Project 1: TextAnalyzer: 300 Points

Due Feb 21

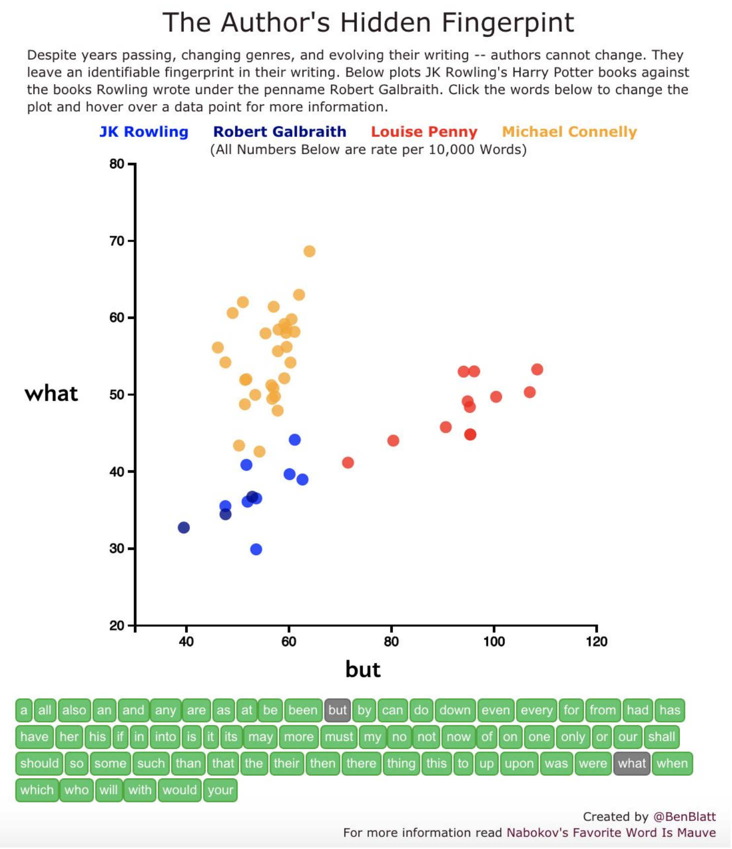
Introduction

What if there was a way to identify the author of an anonymous text? In this project, we will build an infrastructure for analyzing texts so they can be compared for similarity.

Data scientists have successfully achieved this by comparing the frequency of common words in an author's writings. These frequencies form a descriptor of an author's style, which tends to stay constant across their works. We can compare the frequencies of different writings, to see which writings are similar.

This method was able to identify "Robert Galbraith" as the pen name of JK Rowling (note the overlap of the light blue and dark blue dots below).

(To try different word combinations on this graph, see Can You Identify an Author By How Often They Use the Word "The"?)



The Author's Hidden Fingerpint

Despite years passing, changing genres, and evolving their writing -- authors cannot change. They leave an identifiable fingerprint in their writing. Below plots JK Rowling's Harry Potter books against the books Rowling wrote under the penname Robert Galbraith. Click the words below to change the plot and hover over a data point for more information.

(All Numbers Below are rate per 10,000 Words)

Created by @BenBlatt

For more information read Nabokov's Favorite Word Is Mauve

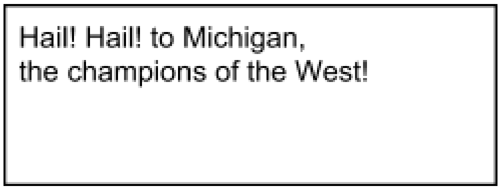
JK Rowling
Robert Galbraith
Louise Penny
Michael Connelly
what
80
70
60
50
40
30
20
40
60
80
100
120
but
a all also an and any are as at be been but by can do down even every for from had has
have her his if in into is it its may more must my no not now of on one only or our shall
should so some such than that the their then there thing this to up upon was were what when
which who will with would your

In this project...

In this project, you will develop a TextAnalyzer class. A TextAnalyzer object will read in a file and do all of the analysis needed to create the frequency “fingerprint” for that text.

Here’s an example of how the TextAnalyzer works, using a very short text.

