CHAPTER 10
Natural Language Processing

Natural language processing (NLP) is a subfield of artificial intelligence used to aid computers in understanding natural human language. Most NLP techniques rely on machine learning to derive meaning from human languages. When text has been provided, the computer utilizes algorithms to extract meaning associated with every sentence and collect essential data from them. NLP manifests itself in different forms across many disciplines under various aliases, including (but not limited to) textual analysis, text mining, computational linguistics, and content analysis.

In the financial landscape, one of the earliest applications of NLP was implemented by the US Securities and Exchange Commission (SEC). The group used text mining and natural language processing to detect accounting fraud. The ability of NLP algorithms to scan and analyze legal and other documents at a high speed provides banks and other financial institutions with enormous efficiency gains to help them meet compliance regulations and combat fraud.

In the investment process, uncovering investment insights requires not only domain knowledge of finance but also a strong grasp of data science and machine learning principles. NLP tools may help detect, measure, predict, and anticipate important market characteristics and indicators, such as market volatility, liquidity risks, financial stress, housing prices, and unemployment.

News has always been a key factor in investment decisions. It is well established that company-specific, macroeconomic, and political news strongly influence the financial markets. As technology advances, and market participants become more connected, the volume and frequency of news will continue to grow rapidly. Even today, the volume of daily text data being produced presents an untenable task for even a large team of fundamental researchers to navigate. Fundamental analysis assisted by NLP techniques is now critical to unlock the complete picture of how experts and the masses feel about the market.

In banks and other organizations, teams of analysts are dedicated to poring over, analyzing, and attempting to quantify qualitative data from news and SEC-mandated reporting. Automation using NLP is well suited in this context. NLP can provide in-depth support in the analysis and interpretation of various reports and documents. This reduces the strain that repetitive, low-value tasks put on human employees. It also provides a level of objectivity and consistency to otherwise subjective interpretations; mistakes from human error are lessened. NLP can also allow a company to garner insights that can be used to assess a creditor's risk or gauge brand-related sentiment from content across the web.

With the rise in popularity of live chat software in banking and finance businesses, NLP-based chatbots are a natural evolution. The combination of robo-advisors with chatbots is expected to automate the entire process of wealth and portfolio management.

In this chapter, we present three NLP-based case studies that cover applications of NLP in algorithmic trading, chatbot creation, and document interpretation and automation. The case studies follow a standardized seven-step model development process presented in Chapter 2. Key model steps for NLP-based problems are data preprocessing, feature representation, and inference. As such, these areas, along with the related concepts and Python-based examples, are outlined in this chapter.

"Case Study 1: NLP and Sentiment Analysis–Based Trading Strategies" on page 362 demonstrates the usage of sentiment analysis and word embedding for a trading strategy. This case study highlights key focus areas for implementing an NLP-based trading strategy.

In "Case Study 2: Chatbot Digital Assistant" on page 383, we create a chatbot and demonstrate how NLP enables chatbots to understand messages and respond appropriately. We leverage Python-based packages and modules to develop a chatbot in a few lines of code.

"Case Study 3: Document Summarization" on page 393 illustrates the use of an NLP-based *topic modeling* technique to discover hidden topics or themes across documents. The purpose of this case study is to demonstrate the usage of NLP to automatically summarize large collections of documents to facilitate organization and management, as well as search and recommendations.

In addition to the points mentioned above, this chapter will cover:

- How to perform NLP data preprocessing, including steps such as tokenization, part-of-speech (PoS) tagging, or named entity recognition, in a few lines of code.
- How to use different supervised techniques, including LSTM, for sentiment analysis.

- Understanding the main Python packages (i.e., NLTK, spaCy and TextBlob) and how to use them for several NLP-related tasks.
- How to build a data preprocessing pipeline using the spaCy package.
- How to use pretrained models, such as word2vec, for feature representation.
- How to use models such as LDA for topic modeling.

