11
Practical Sentiment Analysis

This is going to be a fun chapter. In this chapter, we will explore and demonstrate some practical examples of using **Natural Language Processing** (**NLP**) concepts to understand how unstructured text can be turned into insights. In `Chapter 10`, *Exploring Text Data and Unstructured Data*, we explored the **Natural Language Toolkit** (**NLTK**) library and some fundamental features of working with identifying words, phrases, and sentences. In that process of tokenizing, we learned how to work with data and classify text, but did not go beyond that. In this chapter, we will learn about sentiment analysis, which predicts the underlying tone of text that's input into an algorithm. We will break down the elements that make up an NLP model and the packages used for sentiment analysis before walking through an example together.

In this chapter, we will cover the following topics:

- Why sentiment analysis is important
- Elements of an NLP model
- Sentiment analysis packages
- Sentiment analysis in action

Let's get started.

Technical requirements

You can find the GitHub repository for this book at `https://github.com/PacktPublishing/Practical-Data-Analysis-using-Jupyter-Notebook/tree/master/Chapter11`.

You can download and install the required software for this chapter from the following link: `https://www.anaconda.com/products/individual`.

Why sentiment analysis is important

Today, we are all living in a digital age where data is entangled in our daily lives. However, since most of this data is unstructured and the volume of it is large, it requires statistical libraries and **machine learning** (**ML**) to apply it to technology solutions. The NLTK libraries serve as a framework for us to work with unstructured data, and sentiment analysis serves as a practical use case in NLP. **Sentiment analysis**, or opinion mining, is a type of supervised ML that requires a training dataset to accurately predict an input sentence, phrase, headline, or even tweet is positive, negative, or neutral. Once the model has been trained, you can pass unstructured data into it, like a function, and it will return a value between negative one and positive one. The number will output decimals, and the closer it is to an integer, the more confident the model's accuracy will be. Sentiment analysis is an evolving science, so our focus will be on using the NLTK corpus libraries. As with any NLP model, you will find inaccuracies in the predicted output if you don't have a good sample for the input training data.

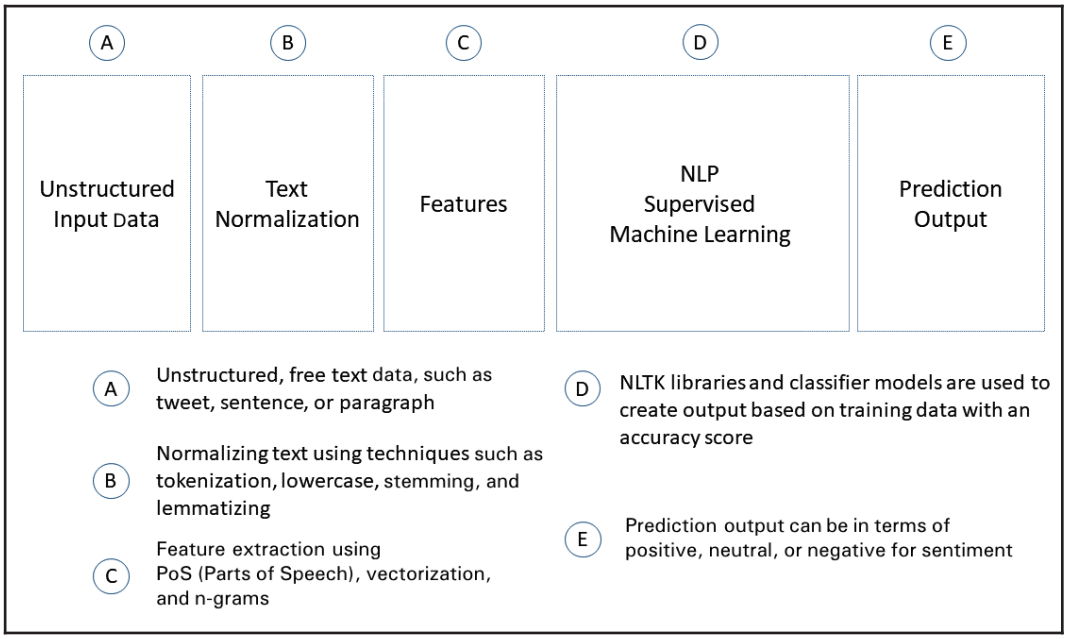
Also, note that NLP and sentiment analysis is a deep subject and should be validated by a data scientist or ML engineering team if you plan on implementing your own models using internal company data sources. That being said, you will notice sentiment analysis in many different applications today, and the exercises in this chapter provide you with another tool for data analysis. Another benefit of learning about how to use sentiment analysis is that it allows you to argue about the data that's output from a model. The ability to defend the accuracy and predictive nature of working with unstructured data will increase your data literacy skills. For example, let's say you are analyzing a population of tweets about a restaurant for a marketing campaign that had a mix of positive and negative reviews in the past. If the results of your analysis come back as 100% positive, you should start questioning the training data, the source of the data, and the model itself. Of course, it's possible for all the tweets to be positive, especially against a small population of data, but is it likely that every single one has a positive sentiment?

