CS 101
Fall 2013
Program 6
Algorithm due November 3
Program due November 10

Who Is The Author Of This Text?

NOTE: Read through this assignment carefully, AT LEAST twice, to be sure you understand what's being asked. Be sure you understand what's required so you don't waste time going in circles or handing in something you think is correct and finding out the hard way that you misunderstood the specification. Ask questions in class or on the Piazza site about anything you're not sure about.

Authorship Detection

Authorship detection is the process of using a program to analyze a large collection of texts, one of which has an unknown author, and trying to determine the author of the unknown text. This is done by analyzing various statistical features of the text to form a linguistic 'signature.' For example, one feature is the number of words per sentence. Hemingway wrote in short sentences. He was direct. Blunt, even. His prose wasn't flowery. Faulkner, on the other hand, wrote sentences that went on over a full page, making digressive references to half-remembered bygones with an air of sad nostalgia, like the wind coming over the fields just before harvest, which for some reason always reminds you of your Aunt Lavinia's pancakes, which is odd because the fields don't particularly smell like pancakes and you never really liked Aunt Lavinia's cooking anyway, but there you have it, and that's all there is to it, so there's really nothing else to say, at least not about Lavinia's cooking, but there's still plenty to say about Faulkner and how long his sentences were, which, if you haven't read Faulkner, can be surprisingly, even tediously, long. (And now you know why I teach computer science and not creative writing.) Once we've calculated the signatures of two different texts we can determine their similarity and estimate how likely it is they were written by the same person.

This is used in plagiarism detection, email filtering, social science research, as forensic evidence in court cases, and in historical research. (Many texts are of uncertain authorship, or are the result of two more more earlier texts being combined, an early example of a mashup.) Professional methods of authorship detection are a good deal more complicated than we're going to use here, but even with the simple methods we're using here, your program can make some reasonable predictions about authorship.

You are provided with a skeleton program containing stubs for most of the functions you will need, with part of the main program logic already completed. The program runs but doesn't do much of anything. You will need to complete the program by filling in the missing pieces.

The program will ask the user for the name of a file containing an unknown text to be analyzed. Your program will compute the linguistic signature for the mystery file, compare it to the profiles of all known authors, and report which author the text most closely matches.

Development notes:

- You're provided with several mystery texts from Project Gutenberg. Use others if you like, though they should be plain text files. Put these files into the same folder as your program.
- You're also provided with a zipfile containing several text files. Each of these files has the signature for a different author, based on several of that author's works. Unzip the file and put

the signature files in the a subfolder (subdirectory) inside the folder containing your program.

- Your program will ask the user for the name of the folder containing the signature files, verify that it exists and is a directory, and move to that directory to read the signature files, moving back to the same folder as it started when finished. Your program should assume that the subdirectory containing the signature files contains *only* signature files.
- See below for a discussion of how the various features are computed.

What's a sentence?

Before we go further, it will be helpful to agree on what we will call a sentence, a word and a phrase. Let's define a **token** to be a string that you get from calling the string method `split` on a line of the file. We define a **word** to be a non-empty token from the file that isn't completely made up of punctuation. You'll find the "words" in a file by using str.split to find the tokens and then removing the punctuation from the words using the clean_up function provided for you. 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.

