Programming Methods NA 2019
Second programming assignment: Abstracts

Deadline: November 7, 2019 18:00

The second programming assignment for the Programming Methods NA course in the fall of 2019 is called Abstracts. The most recent version of this assignment can always be found on BlackBoard. This text is the version of October 7, 2019.

Description

A large problem in the scientific field of Physics is to find research studies that are relevant to your interests. Often, you need to work hard and read a lot of books and papers in order to find relevant topics in your field. Especially when you are focusing on a specific project, you might find that a simple Google search provides a huge result of potentially relevant papers. To search through a large collection, you first need to which relevant words are used.

In this assignment, you will implement a program to get a better insight into the papers to see if they are relevant to your interests. Let's say that you have a large list of papers and you want to quickly see which ones have relevant words in the abstract, the first portion of the paper. You download a large file containing an abstract on each line. To help store such a large file on your machine, you want to **compress it by replacing the words with short index numbers**. The other tasks that you need to do with this file is to **find the term frequency of each word in the abstracts, compared to how often the word appears in the entire file**.

First of all, the program should print the information block. The user is then asked whether he or she wants to encode or decode the file. In each case, the program asks what the original (existing) file and the "target file" are called. The altered input file ends up in this target file; the input file itself remains unchanged (!). The program also asks for an index file, where information about the compression and word frequency are stored during encoding and used to reconstruct the file during decoding. During the decoding, the user can also provide a line number. The corresponding line in the file will be used to calculate the TF/IDF scores of all the words on the line. Line numbers start counting from 1, and if the user inputs 0 or a number higher than the number of lines in the input file, then no calculation is performed.

The **encoding** consists of compression and frequency scoring. Both operations take place while the program reads the file, and it is only read *once*. Read the file *line by line* (how do you detect a new line?). Every word that you find in the file must be compressed using the index table (a list or dictionary), and the word must be added to the frequency tables. A word consists of letters (`A` through `Z` and `a` through `z`), and starts and ends at any character that is not a letter, such as spaces or punctuation. Every time you find a word that you did not see before, add the word to the index table. Replace the entire word with the index number corresponding to the word in the output. So the following line:

```
This is an amazing "abstract" AND this: is the end of this amazing abstract.
```

is encoded as:

```
1 2 3 4 "5" 6 7: 2 8 9 10 7 4 5.
```

To be able to decompress later, we should have assumed that there are no numbers in the line, after all, what would the encoding `1 2 3` mean above — `This is an` or `This is 3`? To solve this problem, digits in the abstracts are preceded by a `\` (backslash) during compression. A backslash itself is encoded with two backslashes. For example, `Hello123hi\bye\\\1` is encoded as `1\1\2\32\\3\\\\\\\1`.

During the compression, you must *also* track the number of times a word is seen in all abstracts. Do this for the *lowercase* variant of each word. Use separate dictionary objects to keep track of these words and only write the index file as a CSV file at the end. If a new word with a capital letter in it is found, for example `This`, then create a new index number for the word `This` normally, but only track the number of times you have seen the word with *small letters*, i.e., `this`. Count both the number of times you see the word in total as well as the number of times you see the word in different abstracts (on different lines). Once you have read all the lines, combine the index table with the *two* frequency tables to form the CSV file, which has the columns `word,index,frequency,abstracts`.

The index CSV file for the example before would look like this:

```
word,index,frequency,abstracts
This,1,0,0
is,2,2,1
an,3,1,1
amazing,4,2,1
abstract,5,2,1
AND,6,0,0
this,7,3,1
the,8,1,1
end,9,1,1
of,10,1,1
and,0,1,1
```

Note that the `abstracts` column can only be zero (for words with uppercase characters) or one (since we only have one abstract, the word can only be in one of them). Larger files with more lines (abstracts) in them might have higher values in this column.

After encoding, the compression rate, which is the ratio of total number of characters that the program has written to an output (the target file *plus* the *words* in the index file), divided by the number of characters in the original file, must be printed on the screen in percentages (neatly rounded to a whole number), as well as the total number of lines processed. The compression rate for the example would be counted as follows: 77 characters in the original file (including newline character) versus 33 characters in the compressed file plus 41 characters in all the words in the index file, which gives a compression rate of about 96%.

