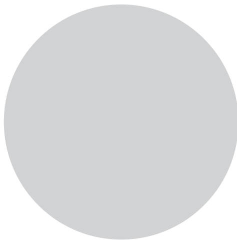
2

ATTRIBUTING AUTHORSHIP WITH STYLOMETRY



Stylometry is the quantitative study of literary style through computational text analysis. It's based on the idea that we all have a unique, consistent, and recognizable style to our writing. This includes our vocabulary, our use of punctuation, the average length of our sentences and words, and so on.

A common application of stylometry is authorship attribution. Do you ever wonder if Shakespeare really wrote all his plays? Or if John Lennon or Paul McCartney wrote the song *In My Life*? Could Robert Galbraith, author of *A Cuckoo's Calling*, really be J. K. Rowling in disguise? Stylometry can find the answer!

Stylometry has been used to overturn murder convictions and even helped identify and convict the Unabomber in 1996. Other uses include detecting plagiarism and determining the emotional tone behind words, such as in social media posts. Stylometry can even be used to detect signs of mental depression and suicidal tendencies.

In this chapter, you'll use multiple stylometric techniques to determine whether Sir Author Conan Doyle or H. G. Wells wrote the novel *The Lost World*.

Project #2: The Hound, The War, and The Lost World

Sir Author Conan Doyle (1859–1930) is best known for the Sherlock Holmes stories, considered milestones in the field of crime fiction. H. G. Wells (1866–1946) is famous for several groundbreaking science-fiction novels including *The War of The Worlds*, *The Time Machine*, *The Invisible Man*, and *The Island of Dr. Moreau*.

In 1912, the *Strand Magazine* published *The Lost World*, a serialized version of a science-fiction novel. It told the story of an Amazon basin expedition, led by zoology professor George Edward Challenger, that encountered living dinosaurs and a vicious tribe of ape-like creatures.

Although the author of the novel is known, for this project let's pretend it's in dispute and it's your job to solve the mystery. Experts have narrowed the field down to two authors, Doyle and Wells. Wells is slightly favored because *The Lost World* is a work of science fiction, which is his purview. It also includes brutish ape-like troglodytes redolent of the morlocks in his 1895 work, *The Time Machine*. Doyle, on the other hand, is known for detective stories and historical fiction.

THE OBJECTIVE

Write a Python program that uses stylometry to determine whether Sir Arthur Conan Doyle or H. G. Wells wrote the novel *The Lost World*.

The Strategy

The science of *natural language processing (NLP)* deals with the interactions between the precise and structured language of computers and the nuanced, frequently ambiguous "natural" language used by humans. Example uses for NLP include machine translations, spam detection, comprehension of search engine questions, and predictive text recognition for cell phone users.

NLP tests for attributing authorship commonly analyze the following features of a text:

- **Word length** A frequency distribution plot of the length of words in a document

- **Stop words** A frequency distribution plot of stop words (short, noncontextual function words like *the*, *but*, and *if*)
- **Parts of speech** A frequency distribution plot of words based on their syntactic functions (such as nouns, pronouns, verbs, adverbs, adjectives, and so on)
- **Most common words** A comparison of the most commonly used words in a text
- **Jaccard similarity** A statistic used for gauging the similarity and diversity of a sample set

If Doyle and Wells have distinctive writing styles, these five tests should be enough to distinguish between them. We'll talk about each test in more detail in the coding section.

To capture and analyze each author's style, you'll need a representative *corpus*, or a body of text. For Doyle, use the famous Sherlock Holmes novel *The Hound of the Baskervilles*, published in 1902. For Wells, use *The War of the Worlds*, published in 1898. Both these novels contain more than 50,000 words, more than enough for a sound statistical sampling. You'll then compare each author's sample to *The Lost World* to determine how closely the writing styles match.

To perform stylometry, you'll use the *Natural Language Toolkit (NLTK)*, a popular suite of programs and libraries for working with human language data in Python. It's free and works on Windows, macOS, and Linux. Created in 2001 as part of a computational linguistics course at the University of Pennsylvania, NLTK has continued to develop and expand with the help of dozens of contributors. To learn more, check out the official NLTK website at *http://www.nltk.org/*.

Installing NLTK

You can find installation instructions for NLTK at *http://www.nltk.org/install.html*. To install NLTK on Windows, open PowerShell and install it with Preferred Installer Program (pip).

```
python -m pip install nltk
```

If you have multiple versions of Python installed, you'll need to specify the version. Here's the command for Python 3.7:

```
py -3.7 -m pip install nltk
```

To check that the installation was successful, open the Python interactive shell and enter the following:

```
>>> import nltk
>>>
```

If you don't get an error, you're good to go. Otherwise, follow the installation instructions at *http://www.nltk.org/install.html*.

Downloading the Tokenizer

To run the stylometric tests, you'll need to break the multiple texts—or *corpora*—into individual words, referred to as *tokens*. At the time of this writing, the `word_tokenize()` method in NLTK implicitly calls `sent_tokenize()`, used to break a corpus into individual sentences. For handling `sent_tokenize()`, you'll need the *Punkt Tokenizer Models*. Although this is part of NLTK, you'll have to download it separately with the handy NLTK Downloader. To launch it, enter the following into the Python shell:

```
>>> import nltk
>>> nltk.download()
```

The NLTK Downloader window should now be open (Figure 2-1). Click either the **Models** or **All Packages** tab near the top; then click **punkt** in the Identifier column. Next, scroll to the bottom of the window and set Download Directory to the Windows default by entering **C:\nltk_data**. Finally, click the **Download** button to download the Punkt Tokenizer Models.

