 and 4 above.
2. A text file containing the output of your code.

In a change from the general policy, late submission will work as follows: 10 percentage points will be subtracted for the first partial or full day, then 30 percentage points for each partial or full day after that.

On Collaboration and Academic Integrity

All code must be produced by yourself (and should use only basic functionality of the programming language). You may, of course, discuss the assignment with other students but keep those discussions limited to concepts and ideas, do not write the actual code together. You may also look for help online but if you use code snippets etc. found online, mark the source. Violations of these policies are subject to penalty.