This is why **Knowing Your Data** (**KYD**) remains important, as covered in `Chapter 1`, *Fundamentals of Data Analysis*, regardless of the technology and tools being used to analyze it. However, why sentiment analysis is important today needs to be stated. First, the accuracy of the models has significantly improved because the more training data there is, the better the prediction's output. The second point is that NLP models can scale beyond what a human can process in the same amount of time. Finally, the alternatives to sentiment analysis available today, such as expert systems, are more costly because of the time and resources required to implement them. Expert system development using text-based logic and wildcard keyword searches is rigid and difficult to maintain.

Now, let's explore what makes up the elements of NLP and the process of how it is used in sentiment analysis.

Elements of an NLP model

To summarize the process required to use an NLP supervised ML model for sentiment analysis, I have created the following diagram, which shows the elements in a logical progression indicated by the letters **A** through **E**:



The process begins with our source **Unstructured Input Data**, which is represented in the preceding diagram with the letter **A**. Since unstructured data has different formats, structures, and forms such as a tweet, sentence, or paragraph, we need to perform extra steps to work with the data to gain any insights.

The next element is titled **Text Normalization** and is represented by the letter **B** in the preceding diagram, and involves concepts such as tokenization, n-grams, and **bag-of-words** (**BoW**), which were introduced in `Chapter 10`, *Exploring Text Data and Unstructured Data*. Let's explore them in more detail so that we can learn how they are applied in sentiment analysis. BoW is when a string of text such as a sentence or paragraph is broken down to determine how many times a word occurs. In the process of **tokenizing** to create the bag-of-words representation, the location of where the word appears in a sentence, tweet, or paragraph becomes less relevant. How each word is classified, categorized, and defined using a classifier will serve as input to the next process.

Think of tokens and bag-of-words as raw ingredients to the sentiment analysis recipe; as in cooking, the ingredients take additional steps of refinement. Hence, the concept of classification becomes important. This is considered a **Features** and is represented by the letter **C** in the preceding diagram. Because tokens are nothing more than ASCII characters to a computer, word embedding and tagging is the process of converting the words into an input for an ML model. An example would be to classify each word with a pair value such as a one or zero to represent true or false. This process also includes finding similar words or groupings in order to interpret the context.

Creating **Features** is known as feature engineering, which is the foundation of supervised ML. Feature engineering is the process of transforming unstructured data elements into specific inputs for the prediction model. Models are abstractions where the output is only as accurate as the input data behind it. This means models need training data with extracted features to improve their accuracy. Without feature engineering, the results of a model would be random guesses.

Creating a prediction output

To see how **features** can be extracted from unstructured data, let's walk through the NLTK gender feature, which includes some minor modifications from the original example. You can find the original source in the *Further reading* section.

Launch a new Jupyter Notebook and name it `ch_11_exercises`. Now, follow these steps:

1. Import the following libraries by adding the following command to your Jupyter Notebook and run the cell. Feel free to follow along by creating your own Notebook. I have placed a copy in this book's GitHub repository for reference:

```
In[]: import nltk
```

The library should already be available using Anaconda. Refer to `Chapter 2`, *Overview of Python and Installing Jupyter Notebook*, for help with setting up your environment.