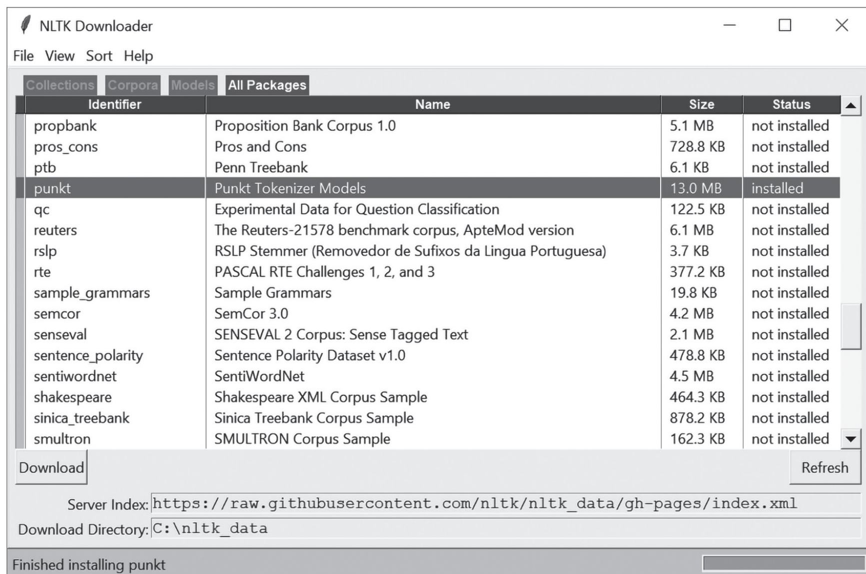


Figure 2-1: Downloading the Punkt Tokenizer Models

Note that you can also download NLTK packages directly in the shell. Here's an example:

```
>>> import nltk
```

```
>>> nltk.download('punkt')
```

You'll also need access to the Stopwords Corpus, which can be downloaded in a similar manner.

Downloading the Stopwords Corpus

Click the **Corpora** tab in the NLTK Downloader window and download the Stopwords Corpus, as shown in Figure 2-2.

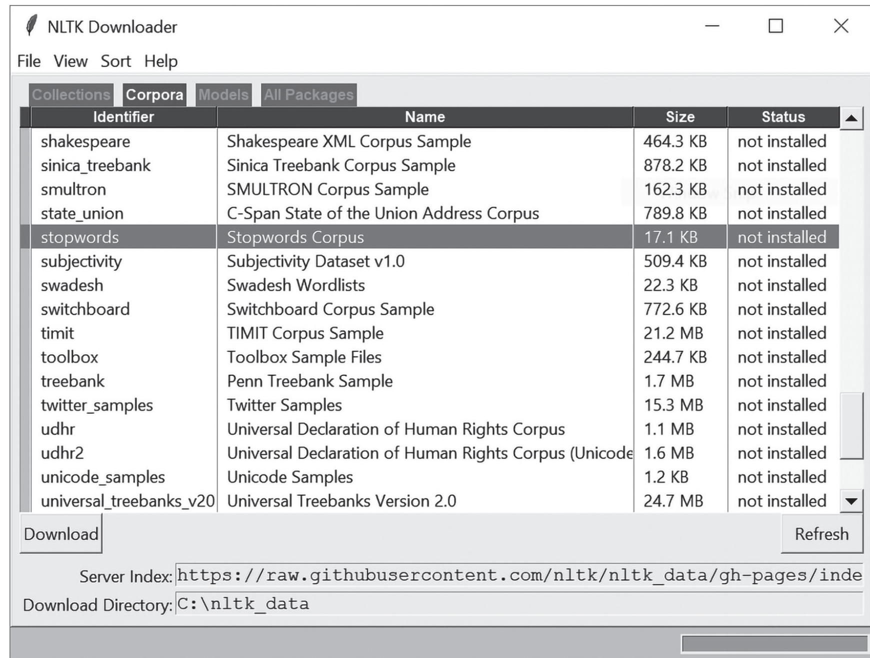


Figure 2-2: Downloading the Stopwords Corpus

Alternatively, you can use the shell.

```
>>> import nltk
```

```
>>> nltk.download('stopwords')
```

Let's download one more package to help you analyze parts of speech, like nouns and verbs. Click the **All Packages** tab in the NLTK Downloader window and download the Averaged Perceptron Tagger.

To use the shell, enter the following:

```
>>> import nltk
```

```
>>> nltk.download('averaged_perceptron_tagger')
```

When NLTK has finished downloading, exit the NLTK Downloader window and enter the following into the Python interactive shell:

```
>>> from nltk import punkt
```

Then enter the following:

```
>>> from nltk.corpus import stopwords
```

If you don't encounter an error, the models and corpus successfully downloaded.

Finally, you'll need `matplotlib` to make plots. If you haven't installed it already, see the instructions for installing scientific packages on page 7.

The Corpora

You can download the text files for *The Hound of the Baskervilles* (*hound.txt*), *The War of the Worlds* (*war.txt*), and *The Lost World* (*lost.txt*) along with the book's code from *https://nostarch.com/XXX*.

These came from Project Gutenberg (*http://www.gutenberg.org*), a great source for public domain literature. So that you can use these texts right away, I've stripped them of extraneous material such as table of contents, chapter titles, copyright information, and so on.

The Stylometry Code

The *stylometry.py* program you'll write next loads the text files as strings, tokenizes them into words, and then runs the five stylometric analyses listed on page XX. The program will output a combination of plots and shell messages that will help you determine who wrote *The Lost World*.

Keep the program in the same folder as the three text files. If you don't want to enter the code yourself, just follow along with the downloadable code available at *https://nostarch.com/XXX*.

Importing Modules and Defining the main() Function

Listing 2-1 imports NLTK and `matplotlib`, assigns a constant, and defines the `main()` function to run the program. The functions used in `main()` will be described in detail later in the chapter.

stylometry.py, part 1

```
import nltk
from nltk.corpus import stopwords
import matplotlib.pyplot as plt

LINES = ['-', ':', '--']  # Line style for plots.

def main():
  ❶ strings_by_author = dict()
    strings_by_author['doyle'] = text_to_string('hound.txt')
    strings_by_author['wells'] = text_to_string('war.txt')
    strings_by_author['unknown'] = text_to_string('lost.txt')

    print(strings_by_author['doyle'][:300])

  ❷ words_by_author = make_word_dict(strings_by_author)
    len_shortest_corpus = find_shortest_corpus(words_by_author)
  ❸ word_length_test(words_by_author, len_shortest_corpus)
```

```
    stopwords_test(words_by_author, len_shortest_corpus)
    parts_of_speech_test(words_by_author, len_shortest_corpus)
    vocab_test(words_by_author)
    jaccard_test(words_by_author, len_shortest_corpus)
```

Listing 2-1: Imports modules and defines the `main()` function

Start by importing NLTK and the Stopwords Corpus. Then import `matplotlib`.

Create a variable called `LINES` and use the all-caps convention to indicate it should be treated as a constant. By default, `matplotlib` plots in color, but you'll still want to designate a list of symbols for color-blind people and this black-and-white book!

