Computing Science 1P/1PX

Laboratory Examination 2 2016-17

20-22 March 2017 **Time allowed:** 1 hour 50 minutes + advance preparation

Weight: this lab exam contributes 10% to the overall assessment for the CS1P course, and 20% to the overall assessment for the CS1PX, respectively.

Threshold: students must attempt at least 75% of all assessments on the CS1P/1PX course in order to gain credit for the course.

Instructions

Before the exam

You should prepare a complete solution to the exam question given in this pack in advance of the exam. Put in as many hours as you need to get a properly working solution **that you fully understand**. You can of course type it into a machine and thoroughly test and debug it.

When you come into the exam, you will be able to bring nothing with you. We will provide you with another copy of the question along with some rough paper to work on. You need to be prepared to reconstruct your solution to the problem during the exam.

You may of course talk to others while preparing for the exam. Remember, however, that you will be on your own in the exam, and so you must understand your solution *thoroughly* before coming to the exam.

During the exam, the machines will be disconnected from the network and from your filestores. Note however that the Python Docs help in the IDLE Help menu will be available. You should familiarise yourself with its contents, in case you get stuck during the exam with Python syntax.

There will be no access to the laboratory, other than to take the lab exam, at any time during the period **20-22 March 2017** inclusive. The lab may also be closed for short periods during the week before and on Thursday/Friday during the lab exam week to allow for preparation by the systems team.

We use a software system called AMS to provide you with template files at the start of the exam, and to collect in your submission at the end of the exam. Please follow the instructions *very carefully* to ensure your submission is delivered to us safely.

Starting the exam

Complete the attendance slip and place it in front of you ***together with your matriculation/student ID card***.

Log in to the system using your special exam username and password as provided on the document in your examination pack. Use AMS to set up LabExam2. This will generate copies of the template program and testing files in your exam account Workspace.

Click on AMS on the Desktop to open it. Select the right course (CS1P or CS1PX) and ensure that LabExam2 is showing. Then press the Setup button. This will generate a copy of the necessary files in your exam account Workspace. ***It is essential that you work on this copy of the program, not a file that you have created yourself. This is particularly relevant to students who are familiar with Jupyter and might be tempted to create an IPython workbook – <u>you should work on (and subsequently submit) the file(s) specified</u>.***

During the exam

As in a normal examination, communication between candidates during the laboratory exam is strictly forbidden.

A candidate who wishes to print a document during the exam should summon an invigilator, who will collect it from the printer and deliver it.

A candidate may leave early, but only after summoning an invigilator, who will ensure that submission has taken place. Any candidate who experiences a hardware or software problem during the examination should summon an invigilator at once.

At the end of the exam

Ensure the program files are exactly as you want them, and in the right location (e.g., CS1P or CS1PX folder in the Workspace folder). Open up AMS again, ensure the right course and LabExam2 are showing, then press Submit. ***Ensure an invigilator has seen the AMS receipt on the screen once submission is complete.***

Leave the laboratory quietly; the exam may still be in progress for other groups.

What you must do and what you must submit

You **may** use any standard Python modules, functions and methods. For example, the `split` function from the `string` module (alternatively, the `split` method of a `string`) and functions from the `re` module are likely to be useful.

You **must** do your work in a single file: **`find_author.py`**

The following files will be available under your AMS setup and have also been made available on Moodle (Lab Exam 2) for you to start work on and check your solution:

- `find_author.py`: Partially complete placeholder for the main program
- `signatures/*.stats`: Signature files to compare against in the `signatures` directory
- `mystery/mystery*.txt`: mystery files to try to guess the authors in the `mystery` directory

Please note that, when you download the different files from Moodle, you will have to then extract/etc. in order to create the above directory hierarchy. Using AMS setup during the exam itself, will generate the above directory structure for you.

Use AMS to submit your assignment. Make sure that the final version of all your code is in the **`find_author.py`** file. If your file is not named exactly as above, your submission **will not get full marks** even if it is otherwise correct.

Lab Exam 2 Problem: An Authorship Detection Program

The purpose of this assignment is to give you more practice writing functions and programs in Python, and in particular writing code with strings, lists, and loops.

