CITS1401 Computational Thinking with Python
Project 2 Semester 2 2019

Project 2: Using Stylometry to Verify Authorship

Submission deadlines:

Stage 1: **5:00pm, Friday 18 October 2019** for the pseudocode

Stage 2: **5:00pm, Friday 25 October 2019** for the code.

Value: **20%** of CITS1401.

To be done individually.

You should construct a Python 3 program containing your solution to the following problem and submit your program electronically on LMS. No other method of submission is allowed.

You are expected to have read and understood the University's guidelines on academic conduct. In accordance with this policy, you may discuss with other students the general principles required to understand this project, but the work you submit must be the result of your own effort. Plagiarism detection, and other systems for detecting potential malpractice, will therefore be used. Besides, if what you submit is not your own work then you will have learnt little and will therefore, likely, fail the final exam.

You must submit your project before the submission deadline listed above. Following UWA policy, a late penalty of 10% will be deducted for each day (or part day), after the deadline, that the assignment is submitted. However, in order to facilitate marking of the assignments in a timely manner, no submissions will be allowed after 7 days following the deadline.

Overview

UWA, like every university around the country (probably around the galaxy) is very worried about ghost-written submissions for assignments. This is also known as contract cheating. Whatever you call it, ghost-writing is about getting someone else to do your work, but submitting it as if it was only your work. In this case we are concerned with essays. The incidence is believed to be low, but it's clearly not a good thing.

Coming from a different angle, debates have raged at various times about whether different authors' works were actually by those authors. For example, were all the works attributed to William Shakespeare actually by him? One approach to examining both of these issues is to use stylometry. That is, rather than looking directly at the content of texts, as one does when looking for suspected plagiarism, stylometic looks for stylistic similarities. In other words, similarities in the ways a particular author uses language, rather than similarities in the actual words on the page, on the assumption that an author will use a similar style for similar sorts of content, fiction, non-fiction, etc.

What you will do for this Project is write a program that reads in either one or two text files containing the works to be analysed and builds a profile for each. Then either the profile is listed, or if there are two text files, the two profiles are compared, returning a score which reflects the distance between the two works in terms of their style; low scores, down to 0, imply that the same author is likely responsible for both works, while large scores imply different authors.

Specification: What your program will need to do

Input:

Your program must define the function main with the following signature:

```
def main(textfile1, textfile2, feature)
```

The first and second arguments are the names of the text files with a work to be analysed. The third argument is the type of feature that will be used to compare the document profiles. The allowed feature names are: "punctuation", "unigrams", "conjunctions" and "composite".

Output:

The function is required to return the following outputs in the order provided below:

- the score from a pairwise comparison rounded to four decimal places,
- the dictionary containing the profile of first file (`textfile1`), and
- the dictionary containing the profile of second file (`textfile1`)

A more detailed specification

- For the purposes of this project, a sentence is a sequence of words followed by either a full-stop, question mark or exclamation mark, which in turn must be followed either by a quotation mark (so the sentence is the end of a quote or spoken utterance), or white space (space, tab or new-line character). Thus:

  This is some text. This is yet more text

  contains one sentence followed by the start of another sentence.

- You are required to create the profile of input files using dictionaries. The profile for each document will contain the number of occurrences of certain words (case insensitive) and pieces of punctuation.

- The counted words or punctuations are dependent on the input feature which can be: "punctuation", "unigrams", "conjunctions" and "composite".

- For **conjunctions**: your program is required to count the number of occurrences of the following words:

  "also", "although", "and", "as", "because", "before", "but", "for", "if", "nor", "of", "or", "since", "that", "though", "until", "when", "whenever", "whereas", "which", "while", "yet"

- For **unigrams**: your program is required to count the number of occurrences of each word in the files. Consider the following three lines of text contained in a file:

  This is a Document.
  This is only a document
  A test should not cause problem

  The word count will be: "a":3, "document":2, "this":2, "is":2, "only":1, "should":1, "not""1, "cause":1, "problem":1

- For **punctuation**: your program should count certain pieces of punctuation: comma and semicolon. In addition, your program should also count single-quote and hyphen, but only under certain circumstances. Specifically, your program should count single-quote marks, but only when they appear as apostrophes surrounded by letters, i.e. indicating a contraction such as "shouldn't" or "won't". (Apostrophe is being included as an indication of more informal writing, perhaps direct speech.). Your program should count dash (minus) signs, but only when they are surrounded by letters, indicating a compound-word, such as "compound-word". Any other punctuation or letters, e.g '.' when not at the end of a sentence, should be regarded as white space, so serve to end words. For these purposes, strings of digits are also words as they convey information. Therefore, in the unlikely event that a floating point number, such as 3.142, appears, that is regarded as two words.

  Note: Some of the texts we will use include double hyphen, i.e. "--". This is to be regarded as a space character.

- For **composite**: your program should contain number of occurrences of punctuations (as explained above) and conjunctions. In addition, your program should also add to the profile two further parameters relating to the text: the average number of words per sentence and the average number of sentences per paragraph, where a paragraph is any number of sentences followed by a blank line or by the end of the text.

- Each of the words and punctuation symbols should be placed, together with their respective counts, in a dictionary, which is called a *profile*.

- The first output by the main function is the distance between the corresponding profiles which should be computed using the standard distance formula:

$$dist = \sqrt{\sum_i (profile1_i - profile2_i)^2}$$

- The second and third outputs returned by the main function are the *profiles* corresponding to the first and second text files respectively. The returned *profiles* as dictionaries in which each word is the key and value is the number of occurrences of the key, such as `{“also”:10, ”got”: 6}` where `“also”` and `“got”` are the keys and have occurred 10 and 6 times respectively.

