Simple Statistics with NLTK: Counting of POS Tags and Frequency Distributions

h2kinfosys.com/blog/simple-statistics-with-nltk-counting-of-pos-tags-and-frequency-distributions

September 30, 2020

Artificial Intelligence Tutorials

In the last tutorial, we discussed how to assign POS tags to words in a sentence using the pos_tag method of NLTK. We said that POS tagging is a fundamental step in the preprocessing of textual data and is especially needed when building text classification models. We went further to discuss Hidden Markov Models (HMMs) and their importance in text analysis. When creating HMMs, we mentioned that you must count the number of each POS tag in the sentence, to determine the transition probabilities and emission probabilities. While the counting process may be a non-issue for a small text, it can be daunting for large datasets.

In such cases, we may need to rely on automatic methods to count tags and words. In this tutorial, we will be discussing the ways to count using python's native Counter function and the FreqDist function of NLTK. Consequently, you will also learn about collocations, bigrams, and trigrams. Let's begin!

Counting the Number of Items in a String/List

Python's collections module has a plethora of functions including the Counter class, ChainMap class, OrderedDict class, and so on. Each of these classes has its own specific capabilities. Here, we will focus on the Counter function, which is used to count the number of items in a list, string, or tuple. It returns a dictionary where the key is the element/item in the list and value is the frequency of that element/item in the list.

Let see this simple example below. Say we want to count the number of times each letter appears in a sentence, the Counter class will come in handy. We start by importing the class from the collections module.

```
#import the Counter class
from collections import Counter
#deifne some text
text = "It is necessary for any Data Scientist to understand Natural Language
Processing"
#convert al letters to lower case
text = text.lower()
#instantiate the Counter classes on the text
the_count = Counter(text)
#print the count
print(the_count)
```

Output:

```
Counter({' ': 11, 'a': 9, 's': 8, 't': 7, 'n': 7, 'i': 6, 'e': 6, 'r': 5, 'c': 3,
'o': 3, 'd': 3, 'u': 3, 'g': 3, 'y': 2, 'l': 2, 'f': 1, 'p': 1})
```

As seen, we had 11 whitespaces, 9 a's, 8 s', 7 t's, 7 n's, and so on.

We can use this same methodology to count the POS tags in a sentence. Take a look.

```
#import nltk library
import nltk
#import the Counter class
from collections import Counter
#define some text
text = "It is necessary for any Data Scientist to understand Natural Language
Processing"
#convert all letters to lower case
text = text.lower()
#tokenize the words in the text
tokens = nltk.word_tokenize(text)
#assign POS tags to each words
pos = nltk.pos_tag(tokens)
#Count the POS tags
the_count = Counter(tag for _, tag in pos)
#print the count
print(the_count)
```

Output:

```
Counter({'NN': 3, 'JJ': 2, 'PRP': 1, 'VBZ': 1, 'IN': 1, 'DT': 1, 'NNS': 1, 'TO': 1,
'VB': 1})
```

Let's do a quick rundown of what each line of code done.

We started off by importing the necessary libraries, after which we defined the text we want to tokenize. It was necessary to convert all the words to lower cases so the compiler does not view two same words as different because of uppercase-lowercase variation. Having done that, we tokenize the words and assign POS tags to each word. Bare in mind that the output of the pos_tag() method is a dictionary with keys and value pairs. The key is the individual word in the text, while the value is the corresponding POS tags.

Here's what happens in the for loop. Recall that the pos_tag returns the words and their POS. We wish to count only the POS tags which are the value of the pos_tag() outputted dictionary. We iterate over each POS tag and count the POS tags with the Counter class.

Let's go on to see how to count using NLTK's FreqDict class.

Frequency Distribution with NLTK

Have you imagined how words that provide key information about a topic or book are found? An easy way to go about this is by finding the words that appear the most in the text/book (excluding stopwords). The counting of the number of times a word appears in a document is called Frequency Distribution. In other words, frequency distribution shows how the words are distributed in a document.