You should spend *at least* one hour reading through this handout to make sure you understand what is required, so that you don't waste your time going in circles, or worse yet, hand in something you think is correct and find out later that you misunderstood the specifications. Highlight the handout, scribble on it, and find all the things you're not sure about as soon as possible.

Authorship Detection

Automated authorship detection is the process of using a computer program to analyse a large collection of texts, one of which has an unknown author, and making guesses about the author of that unattributed text. The basic idea is to use different statistics from the text -- called "features" in the machine learning community -- to form a linguistic "signature" for each text. One example of a simple feature is **the number of words per sentence**. Some authors may prefer short sentences while others tend to write sentences that go on and on with lots and lots of words and not very concisely, just like this one. Once we have calculated the signatures of two different texts we can determine their similarity and calculate a likelihood that they were written by the same person.

We have provided the skeleton of a program that guesses the author of a text file based on comparing it to a set of linguistic signatures. Download find_author.py. This program runs, but it subsequently crashes even if a valid filename is supplied, since some functions are called with un-initialised structures which then result in division by zero errors. Your task is to complete the program by filling in the missing pieces.

An Overview: How the program works

The program begins by asking the user for the name of a file of text whose authorship is unknown (the mystery file).

The program calculates the *linguistic signature* for the mystery file and then calculates scores indicating how well the mystery file's signature matches each of a set of equivalent signatures that have been provided for different authors in the `signatures` directory. The author from the signature file that best matches the mystery file is reported.

Definitions

Before we go further, it will be helpful to agree on the following definitions:

- A **token** is a string that you get from calling the string method `split` on a line of the file.
- A **word** is a non-empty token from the file that isn't completely made up of punctuation.

Hint on implementing `get_words`: You'll find the *words* in a file by using `str.split` (on each line) to find the *tokens* and then removing the punctuation from the words using the `clean_up` function that we have provided in `find_author.py`. If after calling `clean_up` the resulting word is an empty string, then it isn't considered a word. Notice that `clean_up` converts the word to lowercase. This means that once they have been cleaned up, the words yes, Yes and YES! will all be the same.

- A **sentence** will be a sequence of characters that:
    1. is terminated by (but does not include) the characters `! ? .` or the end of the file
    2. excludes whitespace on either end, and
    3. is not empty

Consider this file. Remember that a file is just a linear sequence of characters, even though it looks two dimensional. This file contains these characters:

```
this is the\nfirst sentence. Isn't\nit? Yes ! !! This \n\nlast bit :) is also a sentence, but \nwithout a terminator other than the end of the file\n
```

By our definition, there are four "sentences" in it:

Sentence 1	`"this is the\nfirst sentence"`
Sentence 2	`"Isn't\nit"`
Sentence 3	`"Yes"`
Sentence 4	`"This \n\nlast bit :) is also a sentence, but \nwithout a terminator other than the end of the file"`

Notice that:

- The sentences do not include their terminator character.
- The last sentence was not terminated by a character; it finishes with the end of the file.
- Sentences can span multiple lines of the file.

- **Phrases** are defined as non-empty sections of a sentence that are separated by colons, commas, or semi-colons (`: , ;`). The sentence prior to this one has three phrases by our definition. This sentence right here only has one (because we don't separate phrases based on parentheses).

We realize that these are not the correct definitions for sentences, words or phrases but using them will make the assignment easier. So, for the purposes of this assignment, you have to **adhere** to the above definitions **or your assignment will be marked as incorrect.**

Linguistic features to calculate

- **Average word length.** The average number of characters per word, calculated after the punctuation has been stripped using the already-written `clean_up` function. In the sentence prior to this one, the average word length is 6.211. Notice that the comma and the final period are stripped but the hyphen inside "already-written" and the underscore in "clean_up" are both counted. That's fine. You must **not change** the `clean_up` function that does punctuation stripping.

- **Type-Token Ratio (TTR).** The number of *different* words used in a text divided by the *total* number of words. It's a measure of how repetitive the vocabulary is. "This", “this", "this," and "(this" should *not* be counted as different words. Again, this should be easy to achieve if you use the provided `clean_up` function when implementing `get_words`.

