PROGRAMMING I
ASSIGNMENT 4, DUE DATE DECEMBER 20, 2014 - 11:59PM

Authorship Detection

December 16, 2014

1 AUTOMATED AUTHORSHIP

Automated authorship detection is the process of using a computer program to analyze 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 are not very concise, 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.
Automated authorship detection has uses in plagiarism detection, email-filtering, social-science research, and as forensic evidence in court cases. Also called *authorship attribution*, this is a current research field and the state-of-the-art linguistic features are considerably more complicated than the five simple features that we will use for our program. But even with our very basic features and simplifications, your program may still be able to make some reasonable predictions about authorship.

2 STRING FUNCTIONS

Python provides a lot of built-in string methods. To get a list of all the methods type `dir(str)` in the REPL. To get help on a particular method (for example strip) you can use the help function `help(str.strip)`. Study how the `lower, strip, join, replace` and `split` meth-

ods work. There are also lots of tutorials available on the internet. Two reasonably good ones are:
`http://www.tutorialspoint.com/python/python_strings.htm`)
`http://zetcode.com/lang/python/strings/`
The official documentation for strings is at:
`https://docs.python.org/3/library/string.html`
Section 6.1 of the above contains some useful constants which you can use in your program. Run the following code to display some of the constants:

```
>>> import string
>>> string.digits
'0123456789'
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

3 Definitions

Whitespace is any character or series of whitespace characters that represent horizontal or vertical space on a screen. When rendered, a whitespace character does not correspond to a visible mark, but typically does occupy an area on the screen. For example, the tab character, represented by `\t` in a Python string, has no visible representation on the screen.

Token Consider the following string:

```
>>> myStr = "this is the\nfirst sentence. Isn't\nit? Yes ! !!
    This \n\nlast bit"
```

If we use `split()` on this (with the default whitespace separator) we get a list of strings:

```
>>> myStr.split()
['this', 'is', 'the', 'first', 'sentence.', "Isn't", 'it?',
'Yes', '!', '!!', 'This', 'last', 'bit']
```

Each string in this list is called a **token**. Therefore tokens are strings that you get when you separate an input text using some separator (whitespace in this case)

A Word is a non-empty token from the file that isn't completely made up of punctuation. You can get the words in a string by first splitting the string into tokens and then removing any punctuation from the beginning and end of the tokens. The non-empty strings that remain are classified as words. There is a function called `clean_up` provided to you in the `author_functions.py` file that you can use for this. Notice that `clean_up` also converts the string to lowercase. This means that the cleaned up versions of 'yes', 'Yes' and 'YES!' will all be the same word. Words may contain inner punctuation; for example, "it's" is a word with a length of 4 and 'hard-and-fast' is a word with a length of 13.

A Sentence is a string that:

- is followed by (but doesn't include) the terminator characters ! ? .
- excludes whitespace on either end, and
- is not empty.

Consider the following string:

```
>>> "this is the\nfirst sentence. Isn't\nit? Yes ! !!
    This \n\nlast bit"
```

By our definition there are four sentences in the above string:

Sentence 1: `"this is the\nfirst sentence"`
Sentence 2: `"Isn't\nit"`
Sentence 3: `"Yes"`
Sentence 4: `"This \n\nlast bit"`

A phrase is a non-empty section of a sentence that is separated by colons, commas, or semi-colons. The previous sentence (the one before this one) has three phrases by our definition. This sentence (the one you are reading right now) has only one. This is because we don't separate phrases based on parentheses.

4 Completing `AUTHOR_FUNCTIONS.PY`

In `author_functions.py`, you must implement five required functions to calculate various linguistic features and a required function to compare authors based on these linguistic features. In addition, you must add some helper functions (one required, plus others that you choose to add) to aid with the implementation of these functions. In addition to the descriptions below, please see the docstring descriptions provided in the starter code. Add a second example function call to each of the given docstring descriptions in `author_functions.py`.

1. `avg_word_length`: The first linguistic feature 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 6.04. 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.

   The function takes a list of strings as input. The strings have a newline `\n` character as their last character. The output is the average word length of all the words in the input list. One way to solve this is to use two nested `for-loops`. The outer `for-loop` loops over each string in the list. It cleans up each string splits it into another list (consisting of individual words) using the `split` function. The inner for-loop iterates over this new list to calculate the average word length.

2. `type_token_ratio`: The second linguistic feature, the Type-Token Ratio, is the number of different words used in a text divided by the total number of words in that text. (This feature is a measure of the repetitiveness of the vocabulary.) Again you must use the provided clean_up function so that "this","This","this," and "(this" are considered the same and are not counted as different words.

   To implement this function use a dictionary. The dictionary stores the words and the number of times they occur in the input as key:value pairs. The length of the dictionary then gives the number of different words used and the sum of the values gives the total number of words used.

3. `hapax_legomena_ratio`: The third linguistic feature, the Hapax Legomena Ratio, is similar to Type-Token Ratio in that it is a ratio that uses the total number of words as the denominator. The numerator for the Hapax Legomena Ratio is the number of words that occur exactly once in the text. In your code for this function **you must not use a Python dictionary** or any other technique that keeps a count of the frequency of each word in the text. Instead, use this approach: As you process the text, 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.

   Note: you will be marked not only on correctness, but also on whether you follow the approach specified above.

4. `split_on_separators`: This is a helper function. Several features require the program to split a string on any of a set of different separators, and this function will be used to do that.

   Test this function carefully on its own before trying to use it in your other functions.

5. `avg_sentence_length`: The fourth linguistic feature your code will calculate is the average number of words per sentence.

   Tip: Sentences can span multiple lines. Rather than work with the given list of str (list of lines), start by creating a single huge string containing all of the strings, in order, from the list of lines. Next, call function split_on_separators on that huge string.

6. `avg_sentence_complexity`: The final linguistic feature is sentence complexity, which is measured by 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.

Tip: Sentences can span multiple lines. Rather than work with the given list of str (list of lines), start by creating a single huge string containing all of the strings, in order, from the list of lines. Next, call function split_on_separators on that huge string.

7. `compare_signatures`: 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. 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 absolute 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:

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

5 How to tackle this assignment

This program is much larger than what you have done previously, so you'll need a good strategy for how to tackle it. Here is our suggestion.

5.1 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, start by adding at least one example call to the docstring before you write the function.
- 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!

5.2 Steps

Here is a good order in which to solve the pieces of this assignment.

a) Read this handout thoroughly and carefully, making sure you understand everything in it, particularly the different linguistic features.

b) Read the starter code to get an overview of what you will be writing. It is not necessary at this point to understand every detail of the functions we have provided.

c) Complete `author_functions.py`: add example(s) to the docstring, implement, and test the functions in this order:

   - `avg_word_length`
   - `type_token_ratio`
   - `hapax_legomena_ratio`
   - `split_on_separators`
   - `avg_sentence_length`. Begin by writing code to obtain a list of sentences from the text. Test that this part of your code works correctly before you worry about calculating the average sentence length.
   - `avg_sentence_complexity`
   - `compare_signatures`