2. Next, we need to download the specific corpus we want to use. Alternatively, you can download all the packages using the `all` parameter. If you are behind a firewall, there is an `nltk.set_proxy` option available. Check the documentation at `nltk.org` for more details:

```
In[]: nltk.download("names")
```

The output will look as follows, where the package download is confirmed and the output is verified as `True`:

```
In [19]: nltk.download("names")

         [nltk_data] Downloading package names to /home/nbuser/nltk_data...
         [nltk_data]   Package names is already up-to-date!

Out[19]: True
```

3. We can use the following command to reference the corpus:

```
In[]: from nltk.corpus import names
```

4. To explore the data available in this corpus, let's run the `print` command against the two input sources, `male.txt` and `female.txt`:

```
In[]: print("Count of Words in male.txt:",
len(names.words('male.txt')))
print("Count of Words in female.txt:",
len(names.words('female.txt')))
```

The output will look as follows, where a count of the number of words found in each source file is printed in the Notebook:

```
In [37]: print("Count of Words in male.txt:", len(names.words('male.txt')))
         print("Count of Words in female.txt:", len(names.words('female.txt')))

         Count of Words in male.txt: 2943
         Count of Words in female.txt: 5001
```

We now have a better understanding of the size of the data due to counting the number of words found in each source file. Let's continue by looking at the contents within each source, taking a look at a few samples from each gender file.

5. To see a list of the first few words found in each source, let's run the `print` command against the two input sources, `male.txt` and `female.txt`:

```
In[]: print("Sample list Male names:",
names.words('male.txt')[0:5])
print("Sample list Female names:", names.words('female.txt')[0:5])
```

The output will look as follows, where a list of words found in each source file is printed in the Notebook:

```
In [40]: print("Sample list Male names:", names.words('male.txt')[0:5])
         print("Sample list Female names:", names.words('female.txt')[0:5])

         Sample list Male names: ['Aamir', 'Aaron', 'Abbey', 'Abbie', 'Abbot']
         Sample list Female names: ['Abagael', 'Abagail', 'Abbe', 'Abbey', 'Abbi']
```

Remember that the computer has no idea if a name actually returns a value of `male` or `female`. The corpus has defined them as two different source files as a list of values that the NLTK library has identified as words because they have been defined as such. With thousands of names defined as either male or female, you can use this data as input for sentiment analysis. However, identifying gender alone will not determine whether the sentiment is positive or negative, so additional elements are required.

The next element, labeled **D** in the first diagram, is the actual **NLP supervised ML** algorithm. Remember, building an accurate model involves using feature engineering, along with NLTK libraries and classifier models. When used correctly, the output will be based on the input **training** and **test** data. Models should always be validated and the accuracy should be measured. For our example, which is building a basic gender determination model, we are going to use `NaiveBayesClassifier`, which is available in the NLKT libraries. The Naïve Bayes Classifier is an ML model created from Bayes theorem that is used to determine the probability of an event happening based on how often another similar event has occurred. A classifier is a process that chooses the correct tag value or label based on an inputted feature dataset. The mathematical concepts behind these models and libraries are vast, so I have added some links in the *Further reading* section for additional reference. To complete the elements of sentiment analysis summarized in the first diagram, we will create a prediction output, so let's continue in our Jupyter Notebook session:

1. Create a `gender_features` function that returns the last letter of any input word. The model will use this classifier feature as input to predict the output, which, based on the concept that first names that end in the letters **A**, **E**, and **I** are more likely to be female, while first names ending in **K**, **O**, **R**, **S**, or **T** are more likely to be male. There will be no output after you run the cell:

```
In[]: def gender_features(word):
          return {'last_letter': word[-1]}
```

Remember to indent the second line in your cell so that Python can process the function.

2. To confirm the function will return a value, enter the following command, which prints the last character of any inputted name or word:

```
In[]: gender_features('Debra')
```

The output will look as follows, where the last character from the inputted word `Debra` is printed in the Notebook with `Out[]`:

```
In [16]: gender_features('Debra')
Out[16]: {'last_letter': 'a'}
```

3. Create a new variable named `labeled_names` that loops through both source gender files and assigns a **name-value pair** so that it can be identified as either male or female to be input into the model. To see the results after the loop has completed, we print the first few values to verify that the `labeled_names` variable contains data:

```
In[]: labeled_names = ([(name, 'male') for name in
names.words('male.txt')] + [(name, 'female') for name in
names.words('female.txt')])
print(labeled_names[0:5])
```

The output will look as follows, where each name value from the source file will be combined with a tag of `male` or `female`, depending on which text file source it came from:

```
In [44]: labeled_names = ([(name, 'male') for name in names.words('male.txt')] +
                          [(name, 'female') for name in names.words('female.txt')])
         print(labeled_names[0:5])

         [('Aamir', 'male'), ('Aaron', 'male'), ('Abbey', 'male'), ('Abbie', 'male'), ('Abbot', 'male')]
```