This Chapter's Code Repository

The Python code for this chapter is included under the Chapter 10 - Natural Language Processing folder of the online GitHub repository for this chapter. For any new NLP-based case study, use the common template from the code repository and modify the elements specific to the case study. The templates are designed to run on the cloud (i.e., Kaggle, Google Colab, and AWS).

Natural Language Processing: Python Packages

Python is one of the best options to build an NLP-based expert system, and a large variety of open source NLP libraries are available for Python programmers. These libraries and packages contain ready-to-use modules and functions to incorporate complex NLP steps and algorithms, making implementation fast, easy, and efficient.

In this section, we will describe three Python-based NLP libraries we've found to be the most useful and that we will be using in this chapter.

NLTK

NLTK is the most famous Python NLP library, and it has led to incredible breakthroughs across several areas. Its modularized structure makes it excellent for learning and exploring NLP concepts. However, it has heavy functionality with a steep learning curve.

NLTK can be installed using the typical installation procedure. After installing NLTK, NLTK Data needs to be downloaded. The NLTK Data package includes a pretrained tokenizer `punkt` for English, which can be downloaded as well:

```
import nltk
import nltk.data
nltk.download('punkt')
```

TextBlob

TextBlob is built on top of NLTK. This is one of the best libraries for fast prototyping or building applications with minimal performance requirements. TextBlob makes

text processing simple by providing an intuitive interface to NLTK. TextBlob can be imported using the following command:

```
from textblob import TextBlob
```

spaCy

spaCy is an NLP library designed to be fast, streamlined, and production-ready. Its philosophy is to present only one algorithm (the best one) for each purpose. We don't have to make choices and can focus on being productive. spaCy uses its own pipeline to perform multiple preprocessing steps at the same time. We will demonstrate it in a subsequent section.

spaCy's models can be installed as Python packages, just like any other module. To load a model, use `spacy.load` with the model's shortcut link or package name or a path to the data directory:

```
import spacy
nlp = spacy.load("en_core_web_lg")
```

In addition to these, there are a few other libraries, such as gensim, that we will explore for some of the examples in this chapter.

Natural Language Processing: Theory and Concepts

As we have already established, NLP is a subfield of artificial intelligence concerned with programming computers to process textual data in order to gain useful insights. All NLP applications go through common sequential steps, which include some combination of preprocessing textual data and representing the text as predictive features before feeding them into a statistical inference algorithm. Figure 10-1 outlines the major steps in an NLP-based application.

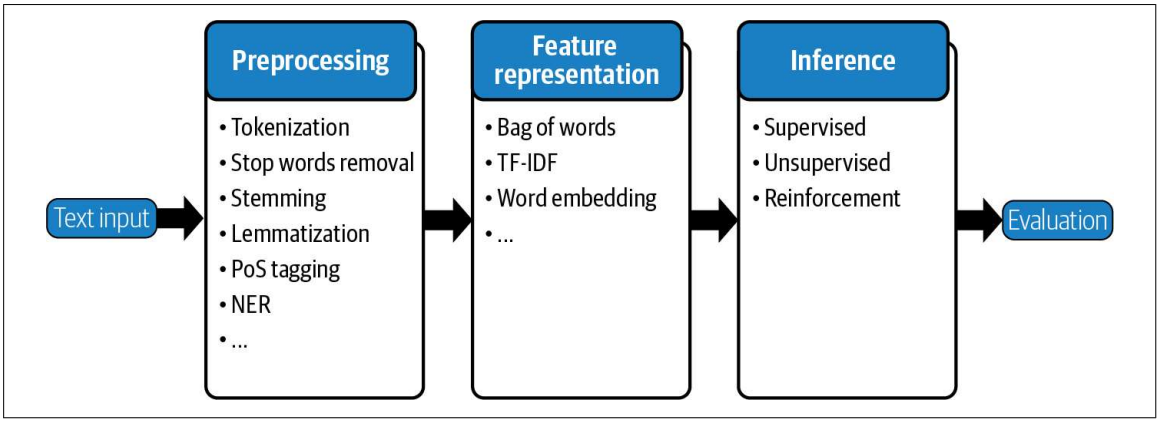


Figure 10-1. Natural language processing pipeline

The next section reviews these steps. For a thorough coverage of the topic, the reader is referred to *Natural Language Processing with Python* by Steven Bird, Ewan Klein, and Edward Loper (O'Reilly).