For the purposes of this assignment, we will consider a **sentence** to be a sequence of characters that (1) is terminated by (but doesn't include) the characters `! ? .` or the end of the file, (2) excludes whitespace on either end, and (3) is not empty. Consider this file:

```
this is the
first sentence. Isn't
it? Yes ! !! This

last bit :) is also a sentence, but
without a terminator other than the end of the 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 terminato 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 sentences 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. More importantly, it will make your results match what we are expecting when we test your code. You may not "improve" these definitions or your assignment will be marked as incorrect.

Linguistic features we will calculate

The first linguistic feature recorded in the signature is the **average word length**. This is simply 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 5.909. 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 is 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. Again you must use the provided clean_up function so that "this","This","this," and "(this" are **not** counted as different words.

Hapax Legomana Ratio is similar to Type-Token Ratio in that it is a ratio using the total number of words as the denominator. The numerator for the Hapax Legomana Ratio is the number of words occurring exactly once in the text. In your code for this function you must use a Python dictionary.

The fourth linguistic feature your code will calculate is the **average number of words per sentence**.

The final linguistic feature is a measure of **sentence complexity** and is 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.

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. Instead, create a single huge string that stores 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.

There are other ways where our assignment design isn't very efficient. For example, having the different linguistic features calculated by separate functions means that our program has to keep going over the text file doing many of the same actions (breaking it into words and cleaning them up) for each feature. This is inefficient if we are certain that anyone using our code would always be calculating all the features. However, our design allows another program to import our module and efficiently calculate a single linguistic feature without calculating the others. It also makes the code easier to understand, which in today's computing environment is often more important than efficiency.

Signature Files

The signature files each 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 Legomana 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.

Determining the best match

In order to determine the best match between an unattributed text and the known signatures, the program uses the function `compare_signatures` which calculates and returns a measure of the similarity of two linguistic signatures. You could imagine developing some complicated schemes but our program will do almost the simplest thing imaginable. The similarity of signatures a and b will be calculated as the sum of the differences on each feature, but with each difference multiplied by a "weight" so that the influence of each feature on the total score can be controlled. 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 16.5. 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

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 they are different on a feature that is weighted higher, there 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.

You are required to complete function compare_signatures according to the docstring and you must not change the header. Notice that the list of weights is provided to the function as a parameter. We have already set the weights that your program will use (see the main block) so you don't need to play around trying different values.

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.
- 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 input statement to use. Do not modify these.
- Functions get_valid_filename and read_directory_name must not use any for loops, or they will lose points. The purpose of this is to make sure you get comfortable with while loops.
- You must not use any break or continue statements. Any functions that do will cost you points. We are imposing this restriction because they are very easy to abuse, resulting in terrible code.

How to tackle this assignment

This program is by far the largest we have done so far. You'll need a good strategy for how to tackle it. Here is our suggestion.

Principles:

- 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. The steps listed below give you a reasonable order in which to write the functions.
- 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 you will have a **huge** advantage over your peers if you do.
- Keep in mind throughout that any function you have might be a useful helper for another function. Part of your marks will be for taking advantage of opportunities to call an existing function.
- As you write each function, begin by designing it in English, using only a few sentences. If your design is longer than that, shorten it by describing the steps at a higher level that leaves out some of the details. When you translate your design into Python, look for steps that are described at such a high level that they don't translate directly into Python. Design a helper function for each of these, and put a call to the helpers into your code. Don't forget to write a great docstring for each helper!

Submitting your assignment

This assignment will be completed on your own without a partner.

Submit your find_author.py file (which must be named that) and nothing else to the Blackboard site.

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 legoamana 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 for this assignment. Suppose you used a dictionary to keep a count of how many times each word appeared in a document. What new linguistic features would that allow you to use? Some authors like to use lots of exclamation marks!!! or perhaps just a lot of punctuation?! Could you devise a linguistic feature to measure this? While this is fun to think about, do **not** add any different linguistic features to the version of the program that you submit for marking.

Another area for thought is how authorship attribution is related to plagiarism detection. Plagiarism detection software has also been used to support the claim that Shakespeare was the author of an unattributed play.

In the field of machine learning (of which authorship detection is a subfield), programs often learn their own configuration values through training. In our case, could the program learn from trying to guess an author and then being told the right answer? Could it learn to adjust the weights being applied to the different features? What about learning a new feature? How would you do this?

Sample run:

```
>>> ================================ RESTART ================================
>>>
enter the name of the file with unknown author:mystery1.txt
enter the path to the directory of signature files: Stats
best author match: jane austen
 with score 2.6188384274162946
>>>
```