- **Hapax Legomena Ratio.** Similar to TTR in that it is a ratio using the total number of words as the denominator. The numerator for the Hapax Legomena Ratio is the number of words occurring *exactly once* in the text.
  Suggestion on how to proceed: As you read the file, keep two lists. The first contains all the words that have appeared *at least once* in the text and the second has all the words that have appeared *at least twice* in the text. Of course, the words on the second list must also appear on the first. Once you've read the whole text, you can use the two lists to calculate the number of words that appeared *exactly* once.

- **Average number of words per sentence**. Self-explanatory feature; *solution already provided for you*

- **Sentence complexity**. The average number of phrases per sentence. We will find the phrases by taking each sentence, as defined above, and splitting it on any of colon, semi-colon or comma.

Since several features require the program to split a string on any of a set of different separators, it makes sense to write a helper function to do this task. To do this, you will complete the function `split_on_separators` as described by the docstring in the code.
Hint: you might want to explore Python’s regular expression (`re`) library which provides functions that will make the implementation of `split_on_separators` trivial.

Reading the (mystery) file and extracting words

A number of linguistic features above require the total number of words in the file as input, whereas others require the total number of sentences (in both cases, following the definitions we have provided above).

For this reason, and in order to avoid having to parse long files multiple times, you are asked to implement a function `get_text_from_valid_file`, which asks the user for a valid filename and it subsequently returns the filename, as well as the entire `text` contained in the file as *a list of strings* (e.g., as returned by the `readlines` method).

Finding the Sentences

Because sentences can span multiple lines of the file, it won't work to process the file one line at a time calling `split_on_separators` on each individual line (or calling `split_on_separators` for each entry of the `text` list returned from `get_text_from_valid_file` above). Instead, create a single huge string that stores the contents of the entire file. Then call `split_on_separators` on that string.

This solution would waste a lot of space for really large files but it will be fine for our purposes. You'll learn about algorithms that can handle large files in future courses.

Signature Files

We have created a set of signature files for you to use that have a fixed format. The first line of each file is the name of the author and the next five lines each contain a single real number. These are values for the five linguistic features in the following order:

- Average Word Length
- Type-Token Ratio
- Hapax Legomena Ratio
- Average Sentence Length
- Sentence Complexity

You are welcome to create additional signature files for testing your code and for fun, but you must not change this format. Our testing of your program will depend on its ability to read the required signature-file format. A function called `read_signature` to read each of the signature files and return a structure for the signature, as well as a routine to read all signature files one by one (by repeatedly calling `read_signature`) *is already provided for you*. Please note that you should *not* change file membership of the `signatures` directory.

Determining the best match

In order to determine the best match between our mystery text and the set of known signatures, the program uses the function `compare_signatures` which calculates and returns a measure of the similarity of two linguistic signatures. The similarity of signatures `a` and `b` will be calculated as the *weighted* sum of the (absolute) differences on each feature. In other words, the similarity of signatures a and b (S_{ab}) is the sum over all five features of: the absolute value of the feature difference times the corresponding weight for that feature. The equation below expresses this definition mathematically:

$$S_{ab} = \sum_{i=1}^{5} |f_{i,a} - f_{i,b}| * w_i$$

where $f_{i,x}$ is the value of feature `i` in signature `x` and w_i is the weight associated with feature `i`.

The example below illustrates. Each row concerns one of the five features. Suppose signature 1 and signature 2 are as shown in columns 2 and 3, and the features are weighted as shown in column 4. The final column shows the contribution of each feature to the overall sum, which is 12. It represents the similarity of signatures 1 and 2.

Feature number	Value of feature in signature 1	Value of feature in signature 2	Weight of feature	Contribution of this feature to the sum
1	4.4	4.3	11	`abs(4.4-4.3)*11 = 1.1`
2	0.1	0.1	33	`abs(0.1-0.1)*33 = 0`
3	0.05	0.04	50	`abs(0.05-0.04)*50 = .5`
4	10	16	0.4	`abs(10-16)*0.4 = 2.4`
5	2	4	4	`abs(2-4)*4 = 8`
SUM				`12`

Notice that if signatures 1 and 2 were exactly the same on every feature, the similarity would add up to zero. (It may have made sense to call this "difference" rather than similarity.) Notice also that, if two signatures are different on a feature that is weighted higher, their overall similarity value goes up more than if they are different on a feature with a low weight. This is how weights can be used to tune the importance of different features.