Define `main()` at the start of the program. The steps in this function are almost as readable as pseudocode and provide a good overview of what the program will do. The first step will be to initialize a dictionary to hold the text for each author ❶. The `text_to_string()` function will load each corpus into this dictionary as a string. The name of each author will be the dictionary key (using `unknown` for *The Lost World*), and the string of text from their novel will be the value. For example, here's the key, `Doyle`, with the value text string greatly truncated:

```
{'Doyle': 'Mr. Sherlock Holmes, who was usually very late in the mornings -snip-'}
```

Immediately after populating the dictionary, print the first 300 items for the `doyle` key to ensure things went as planned. This should produce the following printout:

```
Mr. Sherlock Holmes, who was usually very late in the mornings, save
upon those not infrequent occasions when he was up all night, was seated
at the breakfast table. I stood upon the hearth-rug and picked up the
stick which our visitor had left behind him the night before. It was a
fine, thick piec
```

With the corpora loaded correctly, the next step is to tokenize the strings into words. Currently, Python doesn't recognize words but instead works on *characters*, such as letters, numbers, and punctuation marks. To remedy this, you'll use the `make_word_dict()` function to take the `strings_by_author` dictionary as an argument, split out the words in the strings, and return a dictionary called `words_by_author` with the authors as keys and a list of words as values ❷.

Stylometry relies a lot on word counts, so it works best when each corpus is the same length. There are multiple ways to ensure apples-to-apples comparisons. With *chunking*, you divide the text into blocks of, say, 5,000 words, and compare them. You can also normalize by using relative frequencies, rather than direct counts, or by truncating to the shortest corpus.

Let's explore the truncation option. Pass the words dictionary to another function, `find_shortest_corpus()`, that calculates the number of words in each author's list and returns the length of the shortest corpus. Table 2-1 shows the length of each corpus.

Table 2-1: Length (Word Count) of Each Corpus

Corpus	Length
Hound (Doyle)	58,387
War (Wells)	59,469
World (Unknown)	74,961

Since the shortest corpus here represents a robust dataset of almost 60,000 words, you'll use the `len_shortest_corpus` variable to truncate the other two corpora to this length, prior to doing any analysis. The assumption, of course, is that the backend content of the truncated texts is not significantly different from that in the front.

The next five lines call functions that perform the stylometric analysis, as listed in the "Strategy" section on page XX ❸. All the functions take the `words_by_author` dictionary as an argument, and most take `len_shortest_corpus`, as well. We'll look at these functions as soon as we finish preparing the texts for analysis.

Loading Text and Building a Word Dictionary

Listing 2-2 defines two functions. The first reads in a text file as a string. The second builds a dictionary with each author's name as the key and his novel, now tokenized into individual words rather than a continuous string, as the value.

stylometry.py, part 2

```
def text_to_string(filename):
    """Read a text file and return a string."""
    with open(filename) as infile:
        return infile.read()

❶ def make_word_dict(strings_by_author):
    """Return dictionary of tokenized words by corpus by author."""
    words_by_author = dict()
    for author in strings_by_author:
        tokens = nltk.word_tokenize(strings_by_author[author])
      ❷ words_by_author[author] = ([token.lower() for token in tokens
                                     if token.isalpha()])
    return words_by_author
```

Listing 2-2: Defines the `text_to_string()` and `make_word_dict()` functions

First, define the `text_to_string()` function to load a text file. The built-in `read()` function reads the whole file as an individual string, allowing relatively easy file-wide manipulations. Use `with` to open the file so that it will be closed automatically regardless of how the block terminates. Just like putting away your toys, closing files is good practice. It prevents bad things from happening, like running out of file descriptors, locking files from further access, corrupting files, or losing data if writing to files.

Some users may encounter a `UnicodeDecodeError` like the following one when loading the text:

```
UnicodeDecodeError: 'ascii' codec can't decode byte 0x93 in position 365:
ordinal not in range(128)
```

Encoding and *decoding* refer to the process of converting from characters stored as bytes to human-readable strings. The problem is that the default encoding for the built-in function `open()` is platform-dependent and depends on the value of `locale.getpreferredencoding()`. For example, you'll get the following encoding if you run this on Windows 10:

```
>>> import locale
>>> locale.getpreferredencoding()
'cp1252'
```

CP-1252 is a legacy Windows character encoding. If you run the same code on a Mac, it may return something different, like `'US-ASCII'` or `'UTF-8'`.

UTF stands for *Unicode Transformational Format*, which is a text character format designed for backward compatibility with ASCII. Although UTF-8 can handle all character sets—and is the dominant form of encoding used on the World Wide Web—it's not the default option for many text editors.

Additionally, Python 2 assumed all text files were encoded with `latin-1`, used for the Latin alphabet. Python 3 is more sophisticated and tries to detect encoding problems as early as possible. It may throw an error, however, if the encoding isn't specified.

So, the first troubleshooting step should be to pass `open()` the `encoding` argument and specify UTF-8.

```
    with open(filename, encoding='utf-8') as infile:
```

If you still have problems loading the corpora files, try adding an `errors` argument as follows:

```
    with open(filename, encoding='utf-8', errors='ignore') as infile:
```

You can ignore errors because these text files were downloaded as UTF-8 and have already been tested using this approach. For more on UTF-8, see *https://docs.python.org/3/howto/unicode.html*.

Next, define the `make_word_dict()` function that will take the dictionary of strings by author and return a dictionary of words by author ❶. First, initialize an empty dictionary named `words_by_author`. Then, loop through the keys in the `strings_by_author` dictionary. Use NLTK's `word_tokenize()` method and pass it the string dictionary's key. The result will be a list of tokens that will serve as the dictionary value for each author. Tokens are just chopped up pieces of a corpus, typically sentences or words.

The following snippet demonstrates how the process turns a continuous string into a list of tokens (words and punctuation):

```
>>> import nltk
>>> str1 = 'The rain in Spain falls mainly on the plain.'
>>> tokens = nltk.word_tokenize(str1)
>>> print(type(tokens))
<class 'list'>
>>> tokens
['The', 'rain', 'in', 'Spain', 'falls', 'mainly', 'on', 'the', 'plain', '.']
```

This is similar to using Python's built-in `split()` function, but `split()` doesn't achieve tokens from a linguistic standpoint (note that the period is not tokenized).

```
>>> my_tokens = str1.split()
>>> my_tokens
['The', 'rain', 'in', 'Spain', 'falls', 'mainly', 'on', 'the', 'plain.']
```

Once you have the tokens, populate the `words_by_author` dictionary using list comprehension ❷. *List comprehension* is a shorthand way to execute loops in Python. You need to surround the code with square brackets to indicate a list. Convert the tokens to lowercase and use the built-in `isalpha()` method, which returns `True` if all the characters in a token are part of the alphabet, and `False` otherwise. This will filter out numbers and punctuation. It will also filter out hyphenated words or names. Finish by returning the `words_by_author` dictionary.