File fightsong.txt



```
Hail! Hail! to Michigan,
the champions of the West!
```

```
>>> fightsong = TextAnalyzer("files_for_testing/fightsong.txt")
>>> fightsong.line_count()
2
>>> fightsong.word_count()
9
>>> fightsong.vocabulary()
['champions', 'hail', 'michigan', 'of', 'the', 'to', 'west']
>>> fightsong.frequencies()
{'hail': 2, 'to': 1, 'michigan': 1, 'the': 2, 'champions': 1, 'of':
1, 'west': 1}
>>> fightsong.frequency_of('champions')
1
>>> fightsong.frequency_of('ohio')
0
>>> fightsong.most_common()
'hail'
>>> fightsong.most_common(2)
['hail', 'the']
>>> fightsong.percent_frequencies()
{'hail': 0.2222222222222222, 'to': 0.1111111111111111, 'michigan':
0.1111111111111111, 'the': 0.2222222222222222, 'champions':
0.1111111111111111, 'of': 0.1111111111111111, 'west':
0.1111111111111111}
```

The code

You will create a class called TextAnalyzer with the following methods. Start from the stub code provided, which looks like the following:

```
class TextAnalyzer:
    def __init__(self, filepath):
        """Initializes the TextAnalyzer object, using the file at filepath"""
    def line_count(self):
        """Returns the number of lines in the file"""
    def word_count(self):
        """Returns the number of words in the file. A word is defined as any
        text that is separated by whitespace (spaces, newlines, or tabs) or
        followed by punctuation (like ? or !)."""
    def vocabulary(self):
        """Returns a list of the unique words in the text, sorted in
        alphabetical order. Capitalization should be ignored, so 'Cat' is the
        same word as 'cat'. The returned words should be all lower-case."""
    def num_unique_words(self):
        """Returns the number of unique words in the text. Capitalization
        should be ignored, so 'Cat' is the same word as 'cat'."""
    def frequencies(self):
        """Returns a dictionary of the words in the text and the count of how
        many times they appear. The words are the keys, and the counts are the
        values. All the words should be lowercase. The order of the keys
        doesn't matter."""
    def frequency_of(self, word):
        """Returns the number of times word appears in the text. Capitalization
        should be ignored, so 'Cat' is the same word as 'cat'."""
    def percent_frequencies(self):
        """Returns a dictionary of the words in the text and the fraction of
        the text. The words are the keys, and the counts are the
        values. All the words should be lowercase. The order of the keys
        doesn't matter."""
    def most_common(self, n=1):
        """Returns a list of the most common n words in the text. By default,
        n is 1. The returned words should be in alphabetical order.
        There might be a case where multiple words have the same frequency,
        but you can only return some of them due to the n value. In that case,
        return the ones that come first alphabetically."""

    def similarity_with(self, other_text_analyzer, n=10):
        """Extra credit. Calculates the similarity between this text and
        the other text using cosine similarity. See the Extra Credit section
       of the project specification for details."""
```

Grading

There are unit tests included in the stub code that test each method. We will use the same tests that we provide to you in order to calculate your final grade.

method	***points***
`def line_count(self):`	15
`def word_count(self):`	30
`def vocabulary(self):`	30
`def num_unique_words(self):`	15
`def frequencies(self):`	75
`def frequency_of(self, word):`	15
`def percent_frequencies(self):`	45
`def most_common(self, n=1):`	75
total	**300**
`def similarity_with(self, other_text_analyzer, n=10):`	15 pts extra credit

If all the the unit tests about that method pass, you get all the points for that method! If only some of the tests pass, you get a fraction of the points for that method. For example, if 2 out of the 3 tests related to line_count() pass, then you get ⅔ of the possible points for line_count (10 points out of 15 points).

Tips

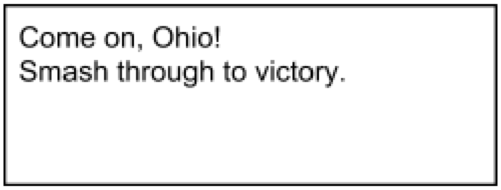
Work on one method at a time. Choose the one that you think is the easiest, and work on it until you can get all the tests related to that method to pass. This is a great strategy, since ***the solution to some methods can be used to quickly complete other methods.***

Make sure you are using Python 3!! Some of the tests won't pass if you are using Python 2.