4. Since the model should be trained using a random list of values to avoid any bias, we will input the random function and shuffle all the name and gender combinations, which will change the sequence of how they are stored in the `labeled_names` variable. I added a `print()` statement so that you can see the difference from the output created in the prior step:

```
In[]: import random
random.shuffle(labeled_names)
print(labeled_names[0:5])
```

The output will look as follows, where each name value from the source file will be combined with a tag of `male` or `female`, depending on which text file source it came from:

```
In [46]: import random
         random.shuffle(labeled_names)
         print(labeled_names[0:5])

         [('Lindie', 'female'), ('Krysta', 'female'), ('Cathy', 'female'), ('Orin', 'male'), ('Siouxie', 'female')]
```

Note because the `random()` function is used, the results of the `print()` function will always change each time you run the cell.

5. Next, we are going to train the model by creating features for each gender using the last letter from each name in the `labeled_names` variable. We will print the new variable called `featuresets` so that you can see how the feature will be used in the next step:

```
In[]: featuresets = [(gender_features(n), gender) for (n, gender)
in labeled_names]
print(featuresets[0:5])
```

The output will look as follows, where each combination of the last letter from the names is assigned to a gender value, thereby creating a list of name-value pairs:

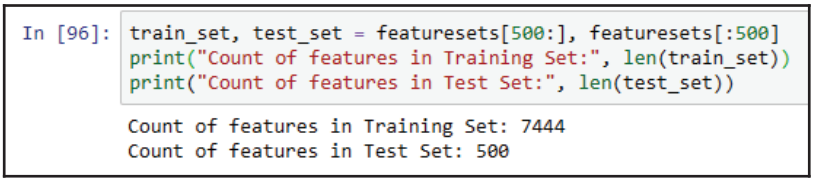
```
In [61]: featuresets = [(gender_features(n), gender) for (n, gender) in labeled_names]
         print(featuresets[0:5])

         [({'last_letter': 'e'}, 'female'), ({'last_letter': 'a'}, 'female'), ({'last_letter': 'y'},
         'female'), ({'last_letter': 'n'}, 'male'), ({'last_letter': 'e'}, 'female')]
```

6. Next, we are going to slice the data from the `featuresets` variable list into two input datasets called `train_set` and `test_set`. Once we have those datasets separated, we can use `train_set` as an input for the classifier. We use the `len()` function to give us a sense of the size of each dataset:

```
In[]: train_set, test_set = featuresets[500:], featuresets[:500]
print("Count of features in Training Set:", len(train_set))
print("Count of features in Test Set:", len(test_set))
```

The output will look as follows, where the results of the `len()` function provide context as to how large each dataset is compared to the others:



7. We will now pass the `train_set` variable as input to the NLTK Naïve Bayes classifier. The model is assigned the name `classifier`, so you can call it like a function in the next step. There will be no output once you run the cell:

```
In[]: classifier = nltk.NaiveBayesClassifier.train(train_set)
```

8. Now, we will validate the results of the model by sending random names into the model using the following commands:

```
In[]: classifier.classify(gender_features('Aaron'))
classifier.classify(gender_features('Marc'))
classifier.classify(gender_features('Debra'))
classifier.classify(gender_features('Deb'))
classifier.classify(gender_features('Seth'))
```

The output will look as follows, where the gender values of either `male` or `female` will be displayed after each name is passed as a parameter in the `classifier` model:

```
In [103]: classifier = nltk.NaiveBayesClassifier.train(train_set)

In [104]: classifier.classify(gender_features('Aaron'))
Out[104]: 'male'

In [105]: classifier.classify(gender_features('Marc'))
Out[105]: 'male'

In [106]: classifier.classify(gender_features('Debra'))
Out[106]: 'female'

In [107]: classifier.classify(gender_features('Deb'))
Out[107]: 'male'

In [108]: classifier.classify(gender_features('Seth'))
Out[108]: 'female'
```

Congratulations – you have successfully created your first supervised ML model! As you can see, the **classifier** model has some accuracy issues and returns incorrect values in some cases. For example, when you pass in the values of `Aaron`, `Marc`, or `Debra`, the gender results are predicted correctly. The name `Aaron` was found in the training data, so that was no surprise. However, the model shows signs of being incomplete or requiring additional features because it returns the incorrect gender when using the nickname of `Deb` for `Debra` and for the name `Seth`, who is male.