Finding the Shortest Corpus

In computational linguistics, *frequency* refers to the number of occurrences in a corpus. Thus, frequency means the *count*, and methods you'll use later return a dictionary of words and their counts. To compare counts in a meaningful way, the corpora should all have the same number of words.

Because the three corpora used here are large (see Table 2-1), you can safely normalize the corpora by truncating them all to the length of the shortest. Listing 2-3 defines a function that finds the shortest corpus in the `words_by_author` dictionary and returns its length.

stylometry.py, part 3

```
def find_shortest_corpus(words_by_author):
    """Return length of shortest corpus."""
    word_count = []
    for author in words_by_author:
        word_count.append(len(words_by_author[author]))
        print('\nNumber of words for {} = {}\n'.
              format(author, len(words_by_author[author])))
    len_shortest_corpus = min(word_count)
    print('length shortest corpus = {}\n'.format(len_shortest_corpus))
    return len_shortest_corpus
```

Listing 2-3: Listing 2-3: Defines the `find_shortest_corpus()`function

Define the function that takes the `words_by_author` dictionary as an argument. Immediately start an empty list to hold a word count.

Loop through the authors (keys) in the dictionary. Get the length of the value for each key, which is a list object, and append the length to the `word_count` list. The length here represents the number of words in the corpus. For each pass through the loop, print the author's name and the length of his tokenized corpus.

When the loop ends, use the built-in `min()` function to get the lowest count, and assign it to the `len_shortest_corpus` variable. Print the answer and then return the variable.

Comparing Word Lengths

Part of an author's distinctive style is the words they use. Faulkner observed that Hemingway never sent a reader running to the dictionary; Hemingway accused Faulkner of using "10-dollar words." This behavior is expressed in the length of words and in vocabulary, which we'll look at later in the chapter.

Listing 2-4 defines a function to compare the length of words per corpus and plot the results as a frequency distribution. In a frequency distribution, the lengths of words are plotted against the number of counts for each length. For words that are six letters long, for example, one author may have a count of 4,000, and another may have a count of 5,500. A frequency distribution allows comparison across a range of word lengths, rather than just at the average word length.

The function in Listing 2-4 uses list slicing to truncate the word lists to the length of the shortest corpus so the results aren't skewed by the size of the novel.

stylometry.py, part 4

```
def word_length_test(words_by_author, len_shortest_corpus):
    """Plot word length freq by author, truncated to shortest corpus
length."""
    by_author_length_freq_dist = dict()
    plt.figure(1)
    plt.ion()

❶ for i, author in enumerate(words_by_author):
        word_lengths = [len(word) for word in words_by_author[author]
                        [:len_shortest_corpus]]
        by_author_length_freq_dist[author] = nltk.FreqDist(word_lengths)
      ❷ by_author_length_freq_dist[author].plot(15,
                                                 linestyle=LINES[i],
                                                 label=author,
                                                 title='Word Length')
    plt.legend()
    #plt.show()  # Uncomment to see plot while coding.
```

Listing 2-4: Listing 2-4: Defines the `word_length_test()`function

All the stylometric functions will use the dictionary of tokens; almost all will use the length of the shortest corpus parameter to ensure consistent sample sizes. Use these variable names as the function parameters.

Start an empty dictionary to hold the frequency distribution of word lengths by author and then start making plots. Since you are going to make multiple plots, start by instantiating a figure object named `1`. So that all the plots stay up after creation, turn on the interactive plot mode with `plt.ion()`.

Next, start looping through the authors in the tokenized dictionary ❶. Use the `enumerate()` function to generate an index for each author that you'll use to choose a line style for the plot. For each author, use list comprehension to get the length of each word in the value list, with the range truncated to the length of the shortest corpus. The result will be a list where each word has been replaced by an integer representing its length.

Now, start populating your new by-author dictionary to hold frequency distributions. Use `nltk.FreqDist()`, which takes the list of word lengths and creates a data object of word frequency information that can be plotted.

You can plot the dictionary directly using the class method `plot()`, without the need to reference `pyplot` through `plt` ❷. This will plot the most frequently occurring sample first, followed by the number of samples you specify, in this case, `15`. This means you will see the frequency distribution of words from 1 to 15 letters long. Use `i` to select from the `LINES` list, and finish by providing a label and a title. The label will be used in the legend, called using `plt.legend()`.

Note that you can change how the frequency data plots using the `cumulative` parameter. If you specify `cumulative=True`, you will see a cumulative distribution (Figure 2-3, left). Otherwise, `plot()` will default to `cumulative=False`, and you will see the actual counts, arranged from highest to lowest (Figure 2-3, right). Continue to use the default option for this project.

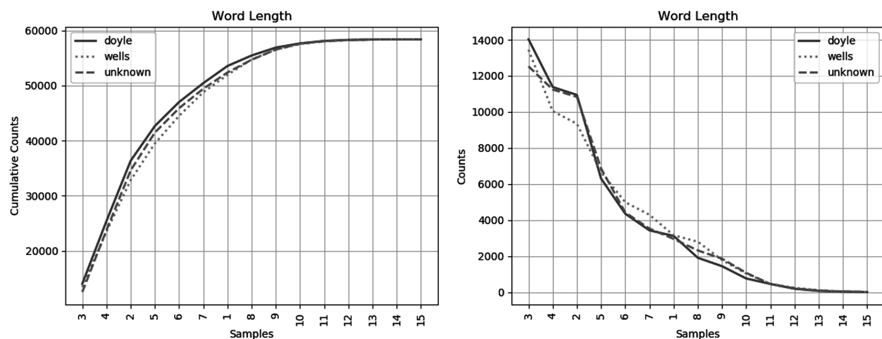


Figure 2-3: The NLTK cumulative plot option, left, versus the default frequency plot option, right

Finish by calling the `plt.show()` method to display the plot, but leave it commented out. If you want to see the plot immediately after coding this function, you can uncomment it. Also note that if you launch this program

via Windows PowerShell, the plots may close immediately unless you use the `block` flag: `plt.show(block=True)`. This will keep the plot up but halt execution of the program until the plot is closed.

Based solely on the word length frequency plot in Figure 2-3, Doyle's style matches the unknown author's more closely, though there are segments where Wells compares the same or better. Now let's run some other tests to see whether we can confirm that finding.