Extra Credit: Calculating similarity

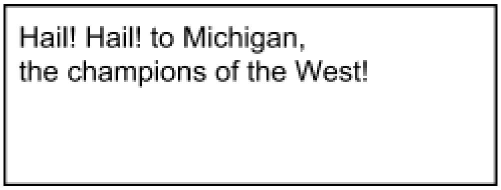
Now let's see how one text compares to another text. Here are two different texts:

File osusong.txt



```
Come on, Ohio!
Smash through to victory.
```

File fightsong.txt



```
Hail! Hail! to Michigan,
the champions of the West!
```

One way to measure their similarity is to compare the percent frequencies of the different words in these texts. We can use the percent frequency that's calculated by the TextAnalyzer, but let's make sure that the words are the same in each.

word	Percent frequency
`'champions'`	`0`
`'come'`	`0.1428571`
`'hail'`	`0`
`'michigan'`	`0`
`'of'`	`0`
`'ohio'`	`0.1428571`
`'on'`	`0.1428571`
`'smash'`	`0.1428571`
`'the'`	`0`
`'to'`	`0.1428571`
`'through'`	`0.1428571`
`'victory'`	`0.1428571`
`'west'`	`0`

word	Percent frequency
`'champions'`	`0.1111111`
`'come'`	`0`
`'hail'`	`0.2222222`
`'michigan'`	`0.1111111`
`'of'`	`0.1111111`
`'ohio'`	`0`
`'on'`	`0`
`'smash'`	`0`
`'the'`	`0.1111111`
`'to'`	`0.2222222`
`'through'`	`0`
`'victory'`	`0`
`'west'`	`0.1111111`

Only the word 'to' is included in both, so we don't expect these texts to be very similar!

These frequencies create a sort of vector for each text. We can measure the similarity of two vectors using something called the *cosine similarity*. So, we can measure the similarity between two texts using the cosine similarity as well.

The cosine similarity of two vectors is:

*The Dot product of the two vectors / (Magnitude of the first vector * Magnitude of the second vector)*

How to calculate the dot product:

For each word, multiply the percent frequency from text 1 with the percent frequency from text 2. Add up all those values.

So, for the texts in this example, it would be:

0 * 0.111111 + 0.1428571 * 0 + 0 * 0.222222 + 0 * 0.111111 + …. + 0 * 0.111111 = 0.0317
('champions') ('come') ('hail') ('michigan') ('west')

(Tip: Note that only the words that both texts have in common actually matter in this calculation)

How to calculate the magnitude:

For one text, add up all of the frequencies squared. Then, take the square root of that.

So, for fightsong.txt, it would be:

sqrt(0.1111111^2 + 0^2 + 0.222222^2 + 0.111111^2 + …. + 0.111111^2) = 0.1605
('champions') ('come') ('hail') ('michigan') ('west')

(Tip: Note that only the words that occur in that particular text actually matter in this calculation)

Cosine similarity returns a number between 0 and 1, where 1 means the texts are identical, and 0 means that the texts are entirely different.

You will implement this calculation in the similarity_to() method.

```
>>> fightsong.similarity_to(osusong, 7)
0.10482848367219184
```

This method also takes an optional parameter `n` that determines the number of words from each text to include in this calculation. By default, n is 10. The n most common words from each text should be used in the cosine similarity calculation. In the example above, the 7 most common words from each text were used. There are only 7 unique words in each text, so we used all the words available.

Miscellaneous

Useful string methods: split() and strip()

```
>>> s = “I love cats. I love every kind of cat!\n”
>>> s. split()
['I', 'love', 'cats.', 'I', 'love', 'every', 'kind', 'of', 'cat!']
>>> s = 'cats.'
>>> s.strip(“.!”)
'cats'
>>> s = 'cat!'
>>> s.strip(“.!”)
'cat'
```

Useful function: sorted()

```
>>> l = ['love', 'every', 'kind', 'of', 'cat']
>>> sorted(l)
['cat', 'every', 'kind', 'love', 'of']
>>> sorted(l, reverse = True)
['of', 'love', 'kind', 'every', 'cat']
>>> sorted(l, key = lambda x : x[-1]) # sort by the last letter
['kind', 'love', 'of', 'cat', 'every']
```

According to the Python documentation: *“It is best to think of a dictionary as an unordered set of key: value pairs”*. As of Python 3.7 and above, the keys do preserve an order, which is the insertion order. But, we will not evaluate you on the order of your dictionaries.