How do we solve this problem? There are a few approaches that can be used, all of which we will explore next.

Sentiment analysis packages

The NLTK libraries include a few packages to help solve the issues we experienced in the gender classifier model. The first is the `SentimentAnalyzer` module, which allows you to include additional features using built-in functions. What's special about these packages is that they go beyond traditional functions where defined parameters are passed in. In Python, arguments (`args`) and keyword arguments (`kwargs`) allow us to pass name-value pairs and multiple argument values into a function. These are represented with asterisks; for example, `*args` or `**kwargs`. The NLTK `SentimentAnalyzer` module is a useful utility for teaching purposes, so let's continue by walking through the features that are available within it.

The second is called **VADER**, which stands for **Valence Aware Dictionary and Sentiment Reasoner**. It was built to handle social media data. The VADER sentiment library has a dictionary known as a **lexicon** and includes a rule-based algorithm specifically built to process acronyms, emoticons, and slang. A nice feature available from VADER is that it already includes training data and we can use a built-in function called `polarity_scores()` that returns key insights in the output that's displayed. The first is a compound score that is between negative one and positive one. This provides you with a normalized sum of VADER's lexicon ratings in a single score. For example, if the output returns `0.703`, this would be an extremely positive sentence, while a compound score of `-0.5719` would be interpreted as negative. The next output from the VADER tool is a distribution score in terms of how positive, negative, or neutral the input is from zero to one.

For example, the sentence `I HATE my school!` would return the results shown in the following screenshot:

```
In [34]: my_input_sentence = "I HATE my school!"
         my_analyzer.polarity_scores(my_input_sentence)

Out[34]: {'compound': -0.6932, 'neg': 0.703, 'neu': 0.297, 'pos': 0.0}
```

As you can see, a compound value of `-0.6932` is returned, which validates the VADER model is accurately predicting the sentiment as very negative. On the same output line, you can see `'neg'`, `'neu'`, and `'pos'`, which are short for negative, neutral, and positive, respectively. Each metric next to the values provides a little more detail about how the compound score was derived. In the preceding screenshot, we can see a value of `0.703`, which means that the model prediction is 70.3% negative, with the remaining 29.7% being neutral. The model returned a value of `0.0` next to `pos`, so there is a `0%` positive sentiment based on the built-in VADER training dataset.

Note that the VADER sentiment analysis scoring methodology has been trained to handle social media data and informal proper grammar. For example, if a tweet includes multiple exclamation points for emphasis, the compound score will increase. Capitalization, the use of conjunctions, and the use of swear words will all be accounted for in the output from the model. So, the main benefit of using VADER is that it already includes those extra steps required to feature and train the model, but you lose the ability to customize it with additional features.

Now that we have a better understanding of the VADER tool, let's walk through an example of using it.

Sentiment analysis in action

Let's continue with our Jupyter Notebook session and walk through how to install and use the VADER sentiment analysis library. First, we will walk through an example of using manual input and then learn how to load data from a file.

Manual input

Follow these steps to learn how to use manual input in VADER:

1. Import the NLTK library and download the `vader_lexicon` library so that all the necessary functions and features will be available:

```
In[]: import nltk
nltk.download('vader_lexicon')
```

   The output will look as follows, where the package download will be confirmed and the output is verified as `True`:

```
In [21]: import nltk

In [22]: nltk.download('vader_lexicon')
         [nltk_data] Downloading package vader_lexicon to
         [nltk_data]     /home/nbuser/nltk_data...
         [nltk_data]   Package vader_lexicon is already up-to-date!
Out[22]: True
```

2. Import `SentimentIntensityAnalyzer` from the NLTK Vader library. There will be no output when you run the cell:

```
In[]:from nltk.sentiment.vader import SentimentIntensityAnalyzer
```

3. To make it easier, we will assign a variable object called `my_ analyzer` and assign it to the `SentimentIntensityAnalyzer()` model. There will be no output after you run the cell:

```
In[]:my_analyzer = SentimentIntensityAnalyzer()
```

4. Next, we will create a variable named `my_input_sentence` and assign it a string value of `I HATE my school!`. On the second line, we will call the model and pass the variable as an argument to the `polarity_scores()` function:

```
In[]:my_input_sentence = "I HATE my school!"
my_analyzer.polarity_scores(my_input_sentence)
```

The output will look as follows, where we can see the result of the VADER sentiment analysis model:

```
In [34]: my_input_sentence = "I HATE my school!"
         my_analyzer.polarity_scores(my_input_sentence)
Out[34]: {'compound': -0.6932, 'neg': 0.703, 'neu': 0.297, 'pos': 0.0}
```

Excellent—you have now utilized the VADER sentiment analysis model and returned results to determine whether a sentence is positive or negative. Now that we understand how the model works with individual input sentences, let's demonstrate how to work with a sample social media file and combine it with what we have learned using the `pandas` and `matplotlib` libraries.