Comparing Stop Words

A *stop word* is a small word used often, like *the*, *by*, and *but*. These words are filtered out for tasks like online searches, because they provide no contextual information, and they were once thought to be of little value in identifying authorship.

But stop words, used frequently and without much thought, are perhaps the best signature for an author's style. And since the texts you're comparing are usually about different subjects, these stop words become important, as they are agnostic to content and common across all texts.

Listing 2-5 defines a function to compare the use of stop words in the three corpora.

stylometry.py, part 5

```
def stopwords_test(words_by_author, len_shortest_corpus):
    """Plot stopwords freq by author, truncated to shortest corpus length."""
    stopwords_by_author_freq_dist = dict()
    plt.figure(2)
    stop_words = set(stopwords.words('english'))  # Use set for speed.
    #print('Number of stopwords = {}\n'.format(len(stop_words)))
    #print('Stopwords = {}\n'.format(stop_words))

    for i, author in enumerate(words_by_author):
        stopwords_by_author = [word for word in words_by_author[author]
                               [:len_shortest_corpus] if word in stop_words]
        stopwords_by_author_freq_dist[author] = nltk.FreqDist(stopwords_by_
author)
        stopwords_by_author_freq_dist[author].plot(50,
                                                   label=author,
                                                   linestyle=LINES[i],
                                                   title=
                                                   '50 Most Common Stopwords')
    plt.legend()
##    plt.show()  # Uncomment to see plot while coding function.
```

Listing 2-5: Defines the `stopwords_test()` *function*

Define a function that takes the words dictionary and the length of the shortest corpus variables as arguments. Then initialize a dictionary to hold the frequency distribution of stop words for each author. You don't want to cram all the plots in the same figure, so start a new figure named `2`.

Assign a local variable, `stop_words`, to the NLTK stop words corpus for English. Sets are quicker to search than lists, so make the corpus a set for faster lookups later. The next two lines, currently commented out, print the number of stop words (179) and the stop words themselves.

Now, start looping through the authors in the `words_by_author` dictionary. Use list comprehension to pull out all the stop words in each author's corpus, and use these as the value in a new dictionary named `stopwords_by_author`. In the next line, you'll pass this dictionary to NLTK's `FreqDist()` method and use the output to populate the `stopwords_by_author_freq_dist` dictionary. This dictionary will contain the data needed to make the frequency distribution plots for each author.

Repeat the code you used to plot the word lengths in Listing 2-4, but set the number of samples to `50` and give it a different title. This will plot the top 50 stop words in use (Figure 2-4).

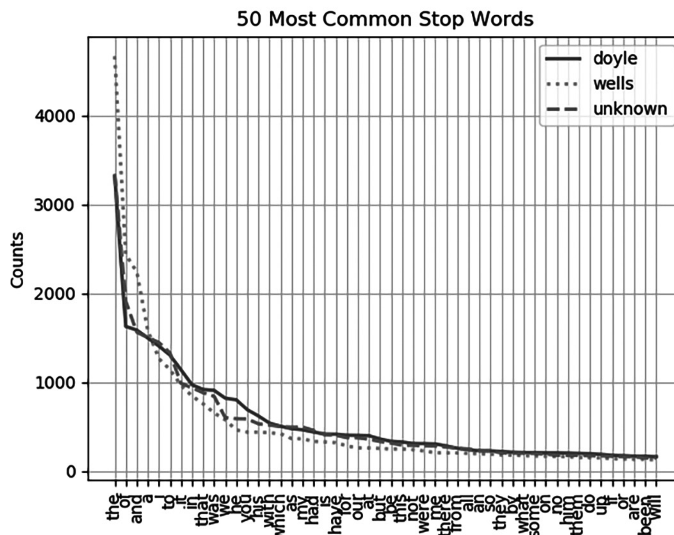


Figure 2-4: Frequency plot of top 50 stop words by author

Both Doyle and the unknown author use stop words in a similar manner. At this point, two analyses have favored Doyle as the most likely author of the unknown text, but there's still more to do.

Comparing Parts of Speech

Now let's compare the parts of speech used in the three corpora. NLTK uses a part-of-speech (POS) tagger, called `PerceptronTagger`, to identify parts of speech. POS taggers process a sequence of tokenized words and attach a POS tag to each word (see Table 2-2).

Table 2-2: Table 2-2: Parts of Speech with Tag Values

Part of Speech	Tag	Part of Speech	Tag
Coordinating conjunction	CC	Possessive pronoun	PRP$
Cardinal number	CD	Adverb	RB
Determiner	DT	Adverb, comparative	RBR
Existential there	EX	Adverb, superlative	RBS
Foreign word	FW	Particle	RP
Preposition or subordinating conjunction	IN	Symbol	SYM
Adjective	JJ	To	TO
Adjective, comparative	JJR	Interjection	UH
Adjective, superlative	JJS	Verb, base form	VB
List item marker	LS	Verb, past tense	VBD
Modal	MD	Verb, gerund or present participle	VBG
Noun, singular or mass	NN	Verb, past participle	VBN
Noun, plural	NNS	Verb, non-third-person singular present	VBP
Noun, proper noun, singular	NNP	Verb, third-person singular present	VBZ
Noun, proper noun, plural	NNPS	Wh-determiner, which	WDT
Predeterminer	PDT	Wh-pronoun, who, what	WP
Possessive ending	POS	Possessive wh-pronoun, whose	WP$
Personal pronoun	PRP	Wh-adverb, where, when	WRB

The taggers are typically trained on large datasets like the *Penn Treebank* or *Brown Corpus*, making them highly accurate though not perfect. You can also find training data and taggers for languages other than English. You don't need to worry about all these various terms and their abbreviations. As with the previous tests, you'll just need to compare lines in a chart.

Listing 2-6 defines a function to plot the frequency distribution of POS in the three corpora.

stylometry.py, part 6

```
def parts_of_speech_test(words_by_author, len_shortest_corpus):
    """Plot author use of parts-of-speech such as nouns, verbs, adverbs,etc."""
    by_author_pos_freq_dist = dict()
    plt.figure(3)
    for i, author in enumerate(words_by_author):
        pos_by_author = [pos[1] for pos in nltk.pos_tag(words_by_author[author]
                                                        [:len_shortest_
corpus])]
        by_author_pos_freq_dist[author] = nltk.FreqDist(pos_by_author)
        by_author_pos_freq_dist[author].plot(35,
                                             label=author,
                                             linestyle=LINES[i],
                                             title='Part of Speech')
```

```
    plt.legend()
    plt.show()
```

Listing 2-6: Defines the `parts_of_speech_test()` *function*

Define a function that takes as arguments—you guessed it—the words dictionary and the length of the shortest corpus. Then initialize a dictionary to hold the frequency distribution for the POS for each author, followed by a function call for a third figure.