You could as well say that the frequency distribution is the term used to count the occurrence of a specific outcome in an experiment. The FreqDist class is used to count the number of times each word token appears in the text. Throughout this tutorial, the textual data will a book from the NLTK corpus, called *Moby Dick*. This is a NLTK's pre-installed corpora.

```
#import the necessary library
from nltk.corpus import gutenberg
#call the book we intend to use and save as text
text = gutenberg.words('melville-moby_dick.txt')
#check number of words in the book
print(len(text))
```

Output: 260819

We can see that it is quite a large book with over 260,000 words. Let's have a peep into what the book looks like.

```
#prints the first 100 words in the book
print(text[:100])
```

Output:

```
['[', 'Moby', 'Dick', 'by', 'Herman', 'Melville', '1851', ']',
'ETYMOLOGY', '.', '(', 'Supplied', 'by', 'a', 'Late', 'Consumptive',
'Usher', 'to', 'a', 'Grammar', 'School', ')', 'The', 'pale', 'Usher', '--
', 'threadbare', 'in', 'coat', ',', 'heart', ',', 'body', ',', 'and',
'brain', ';', 'I', 'see', 'him', 'now', '.', 'He', 'was', 'ever',
'dusting', 'his', 'old', 'lexicons', 'and', 'grammars', ',', 'with', 'a',
'queer', 'handkerchief', ',', 'mockingly', 'embellished', 'with', 'all',
'the', 'gay', 'flags', 'of', 'all', 'the', 'known', 'nations', 'of',
'the', 'world', '.', 'He', 'loved', 'to', 'dust', 'his', 'old',
'grammars', ';', 'it', 'somehow', 'mildly', 'reminded', 'him', 'of',
'his', 'mortality', '.', '"', 'While', 'you', 'take', 'in', 'hand', 'to',
'school', 'others', ',']
```

```
#invoke the FreqDIst class and pass the text as parameter
fdistribution = nltk.FreqDist(text)
```

In the example above, the FreqDist class is instantiated and the text was passed as a parameter. This was saved in the 'fdistribution' variable. Observe that the text has been split into tokenized sentences and tokenized words. If we were dealing with raw text, we must first

tokenize the text using the sent_tokenize() method, then sentence the using the word_tokenize() method. This is because the FreqDist() takes the word tokens as parameters.

To perform the myriad operations on the fdistribution variable, we use the methods FreqDist class holds. There are various methods that can be used to perform different operations on the instantiated class. Let's see some of them.

1. The keys methods. This method is called using the keys() statement. It returns a list of words in the vocabulary in ascending order. If you want to print the first 50 elements, they can be accessed by slicing the list. Check the example below.

```
#return the words in the frequency distribution dictionary
vocabulary = fdistribution.keys()
#prints the first 50 keys of the dictionary
print(vocabulary[:50])
```

Output:

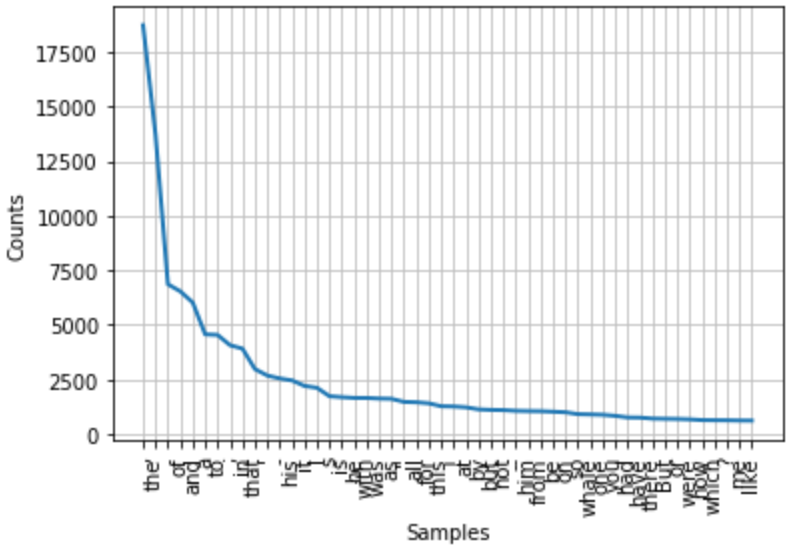
```
['[', 'Moby', 'Dick', 'by', 'Herman', 'Melville', '1851', ']',
'ETYMOLOGY', '.', '(', 'Supplied', 'a', 'Late', 'Consumptive', 'Usher',
'to', 'Grammar', 'School', ')', 'The', 'pale', '--', 'threadbare', 'in',
'coat', ',', 'heart', 'body', 'and', 'brain', ':', 'I', 'see', 'him',
'now', 'He', 'was', 'ever', 'dusting', 'his', 'old', 'lexicons',
'grammars', 'with', 'queer', 'handkerchief', 'mockingly', 'embellished',
'all']
```

1. Plot method. This method is called using the plot() statement. It plots the frequency curve of the words in the vocabulary. The plot statement is a common and important method of the FreqDist class as the pictorial representation gives a solid understanding of how the words are spread in the vocabulary. Check the example below.

#plot the frequency distribution curve for the first 50 words

fdistribution.plot(50)

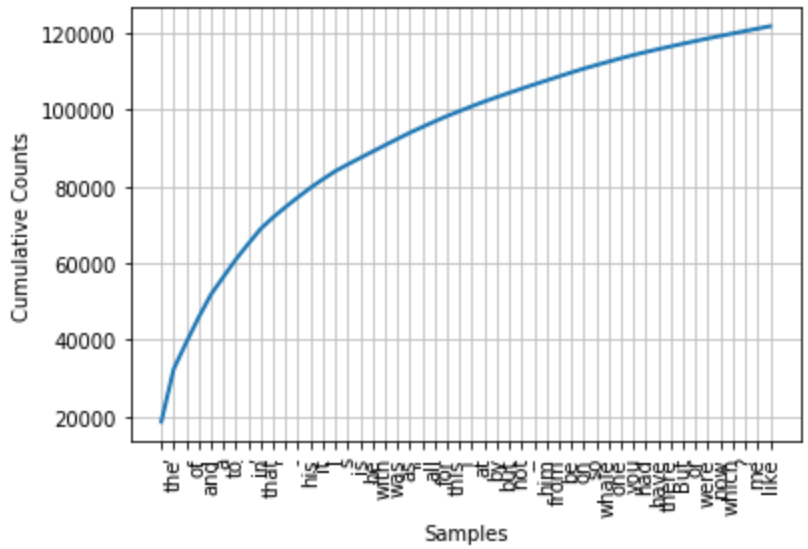
Output:



If we decide to plot the cumulative distribution curve, it is as easy as passing the cumulative argument to be 'True;.

```
#plot the frequency distribution curve for the first 50 words
fdistribution.plot(50, cumulative=True)
```

Output:



Note that you need to have the matplotlib library installed on your machine for the plots to show.

3. Hapaxes method. Sometimes, we may then decide to check for words that are less frequent or even words that appear just once and remove them. The hapaxes() method returns the list of words that are unique in the vocabulary. These words are typically cetological, expostulations, contraband, lexicographers, and so on.

```
#prints the first 50 unique words
fdistribution.hapaxes()[:50]
```