In the next exercise, we are going to work with a text file source that you will need to import into your Jupyter Notebook. This is a small sample CSV file containing example social media type free text, including a hashtag, informal grammar, and extra punctuation.

It has 2 columns and 10 rows of content, with a header row for easy reference, as shown in the following screenshot:

```
id,text
1,I Hate my School!!!
2,@socialmediahandle I learned something new today
3,I need to take a cool trip to Austraila!
4,The restaurant service was amazing!
5,I will never go back there again!
6,You learn something new every day - this place is great
7,First Impressions - this is good but then it went downhill from there
8,A bit pricey
9,Love them
10,meh
```

Social media file input

Let's continue working with our Jupyter Notebook session and walk through how to work with this source file so that it includes a VADER sentiment and then analyze the results:

1. We are going to import some additional libraries so that we can work with and analyze the results, as follows:

```
In[]:import pandas as pd
import numpy as np
%matplotlib inline
```

2. We also have to install a new library named `twython`. Use the following command to install it in your Notebook session. The `twython` library includes features to make it easier to read social media data:

```
In[]:!pip install twython
```

The output will look as follows, where the resulting installation will be displayed. If you need to upgrade `pip`, you may need to run additional commands:

```
In [30]: !pip install twython

         Requirement already satisfied: twython in /home/nbuser/anaconda3_420/lib/python3.5/site-packages (3.8.2)
         Requirement already satisfied: requests>=2.1.0 in /home/nbuser/anaconda3_420/lib/python3.5/site-packages (from twython) (2.14.
         2)
         Requirement already satisfied: requests-oauthlib>=0.4.0 in /home/nbuser/anaconda3_420/lib/python3.5/site-packages (from twytho
         n) (1.3.0)
         Requirement already satisfied: oauthlib>=3.0.0 in /home/nbuser/anaconda3_420/lib/python3.5/site-packages (from requests-oauthli
         b>=0.4.0->twython) (3.1.0)
         WARNING: You are using pip version 19.3.1; however, version 20.1 is available.
         You should consider upgrading via the 'pip install --upgrade pip' command.
```

3. If required, re-import the NLTK library and import the `SentimentIntensityAnalyzer` module. No output will be displayed after you run the cell:

```
In[]:import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
```

4. Define a variable as `analyzer` to make it easier to reference later in the code. No output will be displayed after you run the cell:

```
In[]:analyzer = SentimentIntensityAnalyzer()
```

5. If required, redownload the NLTK `vader_lexicon`:

```
In[]:nltk.download('vader_lexicon')
```

The output will look as follows, where the download result will be displayed:

```
In [33]: nltk.download('vader_lexicon')

         [nltk_data] Downloading package vader_lexicon to
         [nltk_data]     /home/nbuser/nltk_data...
         [nltk_data]   Package vader_lexicon is already up-to-date!

Out[33]: True
```

6. Now, we will read in the `.csv` file using the `pandas` library and assign the result to a variable named `sentences`. To validate the results, you can run the `len()` function:

```
In[]:sentences = pd.read_csv('social_media_sample_file.csv')
len(sentences)
```

Be sure to upload the source CSV file in the correct file location so that you can reference it in your Jupyter Notebook.

The output will look as follows, where the value of `10` will be displayed. This matches the number of records in the source CSV file:

```
In [34]: sentences = pd.read_csv('social_media_sample_file.csv')
         len(sentences)

Out[34]: 10
```

7. To preview the data and verify that your DataFrame is loaded correctly, you can run the `head()` command:

```
In[]:sentences.head()
```

The output will look as follows, where the results of the `head()` function are displayed to verify that the source file is now a DataFrame:

```
In [35]: sentences.head()

Out[35]:
```

	id	text
0	1	I Hate my School!!!
1	2	@socialmediahandle I learned something new today
2	3	I need to take a cool trip to Austraila!
3	4	The restaurant service was amazing!
4	5	I will never go back there again!