Start looping through the authors in the `words_by_author` dictionary and use list comprehension and the NLTK `pos_tag()` method to build a list called `pos_by_author`. For each author, this creates a list with each word in the author's corpus replaced by its corresponding POS tag, as shown here:

```
['NN', 'NNS', 'WP', 'VBD', 'RB', 'RB', 'RB', 'IN', 'DT', 'NNS', -snip-]
```

Next, make a frequency distribution of the POS list, and with each loop plot the curve, using the top 35 samples. Note that there are only 36 POS tags, and several, such as *list item markers*, rarely appear in novels.

This is the final plot you'll make, so call `plt.show()`to draw all the plots to the screen. As pointed out in Listing 2-4, if you're using Windows PowerShell to launch the program, you may need to use `plt.show(block=True)` to keep the plots from closing automatically.

The previous plots, along with the current one (Figure 2-5), should appear after about 10 seconds.

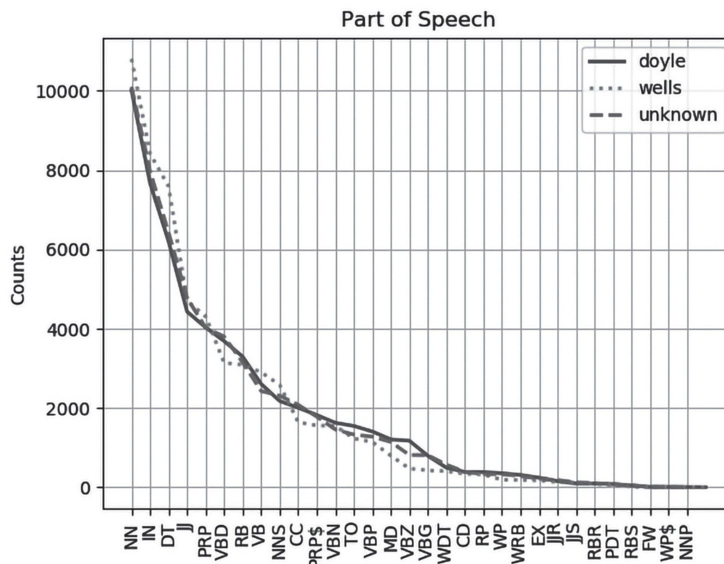


Figure 2-5: Frequency plot of top 35 parts of speech by author

Once again, the match between the Doyle and unknown curves is clearly better than the match of unknown to Wells. This suggests that Doyle is the author of the unknown corpus.

Comparing Author Vocabularies

To compare the vocabularies among the three corpora, you'll use the *chi-squared random variable* (X^2), also known as the *test statistic*, to measure the "distance" between the vocabularies employed in the unknown corpus and each of the known corpora. The closest vocabularies will be the most similar. The formula is

$$X^2 = \sum_{i=1}^{n} \frac{(Oi - Ei)^2}{Ei}$$

where O is the observed word count, and E is the expected word count assuming the corpora being compared are both by the same author.

This means that, if you assume Doyle wrote both novels, they should both have the same—or a similar—proportion of the most common words. The test statistic lets you quantify this by measuring how much the counts for each word differ. The lower the chi-squared test statistic, the greater the similarity between two distributions.

Listing 2-7 defines a function to compare vocabularies among the three corpora.

stylometry.py, part 7

```
def vocab_test(words_by_author):
    """Compare author vocabularies using the Chi Squared statistical test."""
    chisquared_by_author = dict()
    for author in words_by_author:
      ❶ if author != 'unknown':
            combined_corpus = (words_by_author[author] +
                               words_by_author['unknown'])
            author_proportion = (len(words_by_author[author])/
                                 len(combined_corpus))
            combined_freq_dist = nltk.FreqDist(combined_corpus)
            most_common_words = list(combined_freq_dist.most_common(1000))
            chisquared = 0
          ❷ for word, combined_count in most_common_words:
                observed_count_author = words_by_author[author].count(word)
                expected_count_author = combined_count * author_proportion
                chisquared += ((observed_count_author -
                                expected_count_author)**2 /
                               expected_count_author)
              ❸ chisquared_by_author[author] = chisquared
            print('Chi-squared for {} = {:.1f}'.format(author, chisquared))
    most_likely_author = min(chisquared_by_author, key=chisquared_by_author.
get)
    print('Most-likely author by vocabulary is {}\n'.format(most_likely_
author))
```

Listing 2-7: Defines the `vocab_test()` function

The `vocab_test()` function needs the word dictionary but not the length of the shortest corpus. Like the previous functions, however, it starts by creating a new dictionary to hold the chi-squared value per author and then loops through the word dictionary.

To calculate chi-squared, you'll need to join each author's corpus with the unknown corpus. You don't want to combine `unknown` with itself, so use a conditional to avoid this ❶. For the current loop, combine the author's corpus with unknown and then get the current author's proportion by dividing the length of his corpus by the length of the combined corpus. Then get the frequency distribution of the combined corpus by calling `nltk.FreqDist()`.

Now, make a list of the 1,000 most common words in the combined text by using the `most_common()` method and passing it `1000`. There is no hard, fast rule for how many words you should consider in a stylometric analysis. Suggestions in the literature call for the most common 100 to 1,000 words. Since you are working with large texts, err to the larger value.

Initialize the `chisquared` variable with `0`; then start a nested `for` loop that works through the `most_common_words` list ❷. The `most_common()`method returns a list of tuples, with each tuple containing the word and its count.

```
[('the', 7778), ('of', 4112), ('and', 3713), ('i', 3203), ('a', 3195), -snip-]
```

Next, you get the observed count per author from the word dictionary. For Doyle this would be the count of the most common words in the corpus of *The Hound of the Baskervilles*. Then, you get the expected count, which for Doyle would be the count you would expect if he wrote both *The Hound of the Baskervilles* and the unknown corpus. To do this, multiply the number of counts in the combined corpus by the previously calculated author's proportion. Then apply the formula for chi-squared and add the result to the dictionary that tracks each author's chi-squared score ❸. Display the result for each author.