1. Preprocessing

There are usually multiple steps involved in preprocessing textual data for NLP. Figure 10-1 shows the key components of the preprocessing steps for NLP. These are tokenization, stop words removal, stemming, lemmatization, PoS (part-of-speech) tagging, and NER (Name Entity Recognition).

1.1. Tokenization

Tokenization is the task of splitting a text into meaningful segments, called tokens. These segments could be words, punctuation, numbers, or other special characters that are the building blocks of a sentence. A set of predetermined rules allows us to effectively convert a sentence into a list of tokens. The following code snippets show sample word tokenization using the NLTK and TextBlob packages:

```
#Text to tokenize
text = "This is a tokenize test"
```

The NLTK data package includes a pretrained `Punkt` tokenizer for English, which was previously loaded:

```
from nltk.tokenize import word_tokenize
word_tokenize(text)
```

`Output`

```
['This', 'is', 'a', 'tokenize', 'test']
```

Let's look at tokenization using TextBlob:

```
TextBlob(text).words
```

`Output`

```
WordList(['This', 'is', 'a', 'tokenize', 'test'])
```

1.2. Stop words removal

At times, extremely common words that offer little value in modeling are excluded from the vocabulary. These words are called stop words. The code for removing stop words using the NLTK library is shown below:

```
text = "S&P and NASDAQ are the two most popular indices in US"

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
stop_words = set(stopwords.words('english'))
text_tokens = word_tokenize(text)
```

```
tokens_without_sw= [word for word in text_tokens if not word in stop_words]

print(tokens_without_sw)
```

Output

```
['S', '&', 'P', 'NASDAQ', 'two', 'popular', 'indices', 'US']
```

We first load the language model and store it in the stop words variable. The `stop words.words('english')` is a set of default stop words for the English language model in NLTK. Next, we simply iterate through each word in the input text, and if the word exists in the stop word set of the NLTK language model, the word is removed. As we can see, stop words, such as *are* and *most*, are removed from the sentence.

1.3. Stemming

Stemming is the process of reducing inflected (or sometimes derived) words to their stem, base, or root form (generally a written word form). For example, if we were to stem the words *Stems*, *Stemming*, *Stemmed*, and *Stemitization*, the result would be a single word: *Stem*. The code for stemming using the NLTK library is shown here:

```
text = "It's a Stemming testing"

parsed_text = word_tokenize(text)

# Initialize stemmer.
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer('english')

# Stem each word.
[(word, stemmer.stem(word)) for i, word in enumerate(parsed_text)
 if word.lower() != stemmer.stem(parsed_text[i])]
```

Output

```
[('Stemming', 'stem'), ('testing', 'test')]
```

1.4. Lemmatization

A slight variant of stemming is *lemmatization*. The major difference between the two processes is that stemming can often create nonexistent words, whereas lemmas are actual words. An example of lemmatization is *run* as a base form for words like *running* and *ran*, or that the words *better* and *good* are considered the same lemma. The code for lemmatization using the TextBlob library is shown below:

```
text = "This world has a lot of faces "

from textblob import Word
parsed_data= TextBlob(text).words
```

```
[(word, word.lemmatize()) for i, word in enumerate(parsed_data)
 if word != parsed_data[i].lemmatize()]
```

Output

```
[('has', 'ha'), ('faces', 'face')]
```

1.5. PoS tagging

Part-of-speech (PoS) tagging is the process of assigning a token to its grammatical category (e.g., verb, noun, etc.) in order to understand its role within a sentence. PoS tags have been used for a variety of NLP tasks and are extremely useful since they provide a linguistic signal of how a word is being used within the scope of a phrase, sentence, or document.