8. The following block of code includes a few steps that look through the DataFrame, analyze the text source, apply the VADER sentiment metrics, and assign the results to a `numpy` array for easier usage. No output will be displayed after you run the cell:

```
In[]:i=0 #reset counter for loop

#initialize variables
my_vader_score_compound = [ ]
my_vader_score_positive = [ ]
```

```
my_vader_score_negative = [ ]
my_vader_score_neutral = [ ]

while (i<len(sentences)):

    my_analyzer =
analyzer.polarity_scores(sentences.iloc[i]['text'])
    my_vader_score_compound.append(my_analyzer['compound'])
    my_vader_score_positive.append(my_analyzer['pos'])
    my_vader_score_negative.append(my_analyzer['neg'])
    my_vader_score_neutral.append(my_analyzer['neu'])
    i = i+1
#converting sentiment values to numpy for easier usage
my_vader_score_compound = np.array(my_vader_score_compound)
my_vader_score_positive = np.array(my_vader_score_positive)
my_vader_score_negative = np.array(my_vader_score_negative)
my_vader_score_neutral = np.array(my_vader_score_neutral)
```

Be sure to double-check your indentations when entering multiple commands in the Jupyter Notebook input cell.

9. Now, we can extend the source DataFrame so that it includes the results from the VADER sentiment model. This will create four new columns. No output will be displayed after you run the cell:

```
In[]:sentences['my VADER Score'] = my_vader_score_compound
sentences['my VADER score - positive'] = my_vader_score_positive
sentences['my VADER score - negative'] = my_vader_score_negative
sentences['my VADER score - neutral'] = my_vader_score_neutral
```

10. To see the changes, run the `head()` function again:

```
In[]:sentences.head(10)
```

The output will look as follows, where the results of the `head()` function are displayed to verify that the DataFrame now includes the new columns that were created from the loop in the previous step:

```
In [40]: sentences.head(10)
```

Out[40]:

	id	text	my VADER Score	my VADER score - positive	my VADER score - negative	my VADER score - neutral
0	1	I Hate my School!!!	-0.6784	0.000	0.696	0.304
1	2	@socialmediahandle I learned something new today	0.0000	0.000	0.000	1.000
2	3	I need to take a cool trip to Austraila!	0.3802	0.302	0.000	0.698
3	4	The restaurant service was amazing!	0.6239	0.506	0.000	0.494
4	5	I will never go back there again!	0.0000	0.000	0.000	1.000
5	6	You learn something new every day - this place...	0.6249	0.313	0.000	0.687
6	7	First Impressions - this is good but then it w...	0.3400	0.254	0.000	0.746
7	8	A bit pricey	0.0000	0.000	0.000	1.000
8	9	Love them	0.6369	0.808	0.000	0.192
9	10	meh	-0.0772	0.000	1.000	0.000

11. While this information is useful, it still requires the user to scan through the results row by row. Let's make it easier to analyze and summarize the results by creating a new column that categorizes the compound score results. No output will be displayed after you run the cell:

```
In[]:i=0 #reset counter for loop

#initialize variables
my_prediction = [ ]

while (i<len(sentences)):
    if ((sentences.iloc[i]['my VADER Score'] >= 0.3)):
        my_prediction.append('positive')
    elif ((sentences.iloc[i]['my VADER Score'] >= 0) &
(sentences.iloc[i]['my VADER Score'] < 0.3)):
        my_prediction.append('neutral')
    elif ((sentences.iloc[i]['my VADER Score'] < 0)):
        my_prediction.append('negative')
    i = i+1
```

12. Similar to before, we will take the results and add a new column to our DataFrame called `my prediction sentiment`. No output will be displayed after you run the cell:

```
In[]:sentences['my predicted sentiment'] = my_prediction
```

13. To see the changes, run the `head()` function again:

```
In[]:sentences.head(10)
```

The output will look as follows, where the results of the `head()` function are displayed to verify that the DataFrame now includes the new column that was created from the loop in the previous step:

```
In [62]: sentences.head(10)
```