To find the author with the lowest chi-squared score, call the built-in `min()` function and pass it the dictionary and dictionary key, which you obtain with the `get()` method. This will yield the *key* corresponding to the minimum *value*. This is important. If you omit this last argument, `min()` will return the minimum *key* based on the alphabetical order of the names, *not* their chi-squared score! You can see this mistake in the following snippet:

```
>>> print(mydict)
{'doyle': 100, 'wells': 5}
>>> minimum = min(mydict)
>>> print(minimum)
'doyle'
>>> minimum = min(mydict, key=mydict.get)
>>> print(minimum)
'wells'
```

It's easy to assume that the `min()` function returns the minimum numerical *value*, but as you saw, it looks at dictionary *keys* by default.

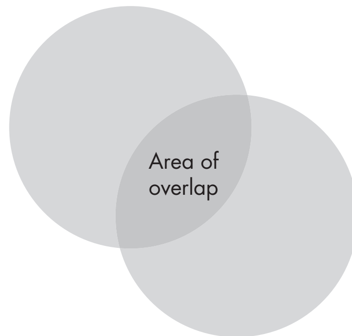
Complete the function by printing the most likely author based on the chi-squared score.

```
Chi-squared for doyle = 4744.4
Chi-squared for wells = 6856.3
Most-likely author by vocabulary is doyle
```

Yet another test suggests that Doyle is the author!

Calculating Jaccard Similarity

To determine the degree of similarity among sets created from the corpora, you'll use the *Jaccard similarity coefficient*. Also called the *intersection over union*, this is simply the area of overlap between two sets divided by the area of union of the two sets (Figure 2-6).



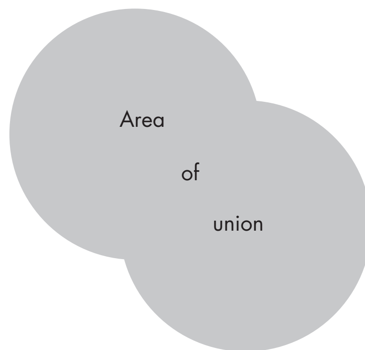


Figure 2-6: Intersection-over-union for a set is the area of overlap divided by the area of union.

The more overlap there is between sets created from two texts, the more likely they were written by the same author. Listing 2-8 defines a function for gauging the similarity of sample sets.

stylometry.py, part 8

```
def jaccard_test(words_by_author, len_shortest_corpus):
    """Calculate Jaccard similarity of each known corpus to unknown corpus."""
    jaccard_by_author = dict()
    unique_words_unknown = set(words_by_author['unknown']
                               [:len_shortest_corpus])
```

```
❶ authors = (author for author in words_by_author if author != 'unknown')
   for author in authors:
       unique_words_author = set(words_by_author[author][:len_shortest_
corpus])
       shared_words = unique_words_author.intersection(unique_words_unknown)
     ❷ jaccard_sim = (float(len(shared_words))/ (len(unique_words_author) +
                                                 len(unique_words_unknown) -
                                                 len(shared_words)))
       jaccard_by_author[author] = jaccard_sim
       print('Jaccard Similarity for {} = {}'.format(author, jaccard_sim))
 ❸ most_likely_author = max(jaccard_by_author, key=jaccard_by_author.get)
   print('Most-likely author by similarity is {}'.format(most_likely_author))

if __name__ == '__main__':
    main()
```

Listing 2-8: Listing 2-8: Defines the `jaccard_test()` function

Like most of the previous tests, the `jaccard_test()` function takes the word dictionary and length of the shortest corpus as arguments. You'll also need a dictionary to hold the Jaccard coefficient for each author.

Jaccard similarity works with unique words, so you'll need to turn the corpora into sets to remove duplicates. First, you'll build a set from the `unknown` corpus. Then you'll loop through the known corpora, turning them into sets and comparing them to the unknown set. Be sure to truncate all the corpora to the length of the shortest corpus when making the sets.

Prior to running the loop, use a generator expression to get the names of the authors, other than `unknown`, from the `words_by_author` dictionary ❶. A *generator expression* is a function that returns an object that you can iterate over one value at a time. It looks a lot like list comprehension, but instead of square brackets, it's surrounded by parentheses. And instead of constructing a potentially memory-intensive list of items, the generator yields them in real time. Generators are useful when you have a large set of values that you need to use only once. I use one here as an opportunity to demonstrate the process.

When you assign a generator expression to a variable, all you get is a type of iterator called a *generator object*. Compare this to making a list, as shown here:

```
>>> mylist = [i for i in range(4)]
>>> mylist
[0, 1, 2, 3]
>>> mygen = (i for i in range(4))
>>> mygen
<generator object <genexpr> at 0x000002717F547390>
```

The generator expression in the previous snippet is the same as this generator function:

```
def generator(my_range):
    for i in range(my_range):
        yield i
```

Whereas the `return` statement ends a function, the `yield` statement *suspends* the function's execution and sends a value back to the caller. Later, the function can resume where it left off. When a generator reaches its end, it's "empty" and can't be called again.

Back to the code, start a `for` loop using the `authors` generator. Find the unique words for each known author, just as you did for `unknown`. Then use the built-in `intersection()` function to find all the words shared between the current author's set of words and the set for `unknown`. The *intersection* of two given sets is the largest set that contains all the elements that are common to both. With this information, you can calculate the Jaccard similarity coefficient ❹.

Update the `jaccard_by_author` dictionary and print each outcome in the interpreter window. Then find the author with the maximum Jaccard value ❺ and print the results.

```
Jaccard Similarity for doyle = 0.34847801578354004
Jaccard Similarity for wells = 0.30786921307869214
Most-likely author by similarity is doyle
```

The outcome should favor Doyle.

Finish *stylometry.py* with the code to run the program as an imported module or in stand-alone mode.

Summary

The true author of *The Lost World* is Doyle, so we'll stop here and declare victory. If you want to explore further, a next step might be to add more known texts to `doyle` and `wells` so that their combined length is closer to that for *The Lost World* and you don't have to truncate it. You could also test for sentence length and punctuation style or employ more sophisticated techniques like neural nets and genetic algorithms.

You can also refine existing functions, like `vocab_test()` and `jaccard_test()`, with *stemming* and *lemmatization* techniques that reduce words to their root forms for better comparisons. As the program is currently written, *talk*, *talking*, and *talked* are all considered completely different words even though they share the same root.