Note that a function to compare two signatures (`compare_signatures`), as well as a vector of `weights` have *already been provided for you*.

Additional requirements

- Where a docstring says a function can assume something about a parameter (e.g., it might say "text is a non-empty list") the function should not check that this thing is actually true. Instead, when you call the function make sure that it is indeed true.
- We suggest that you do not change any of the existing code. Add to it as specified in the comments.
- Do not add any user input or output, except where you are explicitly told to. In those cases, we have provided the exact `print` or `raw_input` statement to use. Do not modify these.

Suggestions on how to proceed

- Read this handout thoroughly and carefully, making sure you understand everything in it, particularly the different linguistic features.
- Read the starter code to get an overview of what you will be writing. Draft a plan for your code to get an understanding of how everything fits together.
- To avoid getting overwhelmed, deal with one function at a time. Start with functions that don't call any other functions; this will allow you to test them right away.
- You have overall eight (8) functions to write/complete, while another four (4) plus the main section of the program have been provided for you. Arguably, the most important ones to make sure they adhere to the specification are the `get_words` and `get_sentences` functions, since the rest depend on what these two return. A significant fraction of the rest of the functions are simple calculations on the values of the structures (lists) returned by these two functions: the total words and the total sentences of the (mystery) file.
- Most of the functions you are asked to write should only be a few lines long, so if you find yourself writing lots of code for any one of them, then you probably want to revisit your plan.
- For each function that you write, plan test cases for that function and actually write the Python code to implement those tests *before* you write the function. It is hard to have the discipline to do this, but it will help you understand the problem faster if you do.
- Once you have implemented all the functions, run the `a2_self_test` module (see below) to confirm that your code passes some basic tests. Correct any errors that this uncovers. But, of course, if you did a great job of testing your functions as you wrote them, you won't find any new errors now.
- You should now be ready to run the full author detection program: run `find_author.py`. To do this, you will need to set up a directory hierarchy as shown in the section "*What you must do and what you must submit*" above (at the time of the exam in the lab machines, AMS will replicate this setup for you). Signatures as well as mystery files are being provided for you on Moodle

Testing your code

We are providing a module called a2_self_test that imports your `find_author` and checks that most of the required functions satisfy some of the basic requirements. It does not check the `get_text_from_valid_file` function and only implicitly checks `get_words` and `get_sentences`.

When you run `a2_self_test.py`, it should produce no errors and its output should consist only of one thing: the word "okay". If there is any other output at all, or if any input is required from the user, then your code is not following the assignment specifications correctly and will not get full marks. You should go back and fix the error(s).

While you are playing with your program, you may want to use the signature files and mystery text files we have provided. This, in conjunction with `a2_self_test` is still **not** sufficient testing to thoroughly test all of your functions under all possible conditions. With the functions on this assignment, there are many more possible cases to test (and cases where your code could go wrong). If you want to get a great mark on the correctness of your functions, do a great job of testing them under all possible conditions. Then we won't be able to find any errors that you haven't already fixed!

Marking guidelines

The following aspects of the program will be taken into account in marking:

- Correctness: Your code should perform as specified.
- Use of appropriate and correct algorithms and Python control structures
- Program efficiency
- Correct Python syntax
- Good programming style, including layout (clear and simple code), use of explanatory comments (the more complicated parts of your code), choice of identifiers (meaningful variable names), and appropriate use of functions
- Appropriate error checking (when required)
- Good use of helper functions (when required): If you find yourself repeating a task, you should add a helper function and call that function instead of duplicating the code. And if a function is more than about 20 lines long, consider introducing helper functions to do some of the work -- even if they will only be called once.

Future Thoughts

If you carefully examine the mystery files and correctly code up your assignment, you'll see that it correctly classifies many of the documents -- but not all. Some of the linguistic features that we have used (type-token ratio and hapax legomena ratio in particular) are standard techniques, but they may not be sufficient to do the classification task. There are other standard features and more sophisticated techniques that were too complicated to consider for the scope of this assignment.