After a sentence is split into tokens, a tagger, or PoS tagger, is used to assign each token to a part-of-speech category. Historically, hidden Markov models (HMM) were used to create such taggers. More recently, artificial neural networks have been leveraged. The code for PoS tagging using the TextBlob library is shown here:

```
text = 'Google is looking at buying U.K. startup for $1 billion'
TextBlob(text).tags
```

Output

```
[('Google', 'NNP'),
 ('is', 'VBZ'),
 ('looking', 'VBG'),
 ('at', 'IN'),
 ('buying', 'VBG'),
 ('U.K.', 'NNP'),
 ('startup', 'NN'),
 ('for', 'IN'),
 ('1', 'CD'),
 ('billion', 'CD')]
```

1.6. Named entity recognition

Named entity recognition (NER) is an optional next step in data preprocessing that seeks to locate and classify named entities in text into predefined categories. These categories can include names of persons, organizations, locations, expressions of times, quantities, monetary values, or percentages. The NER performed using spaCy is shown below:

```
text = 'Google is looking at buying U.K. startup for $1 billion'

for entity in nlp(text).ents:
    print("Entity: ", entity.text)
```

```
Output
```

```
Entity:  Google
Entity:  U.K.
Entity:  $1 billion
```

Visualizing named entities in text using the `displacy` module, as shown in Figure 10-2, can also be incredibly helpful in speeding up development and debugging the code and training process:

```
from spacy import displacy
displacy.render(nlp(text), style="ent", jupyter = True)
```

Figure 10-2. NER output

1.7. spaCy: All of the above steps in one go. All the preprocessing steps shown above can be performed in one step using spaCy. When we call *nlp* on a text, spaCy first tokenizes the text to produce a *Doc* object. The *Doc* is then processed in several different steps. This is also referred to as the *processing pipeline*. The pipeline used by the default models consists of a *tagger*, a *parser*, and an *entity recognizer*. Each pipeline component returns the processed *Doc*, which is then passed on to the next component, as demonstrated in Figure 10-3.

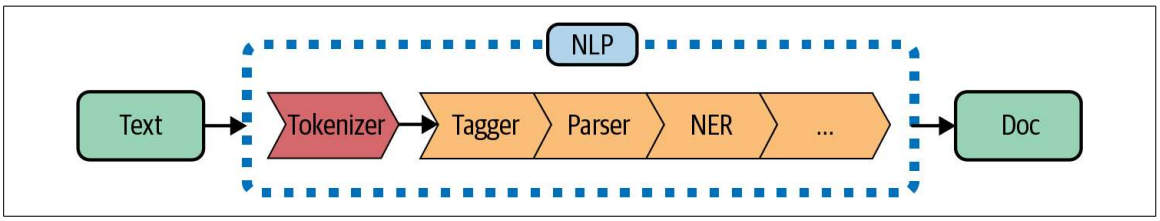


Figure 10-3. spaCy pipeline (based on an image from the spaCy website.

```
Python code text = 'Google is looking at buying U.K. startup for $1 billion'
doc = nlp(text)
pd.DataFrame([[t.text, t.is_stop, t.lemma_, t.pos_]
              for t in doc],
             columns=['Token', 'is_stop_word', 'lemma', 'POS'])
```

```
Output
```

	Token	is_stop_word	lemma	POS
0	Google	False	Google	PROPN
1	is	True	be	VERB
2	looking	False	look	VERB
3	at	True	at	ADP

	Token	is_stop_word	lemma	POS
4	buying	False	buy	VERB
5	U.K.	False	U.K.	PROPN
6	startup	False	startup	NOUN
7	for	True	for	ADP
8	$	False	$	SYM
9	1	False	1	NUM
10	billion	False	billion	NUM

The output for each of the preprocessing steps is shown in the preceding table. Given that spaCy performs a wide range of NLP-related tasks in a single step, it is a highly recommended package. As such, we will be using spaCy extensively in our case studies.