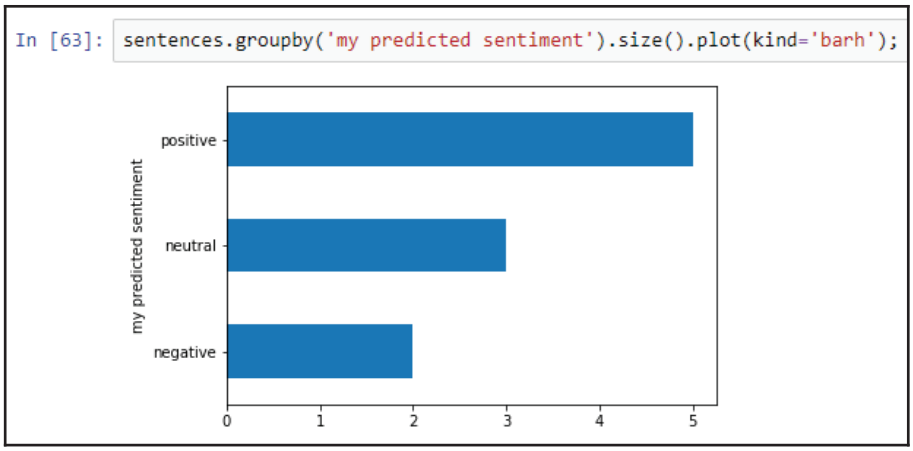
Out[62]:

	id	text	my VADER Score	my VADER score - positive	my VADER score - negative	my VADER score - neutral	predicted sentiment	my predicted sentiment
0	1	I Hate my School!!!	-0.6784	0.000	0.696	0.304	negative	negative
1	2	@socialmediahandle I learned something new today	0.0000	0.000	0.000	1.000	neutral	neutral
2	3	I need to take a cool trip to Austraila!	0.3802	0.302	0.000	0.698	positive	positive
3	4	The restaurant service was amazing!	0.6239	0.506	0.000	0.494	positive	positive
4	5	I will never go back there again!	0.0000	0.000	0.000	1.000	neutral	neutral
5	6	You learn something new every day - this place...	0.6249	0.313	0.000	0.687	positive	positive
6	7	First Impressions - this is good but then it w...	0.3400	0.254	0.000	0.746	positive	positive
7	8	A bit pricey	0.0000	0.000	0.000	1.000	neutral	neutral
8	9	Love them	0.6369	0.808	0.000	0.192	positive	positive
9	10	meh	-0.0772	0.000	1.000	0.000	negative	negative

14. To make it easier to interpret the results, let's create a data visualization against the DataFrame by summarizing the results using an aggregate `groupby`. We'll use the `plot()` function from the `matplotlib` library to display a horizontal bar chart:

```
In[]:sentences.groupby('my predicted
sentiment').size().plot(kind='barh');
```

The output will look as follows, where a horizontal bar chart will be displayed showing a summary of the count of the text by sentiment in terms of positive, negative, and neutral:



As you can see, we have more positive opinions in our data source. It was much faster to interpret the results like this because we visualized the results to make it easier to consume them visually. We now have a reusable workflow to analyze much larger volumes of unstructured data by looking at a source data file and applying the VADER sentiment analysis model to each record. If you replace the sample CSV file with any social media source, you can rerun the same steps and see how the analysis changes.

The accuracy score for VADER models is around 96%, which has been proven to be more accurate than a human interpretation according to research on the subject.

There is some bias in the analysis since the bins of **positive**, **negative**, and **neutral** can be adjusted in the code. As a good data analyst, understanding the bias can help you either adjust it for your specific needs or be able to communicate the challenges of working with free text data.

Summary

Congratulations—you have successfully walked through the foundations of NLP and should have a high-level understanding of supervised ML using the NLTK libraries! Sentiment analysis is a fascinating and evolving science that has many different moving parts. I hope this introduction is a good start to your continued research so that you can utilize it in your data analysis. In this chapter, we learned about the various elements of sentiment analysis, such as feature engineering, along with the process of how an NLP ML algorithm works. We also learned how to install NLP libraries in Jupyter to work with unstructured data, along with how to analyze the results created by a classifier model. With this knowledge, we walked through an example of how to use the VADER sentiment analysis model and visualized the results for analysis.

In our last chapter, `Chapter 12`, *Bringing it all Together*, we will bring together all the concepts we've covered in this book and walk through some real-world examples.

Further reading

- NLTK sentiment analysis example: `https://www.nltk.org/howto/sentiment.html`
- The source code for VADER and its documentation: `https://github.com/cjhutto/vaderSentiment`
- Bayes theorem explained: `https://plato.stanford.edu/entries/bayes-theorem/`
- VADER sentiment analysis research: `http://comp.social.gatech.edu/papers/icwsm14.vader.hutto.pdf`