Sentiment Analysis: Python's NLTK Library

Sentiment analysis can help you determine the ratio of positive to negative engagements about a specific topic. You can analyse bodies of text, such as comments, tweets, and product reviews, to obtain insights from your audience. Here we will explore the important features of NLTK for processing text data and the different approaches you can use to perform sentiment analysis on your data.

Getting Started With NLTK

The NLTK library contains various utilities that allow you to effectively manipulate and analyse linguistic data. Among its advanced features are **text classifiers** that you can use for many kinds of classification, including sentiment analysis.

Sentiment analysis is the practice of using algorithms to classify various samples of related text into overall positive and negative categories. With NLTK, you can employ these algorithms through powerful built-in machine learning operations to obtain insights from linguistic data.

Installing and Importing

We need to obtain a few additional resources. Some of them are text samples, and others are data models that certain NLTK functions require.

To get the resources you'll need, use `nltk.download()`:

```
import nltk

nltk.download()
```

NLTK will display a download manager showing all available and installed resources. Here are the ones you'll need to download for this tutorial:

- **`names`:** A list of common English names compiled by Mark Kantrowitz
- **`stopwords`:** A list of really common words, like articles, pronouns, prepositions, and conjunctions
- **`state_union`:** A sample of transcribed State of the Union addresses by different US presidents, compiled by Kathleen Ahrens
- **`twitter_samples`:** A list of social media phrases posted to Twitter
- **`movie_reviews`:** Two thousand movie reviews categorized by Bo Pang and Lillian Lee
- **`averaged_perceptron_tagger`:** A data model that NLTK uses to categorize words into their part of speech
- **`vader_lexicon`:** A scored list of words and jargon that NLTK references when performing sentiment analysis, created by C.J. Hutto and Eric Gilbert

- **`punkt`:** A data model created by Jan Strunk that NLTK uses to split full texts into word lists

Note: Throughout this tutorial, you'll find many references to the word **corpus** and its plural form, **corpora**. A corpus is a large collection of related text samples. In the context of NLTK, corpora are compiled with features for natural language processing (NLP), such as categories and numerical scores for particular features.

A quick way to download specific resources directly from the console is to pass a list to `nltk.download()`:

```
>>> import nltk

>>> nltk.download([
...     "names",
...     "stopwords",
...     "state_union",
...     "twitter_samples",
...     "movie_reviews",
...     "averaged_perceptron_tagger",
...     "vader_lexicon",
...     "punkt",
... ])
[nltk_data] Downloading package names to /home/user/nltk_data...
[nltk_data]   Unzipping corpora/names.zip.
[nltk_data] Downloading package stopwords to /home/user/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package state_union to
[nltk_data]     /home/user/nltk_data...
[nltk_data]   Unzipping corpora/state_union.zip.
[nltk_data] Downloading package twitter_samples to
[nltk_data]     /home/user/nltk_data...
[nltk_data]   Unzipping corpora/twitter_samples.zip.
[nltk_data] Downloading package movie_reviews to
[nltk_data]     /home/user/nltk_data...
[nltk_data]   Unzipping corpora/movie_reviews.zip.
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /home/user/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
[nltk_data] Downloading package vader_lexicon to
[nltk_data]     /home/user/nltk_data...
[nltk_data] Downloading package punkt to /home/user/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
True
```

This will tell NLTK to find and download each resource based on its identifier.

Should NLTK require additional resources that you haven't installed, you'll see a helpful `LookupError` with details and instructions to download the resource:

```
>>> import nltk

>>> w = nltk.corpus.shakespeare.words()
...
LookupError:
**********************************************************************
  Resource shakespeare not found.
  Please use the NLTK Downloader to obtain the resource:

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

The `LookupError` specifies which resource is necessary for the requested operation along with instructions to download it using its identifier.

Compiling Data

NLTK provides a number of functions that you can call with few or no arguments that will help you meaningfully analyse text before you even touch its machine learning capabilities. Many of NLTK's utilities are helpful in preparing your data for more advanced analysis.

Soon, you'll learn about frequency distributions, concordance, and collocations. But first, you need some data.

Start by loading the State of the Union corpus you downloaded earlier:

```
words = [w for w in nltk.corpus.state_union.words() if w.isalpha()]
```

Note that you build a list of individual words with the corpus's `.words()` method, but you use `str.isalpha()` to include only the words that are made up of letters. Otherwise, your word list may end up with "words" that are only punctuation marks.

Have a look at your list. You'll notice lots of little words like "of," "a," "the," and similar. These common words are called **stop words**, and they can have a negative effect on your analysis because they occur so often in the text. Thankfully, there's a convenient way to filter them out.

NLTK provides a small corpus of stop words that you can load into a list:

```
stopwords = nltk.corpus.stopwords.words("english")
```

Make sure to specify `english` as the desired language since this corpus contains stop words in various languages.

Now you can remove stop words from your original word list:

```
words = [w for w in words if w.lower() not in stopwords]
```

Since all words in the `stopwords` list are lowercase, and those in the original list may not be, you use `str.lower()` to account for any discrepancies. Otherwise, you may end up with mixedCase or capitalized stop words still in your list.

While you'll use corpora provided by NLTK for this tutorial, it's possible to build your own text corpora from any source. Building a corpus can be as simple as loading some plain text or as complex as labelling and categorizing each sentence. Refer to NLTK's documentation for more information on how to work with corpus readers.

For some quick analysis, creating a corpus could be overkill. If all you need is a word list, there are simpler ways to achieve that goal. Beyond Python's own string manipulation methods, NLTK provides `nltk.word_tokenize()`, a function that splits raw text into individual words. While **tokenization** is itself a bigger topic (and likely one of the steps you'll take when creating a custom corpus), this tokenizer delivers simple word lists really well.

To use it, call `word_tokenize()` with the raw text you want to split:

```
>>> from pprint import pprint

>>> text = """
... For some quick analysis, creating a corpus could be overkill.
... If all you need is a word list,
... there are simpler ways to achieve that goal."""
>>> pprint(nltk.word_tokenize(text), width=79, compact=True)
['For', 'some', 'quick', 'analysis', ',', 'creating', 'a', 'corpus',
'could',
 'be', 'overkill', '.', 'If', 'all', 'you', 'need', 'is', 'a', 'word',
'list',
 ',', 'there', 'are', 'simpler', 'ways', 'to', 'achieve', 'that', 'goal',
'.']
```

Now you have a workable word list! Remember that punctuation will be counted as individual words.

Creating Frequency Distributions

Now you're ready for **frequency distributions**. A frequency distribution is essentially a table that tells you how many times each word appears within a given text. In NLTK, frequency distributions are a specific object type implemented as a distinct class called `FreqDist`. This class provides useful operations for word frequency analysis.

To build a frequency distribution with NLTK, construct the `nltk.FreqDist` class with a word list:

```
tkwords= nltk.word_tokenize(text)
fd = nltk.FreqDist(tkwords)
```

This will create a frequency distribution object similar to a Python dictionary but with added features.

After building the object, you can use methods like `.most_common()` and `.tabulate()` to start visualizing information:

```
>>> fd.most_common(6)
[(',', 3), ('a', 2), ('For', 1), ('some', 1), ('quick', 1), ('analysis',
1)]
>>> fd.tabulate(6)
  ,        a      For     some    quick analysis
  3        2        1        1        1        1
```

These methods allow you to quickly determine frequently used words in a sample. With `.most_common()`, you get a list of tuples containing each word and how many times it appears in your text. You can get the same information in a more readable format with `.tabulate()`.

In addition to these two methods, you can use frequency distributions to query particular words. You can also use them as iterators to perform some custom analysis on word properties.

For example, to discover differences in case, you can query for different variations of the same word:

```
>>> fd["analysis"]
1
>>> fd["Analysis"]  # Note this doesn't result in a KeyError
0
>>> fd["ANALYSIS"]
0
```

These return values indicate the number of times each word occurs exactly as given.

Since frequency distribution objects are iterable, you can use them within list comprehensions to create subsets of the initial distribution. You can focus these subsets on properties that are useful for your own analysis.

Try creating a new frequency distribution that's based on the initial one but normalizes all words to lowercase:

```
lower_fd = nltk.FreqDist([w.lower() for w in fd])
```

Now you have a more accurate representation of word usage regardless of case.

Think of the possibilities: You could create frequency distributions of words starting with a particular letter, or of a particular length, or containing certain letters. Your imagination is the limit!

Extracting Concordance and Collocations

In the context of NLP, a **concordance** is a collection of word locations along with their context. You can use concordances to find:

1. How many times a word appears
2. Where each occurrence appears
3. What words surround each occurrence

In NLTK, you can do this by calling `.concordance()`. To use it, you need an instance of the `nltk.Text` class, which can also be constructed with a word list.

Before invoking `.concordance()`, build a new word list from the original corpus text so that all the context, even stop words, will be there:

```
>>> text = nltk.Text(nltk.corpus.state_union.words())
>>> text.concordance("america", lines=5)
Displaying 5 of 1079 matches:
 would want us to do . That is what America will do . So much blood has
already
ay , the entire world is looking to America for enlightened leadership to
peace
beyond any shadow of a doubt , that America will continue the fight for
freedom
 to make complete victory certain , America will never become a party to
any pl
nly in law and in justice . Here in America , we have labored long and hard
to
```

Note that `.concordance()` already ignores case, allowing you to see the context of all case variants of a word in order of appearance. Note also that this function doesn't show you the location of each word in the text.

Additionally, since `.concordance()` only prints information to the console, it's not ideal for data manipulation. To obtain a usable list that will also give you information about the location of each occurrence, use `.concordance_list()`:

```
>>> concordance_list = text.concordance_list("america", lines=2)
>>> for entry in concordance_list:
...     print(entry.line)
...
 would want us to do . That is what America will do . So much blood has
already
ay , the entire world is looking to America for enlightened leadership to
peace
```

`.concordance_list()` gives you a list of `ConcordanceLine` objects, which contain information about where each word occurs as well as a few more properties worth exploring. The list is also sorted in order of appearance.

The `nltk.Text` class itself has a few other interesting features. One of them is `.vocab()`, which is worth mentioning because it creates a frequency distribution for a given text.

Revisiting `nltk.word_tokenize()`, check out how quickly you can create a custom `nltk.Text` instance and an accompanying frequency distribution:

```
>>> new_words= nltk.word_tokenize(
...     """Beautiful is better than ugly.
...     Explicit is better than implicit.
...     Simple is better than complex."""
... )
>>> text = nltk.Text(new_words)
>>> fd = text.vocab()  # Equivalent to fd = nltk.FreqDist(words)
>>> fd.tabulate(3)
    is better   than
     3      3      3
```

`.vocab()` is essentially a shortcut to create a frequency distribution from an instance of `nltk.Text`. That way, you don't have to make a separate call to instantiate a new `nltk.FreqDist` object.

Another powerful feature of NLTK is its ability to quickly find **collocations** with simple function calls. Collocations are series of words that frequently appear together in a given text. In the State of the Union corpus, for example, you'd expect to find the words *United* and *States* appearing next to each other very often. Those two words appearing together is a collocation.

Collocations can be made up of two or more words. NLTK provides classes to handle several types of collocations:

- **Bigrams:** Frequent two-word combinations
- **Trigrams:** Frequent three-word combinations
- **Quadgrams:** Frequent four-word combinations

NLTK provides specific classes for you to find collocations in your text. Following the pattern you've seen so far, these classes are also built from lists of words:

```
words = [w for w in nltk.corpus.state_union.words() if w.isalpha()]
finder = nltk.collocations.TrigramCollocationFinder.from_words(words)
```

The `TrigramCollocationFinder` instance will search specifically for trigrams. As you may have guessed, NLTK also has the `BigramCollocationFinder` and `QuadgramCollocationFinder` classes for bigrams and quadgrams, respectively. All these classes have a number of utilities to give you information about all identified collocations.

One of their most useful tools is the `ngram_fd` property. This property holds a frequency distribution that is built for each collocation rather than for individual words.

Using `ngram_fd`, you can find the most common collocations in the supplied text:

```
>>> finder.ngram_fd.most_common(2)
[(('the', 'United', 'States'), 294), (('the', 'American', 'people'), 185)]
>>> finder.ngram_fd.tabulate(2)
  ('the', 'United', 'States') ('the', 'American', 'people')
                          294                           185
```

You don't even have to create the frequency distribution, as it's already a property of the collocation finder instance.

Now that you've learned about some of NLTK's most useful tools, it's time to jump into sentiment analysis!