Output:

```
['Herman', 'Melville', ']', 'ETYMOLOGY', 'Late', 'Consumptive', 'School',
'threadbare', 'lexicons', 'mockingly', 'flags', 'mortality', 'signification',
'HACKLUYT', 'Sw', 'HVAL', 'roundness', 'Dut', 'Ger', 'WALLEN', 'WALW', 'IAN',
'RICHARDSON', 'KETOS', 'GREEK', 'CETUS', 'LATIN', 'WHOEL', 'ANGLO', 'SAXON', 'WAL',
'HWAL', 'SWEDISH', 'ICELANDIC', 'BALEINE', 'BALLENA', 'FEGEE', 'ERROMANGOAN',
'Librarian', 'painstaking', 'burrower', 'grub', 'Vaticans', 'stalls', 'higgledy',
'piggledy', 'gospel', 'promiscuously', 'commentator', 'belongest']
```

The uncommon words may be numerous, making it really difficult to substantiate how critical they are. Let's see the number of unique words in our text.

```
len(fdistribution.hapaxes())
```

Output: 9002

There are over 9000 unique words in this text. Obviously, these words cannot classify the book in any way. What do we do hence?

Making Finer Filtering

What about if we turn our attention to longer words? The most common words in the English Language are short words such as for, of, is, a, an, but, then, etc.

We could adjust the condition for selection such that only words with characters more than 10 are returned. It can be done using the code below.

```
#remove duplicate words
each_word = set(text)
#returns only words longer than 16 letters
lengthy_words = [word for word in each_word if len(word) > 15]
#print the lengthy words
print(lengthy_words)
```

Output:

```
['undiscriminating', 'irresistibleness', 'indiscriminately', 'physiognomically',
'subterraneousness', 'uncompromisedness', 'preternaturalness', 'indispensableness',
'circumnavigations', 'characteristically', 'CIRCUMNAVIGATION', 'cannibalistically',
'Physiognomically', 'superstitiousness', 'supernaturalness', 'uncomfortableness',
'hermaphroditical', 'responsibilities', 'comprehensiveness', 'uninterpenetratingly',
'apprehensiveness', 'simultaneousness', 'circumnavigating', 'circumnavigation']
```

While this looks like a breathrough, sometimes informal texts could contain words like harrayyyyyyyy, yeaaaaaaaaaa, waaaaaaaaaaaaat, etc. These are long words but they certainly do not classify the text.

To overcome this challenge, we can extract both the frequently used words and longer words together. Most of the informal words are unique words and will be filtered out if we use most occurring long words. The code below shows an example.

```
#invoke the FreqDIst class and pass the text as parameter
fdistribution = nltk.FreqDist(text)
#remove duplicate words
each_word = set(text)
#returns only words longer than 10 letters and occurs more than 10 times
words = sorted([word for word in each_word if len(word) > 10 and fdistribution[word]
> 10])
#print the words
print(words)
```

Output:

```
['Nantucketer', 'Nevertheless', 'circumstance', 'circumstances', 'considerable',
'considering', 'continually', 'countenance', 'disappeared', 'encountered',
'exceedingly', 'experienced', 'harpooneers', 'immediately', 'indifferent',
'indispensable', 'involuntarily', 'naturalists', 'nevertheless', 'occasionally',
'peculiarities', 'perpendicular', 'significant', 'simultaneously', 'straightway',
'unaccountable']
```

Let's take this a little further.

In real-life applications, most keywords are not single words. They are rather words in pairs or a combination of three words. These words are called collocations. Using collocations typically makes a robust machine learning model and thus, it is important in our discussion. Let's talk about collocations in more detail.

What are Collocations

As mentioned earlier, collocations are words that mostly appear together in a document. They are a sequence of words that occurs unusually often in a document. It can be calculated by dividing the number of times two or three words appear together by the number of words in the documents.

Examples of collocations include fast food, early riser, UV rays, etc. These words are most likely to follow each other in a document. Moreso, one of the words cannot stand in the gap for the other. In the case of UV rays, while UV is a thing, it would be unclear to use the only UV in a sentence rather than UV rays. Hence, it's a collocation.

Collocations can be classified as bigrams and trigrams.