Example:

Download the project2data.zip file from the folder of Project 2 on LMS. An example interaction is provided as a sampleanswers.txt which you can find in sampleresult.txt. The results are based on three files: sample1.txt and sample2.txt, both excerpts taken from "Life on the Mississippi", by Mark Twain.

Some Text Files to Examine

Some text files are also included in the zip file for you to try out. All of the texts, apart from "Kangaroo", were obtained from Project Gutenberg (www.gutenberg.org). All the files have a long text at the end which contains Project Gutenberg license and terms of use. I have provided the Gutenberg terms and license in the file licenseNterms.txt rather than left them in the texts because that may affect the profiles.

Author	**Title**	**Fiction/Non-fiction**
Henry Lawson	Children of the Bush	Fiction
D. H. Lawrence	Fantasia of the Unconscious	Non Fiction
Mark Twain	Life on the Mississippi	Non Fiction
D. H. Lawrence	Sea and Sardinia	Non Fiction
D. H. Lawrence	Kangaroo	Fiction
Mark Twain	Adventures of Hucklebery Finn	Fiction
Andrew Barton 'Banjo' Paterson	Three Elephant Power	Fiction

A small note of warning. If you decide to download your own texts from Project Gutenberg, please be aware that many of the texts include spurious Unicode characters. Unfortunately, the file input-output functions we use in CITS1401 (and I use on a daily basis) only work with the standard ASCII character set, so will cause an exception if Unicode characters are in the text. While Python is well able to deal with Unicode, special input-output functions are needed, which are beyond the scope of this unit. What I have done is use the Unix command: cat -vet *filename* to make the Unicode characters visible in the ASCII character set, and then use a text editor to remove them. (Tedious.)

Important:

You will have noticed that you have not been asked to write specific functions. That has been left to you. However, as in Project 1, **it is important that your program defines the top-level function `main()`** as described above. `main()` should then call the other functions. (Of course, these may call further functions.)

The reason this is important is that when I test your program, my testing program will call your `main()` function. So, if you fail to define `main()`, or define it with a different signature, my program will not be able to test your program.

Things to avoid:

There are a few things for your program to avoid.

- You are **not allowed** to import **any** Python module except `math` or `os`. While use of other modules are perfectly sensible thing to do (and the way I often may do it), it takes away much of the point of different aspects of the project, which is about getting practice creating code to accurately extract the parts of strings that that you need, and use of basic Python structures, in this case dictionaries.
- Please do **not** assume that the input file names will end in .txt. File name suffixes such as .csv and .txt are not mandatory in systems other than Microsoft Windows.
- Please make sure your program does **NOT** call the `input()` or `print()` functions. That will cause your program to hang, waiting for input that my automated testing system will not provide. In fact, what will happen is that the marking program detects the call(s), and will not test your code at all.

Submission:

Stage 1:

Submit a single PDF file containing your approach and/or pseudocode for the solution of the problem as per guidelines discussed in Lecture L2 Software Development Process and Project 1 Stage 1 submission. You need to discuss the document with lab demonstrator before submission. It is mandatory to submit this file before **5:00pm 18 October 2019** on LMS to avoid 10% deduction in Project 2 grading. This will be a formative feedback of your problem solving skills developed in the course. In case you do not submit the file, 10% of the total marks of the project will be deducted from your obtained grade of the Stage 2 submission.

Stage 2:

Submit a single Python (.py) file containing all of your functions via LMS before **5:00pm 25 October 2019** on LMS

You need to contact unit coordinator if you have special considerations or making late submission.

Marking Rubric:

Your program will be marked out of 30 (later scaled to be out of 20% of the final mark).

22 out of 30 marks will be awarded based on how well your program completes a number of tests, reflecting normal use of the program, and also how the program handles various error states, such as the input file not being present. Other than

things that you were asked to assume, you need to think creatively about the inputs your program may face.

8 out of 30 marks will be *style* (4/8) "the code is clear to read" and *efficiency* (4/8) "your program is well constructed and runs efficiently". For style, think about use of comments, sensible variable names, your name at the top of the program, etc. (Please look at your lecture notes, where this is discussed.)

Style Rubric:

0	Gibberish, impossible to understand
1-2	Style is really poor
3	Style is good or very good, with small lapses
4	Excellent style, really easy to read and follow

Your program will be traversing text files of various sizes (possibly including large corpora) so try to minimise the number of times your program looks at the same data items. You may wish to use dictionaries (or sets, if you are prepared to read the documentation), rather than lists.

Efficiency Rubric:

0	Code too incomplete to judge efficiency, or wrong problem tackled
1	Very poor efficiency, additional loops, inappropriate use of `readline()`
2	Acceptable efficiency, one or more lapses
3	Good efficiency, small lapses
4	Excellent efficiency, should have no problem on large files

Automated testing is being used so that all submitted programs are being tested the same way. Sometimes it happens that there is one mistake in the program that means that no tests are passed. If the marker is able to spot the cause and fix it readily, then they are allowed to do that and your - now fixed - program will score whatever it scores from the tests, minus 2 marks, because other students will not have had the benefit of marker intervention. Still, that's way better than getting zero. On the other hand, if the bug is too hard to fix, the marker needs to move on to other submissions.