Finally for encoding, track for every letter how often a word in the input file starts with that letter, and how many unique words there are for each letter (count uppercase and lowercase letters as the same). Use for example a list to track these counts and save the count for `a` under index `0`, `b` under `1`, etc. (Tip: Use the function `ord` and make use of the frequency table). Display a *textual* histogram with horizontal bars that are at most 40 characters long. Determine which frequency is the highest and give this letter a bar of 40 characters. For all other letters, calculate the width proportionally to the longest one. For each letter, two bars are drawn over each other: the bar with the number of distinct words receives + characters

and the bar with the frequency of words beginning with the letter has = characters. If a letter has no words that start with it, then it is not included in the histogram and no line is printed. To the right of the bar, show the number of unique words and the frequency of all words that begin with the letter, neatly aligned. An example histogram looks like this:

```
a | ++++++++++++++++++++++++++============= |     4 |     6
e | +++++++                                  |     1 |     1
i | +++++++======                           |     1 |     2
o | +++++++                                  |     1 |     1
t | ++++++++++++==============               |     2 |     4
```

During the **decoding**, use the index file to determine which number in the compressed file should be replaced with which word. Correctly handle numbers with backslashes and backslashes with backslashes in front of them. Make sure that you get the same contents in the target file as you did before compressing the first file! Also, when you reach the line that the user selected for frequency scoring, use the same principle as during compression to create a frequency table, but *only* for this line. Once you have read *all* the lines, compare the newly created frequency table against the `abstracts` column in the index file as follows:

TF/IDF (*term frequency—inverse document frequency*) is a metric which is used to reflect how important a word is to a single abstract in a large collection, since a term with a higher frequency in a single abstract can be trivial, such as "the", "an", "a", etc. The formula to calculate the TF/IDF score for one word is:

$$W_i = tf_i \times \log(\frac{N}{df_i}) \tag{1}$$

In equation 1, tf_i represents the number of occurrences of word i in the current abstract, whereas df_i represents the number of abstracts containing word i while N is the total number of abstracts. $\log$ is the natural logarithm.

Display the TF/IDF scores as a top-10 list, showing the word with the highest score first. Show both the word and the score. Note: You can only calculate and print these scores once the whole file is processed, since only then you know the value of N in formula 1!

Report

The following sections are expected in the report:

- Your own title, with the author names and student numbers (of course)
- A chapter that *briefly* explains what the program does based on a (self-made) small sample file (one or two lines, with `\begin{verbatim}` and `\end{verbatim}` around it). Explain how the file is compressed and decompressed, and show the terminal output.
- A chapter that briefly indicates when the program does not work, if applicable.
- A time record in hours per week; for pairs, show the hours for both. Use a table; see `https://en.wikibooks.org/wiki/LaTeX/Tables`. Put the total hours at the bottom.
- And finally the code of the program, neatly printed with `lstlistings`.

Use the template from the first assignment (lab 3) to create the report. Do not forget to set the language of your report and code.

Approach and remarks

Start by implementing compression and then decompression. See the description of lab 5 for a step-by-step tutorial. Test files are available on BlackBoard. Rough indication for the length of the Python program: approximately 250 to 300 lines (including blank lines and comments).

- We assume that the user is kind enough not to make mistakes when entering the file names and line numbers. When the user provides a choice to compress or decompress, check if the input (a word, number or character) is correct according to what you ask to the user. If the input is not correct, ask the question again until a correct input is entered.
- You may use the following modules in the program: `sys`, if you need `sys.exit`; `math` for the TF/IDF formula and display; `csv` for the index file (optional); `pathlib` or `os.path` for testing if files exist, if you want to. Do not use any other Python libraries!
- An important part of this assignment is the writing of functions. Your program must therefore consist of several functions, such as compression, decompression, asking the user questions, printing a histogram, TF/IDF calculation, top-10, and so on.

  It is difficult to keep the compression and decompression functions within 30 lines. These two functions may therefore be 60 – 70 lines long. Do **not** use global variables!
- The input file may be read symbol by symbol or line by line. See also Chapter 9 from the lecture notes. No matter how you read the file, only read each character once. Do not scan a line more than once, and do not write characters back to the input file.
- Remember the *information block* for the user that appears on the screen when the program starts. It should introduce the program, the people who created it and what the user is expected to do with the program. At the top of the code of the program we would like to see *comments* with the names and student numbers of the makers, which assignment it concerns, briefly what the program does and the date of the last edit.

Submission (Deadline: November 7, 2019 18:00)

Method of submission (per pair of two, only *one* has to submit):

- Submit the Python code (a `.py` file) and the report (in PDF format) through BlackBoard. You can find the submission page below the "Contents / Assignments" menu at the "Second assignment" link. The last submission before the deadline will be graded.
- Name the files like so: `XXXXXXX-YYYYYYY-opdr2.py` and `XXXXXXX-YYYYYYY-opdr2.pdf`, where you replace `XXXXXXX` and `YYYYYYY` with the two student numbers of the creators.
- Specify the submission date and the names of the two creators everywhere, especially in the comments on the first lines of the Python code as well as the report. You **must** use the template from the third lab session to make a simple report (see above).
- *You can only submit digitally, copies on paper are not accepted.*
- Use Python 3.5 or higher. The program should run on Linux, Mac and Windows.

Grading key: (consistent) layout 1; report 1; comments 2; modularity 2; functionality 4.