At the end of the day, stylometry can't prove with absolute certainty that Sir Author Conan Doyle wrote *The Lost World*. It can only suggest, through weight of evidence, that he is the more likely author than Wells. Framing the question very specifically is important, since you can't evaluate all possible authors. For this reason, successful authorship attribution begins with good old-fashioned detective work that trims the list of candidates to a manageable length.

Further Reading

Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit (O'Reilly, 2009), by Steven Bird, Ewan Klein, and Edward Loper, is an accessible introduction to NLP using Python, with lots of exercises and useful integration with the NLTK website. A new version of the book, updated for Python 3 and NLTK 3, is available online at *http://www.nltk.org/book/*.

In 1995, novelist Kurt Vonnegut proposed the idea that "stories have shapes that can be drawn on graph paper" and suggested "feeding them into computers." In 2018, researchers followed up on this idea using more than 1,700 English novels. They applied an NLP technique called *sentiment analysis* that finds the emotional tone behind words. An interesting summary of their results, "Every Story in the World Has One of These Six Basic Plots," can be found on the BBC.com website: *http://www.bbc.com/culture/story/20180525-every-story-in-the-world-has-one-of-these-six-basic-plots*.

Practice Project: Hunting the Hound with Dispersion

NLTK comes with a fun little feature, called a *dispersion plot*, that lets you post the location of a word in a text. More specifically, it plots the occurrences of a word versus how many words from the beginning of the corpus that it appears.

Figure 2-7 is a dispersion plot for major characters in *The Hound of the Baskervilles*.

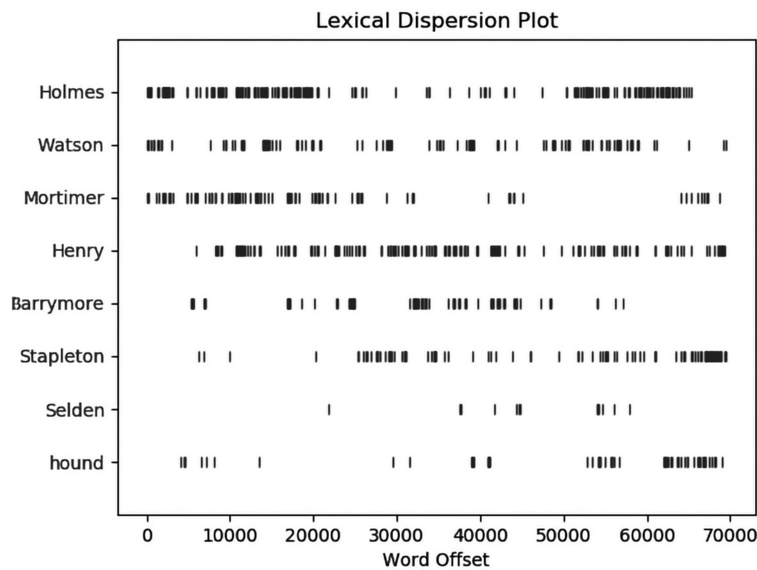


Figure 2-7: Dispersion plot for major characters in The Hound of the Baskervilles

If you're familiar with the story—and I won't spoil it if you're not—then you'll appreciate the sparse occurrence of Holmes in the middle, the almost bimodal distribution of Mortimer, and the late story overlap of Barrymore, Selden, and the hound.

Dispersion plots can have more practical applications. For example, as the author of technical books, I need to define a new term when it first appears. This sounds easy, but sometimes the editing process can shuffle whole chapters, and issues like this can fall through the cracks. A dispersion plot, built with a long list of technical terms, can make finding these first occurrences a lot easier.

For another use case, imagine you're a data scientist working with paralegals on a criminal case involving insider trading. To find out whether the accused talked to a certain board member just prior to making the illegal trades, you can load the subpoenaed emails of the accused as a continuous string and generate a dispersion plot. If the board member's name appears as expected, case closed!

For this practice project, write a Python program that reproduces the dispersion plot shown in Figure 2-7. If you have problems loading the *hound.txt* corpus, revisit the Listing 2-2 Unicode discussion on page 34. You can find a solution, *hound_dispersion.py*, in the appendix and online.

Practice Project: Punctuation Heatmap

A *heatmap* is a diagram that uses colors to represent data values. Heatmaps have been used to visualize the punctuation habits of famous authors (*https://www.fastcompany.com/3057101/the-surprising-punctuation-habits-of-famous-authors-visualized*) and may prove helpful in attributing authorship for *The Lost World*.

Write a Python program that tokenizes the three novels used in this chapter based solely on punctuation. Then focus on the use of semicolons. For each author, plot a heatmap that displays semicolons as blue and all other marks as yellow or red. Figure 2-8 shows example heatmaps for Wells' *The War of the Worlds* and Doyle's *The Hound of the Baskervilles*.

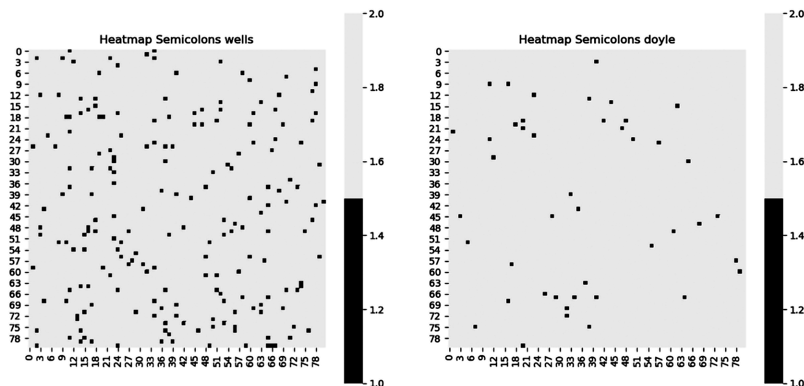


Figure 2-8: Heatmap of semicolon use (dark squares) for Wells (left) and Doyle (right)

Compare the three heatmaps. Do the results favor Doyle or Wells as the author for *The Lost World*?

You can find a solution, *heatmap_semicolon.py*, in the appendix and online.

Challenge Project: Fixing Frequency

As noted previously, frequency in NLP refers to counts, but it can also be expressed as the number of occurrences per unit time. Alternatively, it can be expressed as a ratio or percent.

Define a new version of the `nltk.FreqDist()` method that uses percent, rather than counts, and use it to make the charts in the *stylometry.py* program. For help, see the Clearly Erroneous blog (*https://martinapugliese.github.io/plotting-the-actual-frequencies-in-a-FreqDist-in-nltk/*).