1. Bigrams are a combination of two words. The NLTK library has a built-in method – bigrams(), that can be used to extract bigrams in a document. See the example below.

```
#prints the bigrams for the first 50 words in the text
print(list(nltk.bigrams(text[:50])))
```

Output:

```
[('[', 'Moby'), ('Moby', 'Dick'), ('Dick', 'by'), ('by', 'Herman'), ('Herman',
'Melville'), ('Melville', '1851'), ('1851', ']'), (']', 'ETYMOLOGY'), ('ETYMOLOGY',
'.'), ('.', '('), ('(', 'Supplied'), ('Supplied', 'by'), ('by', 'a'), ('a', 'Late'),
('Late', 'Consumptive'), ('Consumptive', 'Usher'), ('Usher', 'to'), ('to', 'a'),
('a', 'Grammar'), ('Grammar', 'School'), ('School', ')'), (')', 'The'), ('The',
'pale'), ('pale', 'Usher'), ('Usher', '--'), ('--', 'threadbare'), ('threadbare',
'in'), ('in', 'coat'), ('coat', ','), (',', 'heart'), ('heart', ','), (',', 'body'),
('body', ','), (',', 'and'), ('and', 'brain'), ('brain', ';'), (';', 'I'), ('I',
'see'), ('see', 'him'), ('him', 'now'), ('now', '.'), ('.', 'He'), ('He', 'was'),
('was', 'ever'), ('ever', 'dusting'), ('dusting', 'his'), ('his', 'old'), ('old',
'lexicons'), ('lexicons', 'and')]
```

As seen, the bigram() splits the words into pairs.

2. Trigrams, as the name suggests, are the combination of three words that follow each other. To extract the trigrams in a text, the trigram() method is called, passing the tokens as an argument. Let's see the example below.

```
#prints the trigrams for the first 50 words in the text
print(list(nltk.trigrams(text[:50])))
```

Output:

```
[('[', 'Moby', 'Dick'), ('Moby', 'Dick', 'by'), ('Dick', 'by', 'Herman'), ('by',
'Herman', 'Melville'), ('Herman', 'Melville', '1851'), ('Melville', '1851', ']'),
('1851', ']', 'ETYMOLOGY'), (']', 'ETYMOLOGY', '.'), ('ETYMOLOGY', '.', '('), ('.',
'(', 'Supplied'), ('(', 'Supplied', 'by'), ('Supplied', 'by', 'a'), ('by', 'a',
'Late'), ('a', 'Late', 'Consumptive'), ('Late', 'Consumptive', 'Usher'),
('Consumptive', 'Usher', 'to'), ('Usher', 'to', 'a'), ('to', 'a', 'Grammar'), ('a',
'Grammar', 'School'), ('Grammar', 'School', ')'), ('School', ')', 'The'), (')',
'The', 'pale'), ('The', 'pale', 'Usher'), ('pale', 'Usher', '--'), ('Usher', '--',
'threadbare'), ('--', 'threadbare', 'in'), ('threadbare', 'in', 'coat'), ('in',
'coat', ','), ('coat', ',', 'heart'), (',', 'heart', ','), ('heart', ',', 'body'),
(',', 'body', ','), ('body', ',', 'and'), (',', 'and', 'brain'), ('and', 'brain',
';'), ('brain', ';', 'I'), (';', 'I', 'see'), ('I', 'see', 'him'), ('see', 'him',
'now'), ('him', 'now', '.'), ('now', '.', 'He'), ('.', 'He', 'was'), ('He', 'was',
'ever'), ('was', 'ever', 'dusting'), ('ever', 'dusting', 'his'), ('dusting', 'his',
'old'), ('his', 'old', 'lexicons'), ('old', 'lexicons', 'and')]
```

Here, the words are split into threes.

Bigrams and trigrams are especially useful for extracting features in text-based sentimental analysis. A high concentration of bigrams or trigrams in a text is an indication of a keyword or a key feature.

Facebook Comments