THE UNIVERSITY OF WESTERN AUSTRALIA

Department of Computer Science and Software Engineering

CITS1401 Computational Thinking with Python

Project 2: Using Stylometry to Verify Authorship

Submission deadline: 11:59pm, Monday 20 May 2019.
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 using *cssubmit*. 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 planet) 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, arg2, normalize=False)`
The first, compulsory argument is the name of a text file with a work to be analysed. The second, compulsory argument will either be the name of a second text file to be analysed or will be the string `listing`. The final optional argument (default value False) is whether the profile values, excluding sentences per paragraph and words per setence (discussed below), are to be normlised.

Output
The output will either be some text with the score from a pairwise comparison, or the listing of the first file's profile. These will be printed on standard output.

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.
- Your program will need to count the number of occcurences of certain words and certain pieces of punctuation. Specifically, the list of words to be counted is:
  `"also", "although", "and", "as", "because", "before", "but", "for", "if", "nor", "of", "or", "since", "that","though", "until", "when", "whenever", "whereas", "which", "while", "yet"`
  If you are wondering why that particular set of words is being used, they are conjunctions, which can indicate more complex sentences without relying on the content of the text.

- Your program should also 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.)
  Finally, 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**

- Each of the words and punctuation symbols should be placed, together with their respective counts, in a dictionary, which I shall call a *profile*.
- You 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.
- If the third, optional parameter in `main` is set to `True`, the profile values are to be normalised. That is, except for the words per sentence and sentences per paragraph parameters, each of the others is to be divided by the number of sentences in the respective text. (Clearly, if two texts are nominated, and normalise is set to `True`, both profiles are to be normalised.)
- If the second argument is the string `listing`, the profile corresponding to the first file should be printed on standard output, one item per line. On the other hand, if the second argument is another text file, the distance between the

corresponding profiles should be computed using the standard distance formula:

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

Example

An example interaction, which you can find here, is based on three files: `sample1.txt` and `sample2.txt`, both excerpts taken from "Life on the Mississippi", by Mark Twain, and `sample3.txt`, which is taken from Banjo Patterson's collection of stories, "Three Elephant Power"..

Some Text Files to Examine

Here are some files for you to try out. All of the texts, apart from "Kangaroo", were obtained from Project Gutenberg. All the files have a long text at the end which contains Project Gutenberg license and terms of use. I have linked the Gutenberg terms and license here 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 Huckleberry 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 (recommended), 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 couple things for your program to avoid.

- Please do **not** import **any** Python module, other than `math`. While use of these modules such `re` is a 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()` function. 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

Submit a single Python (`.py`) file containing all of your functions via *cssubmit*.

Marking Rubric

Once again, your program will be marked out of 20.

- 60% of the marks (12/20) 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.
- 40% (8/20) 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. (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 (possibily 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, addtional 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, right? (On the other hand, if the bug is too hard to fix, the marker needs to move on to other submissions.)