In addition to the above preprocessing steps, there are other frequently used preprocessing steps, such as *lower casing* or *nonalphanumeric data removing*, that we can perform depending on the type of data. For example, data scraped from a website has to be cleansed further, including the removal of HTML tags. Data from a PDF report must be converted into a text format.

Other optional preprocessing steps include dependency parsing, coreference resolution, triplet extraction, and relation extraction:

Dependency parsing
: Assigns a syntactic structure to sentences to make sense of how the words in the sentence relate to each other.

Coreference resolution
: The process of connecting tokens that represent the same entity. It is common in languages to introduce a subject with their name in one sentence and then refer to them as him/her/it in subsequent sentences.

Triplet extraction
: The process of recording subject, verb, and object triplets when available in the sentence structure.

Relation extraction
: A broader form of triplet extraction in which entities can have multiple interactions.

These additional steps should be performed only if they will help with the task at hand. We will demonstrate examples of these preprocessing steps in the case studies in this chapter.

2. Feature Representation

The vast majority of NLP-related data, such as news feed articles, PDF reports, social media posts, and audio files, is created for human consumption. As such, it is often stored in an unstructured format, which cannot be readily processed by computers. In order for the preprocessed information to be conveyed to the statistical inference algorithm, the tokens need to be translated into predictive features. A model is used to embed raw text into a *vector space*.

Feature representation involves two things:

- A vocabulary of known words.
- A measure of the presence of known words.

Some of the feature representation methods are:

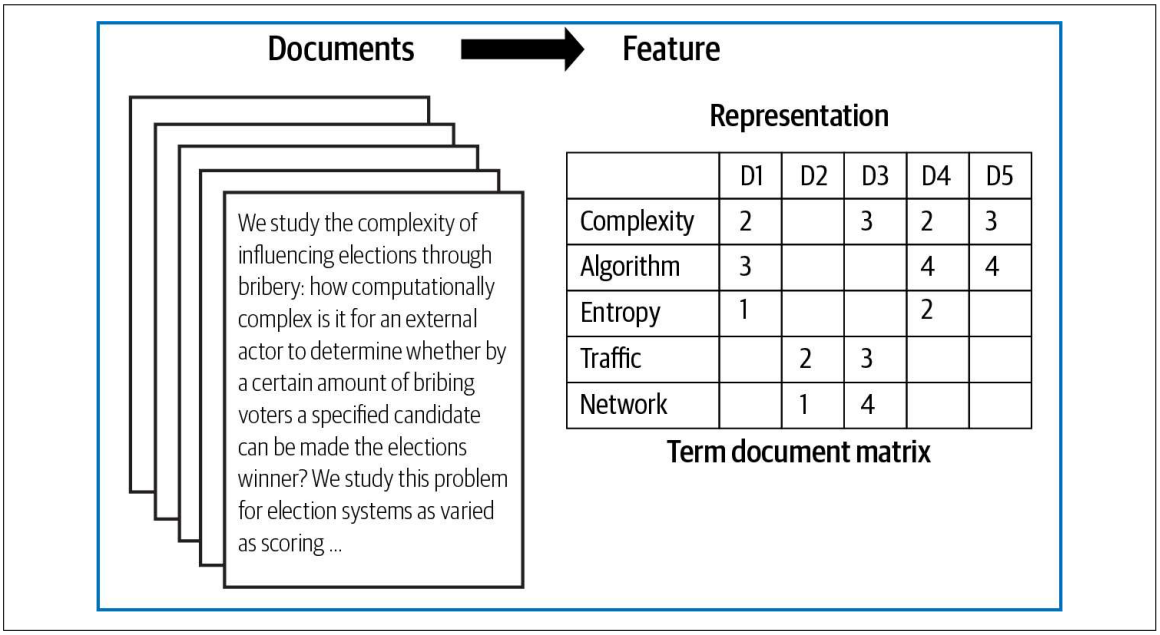
- Bag of words
- TF-IDF
- Word embedding
  - — Pretrained models (e.g., word2vec, GloVe, spaCy's word embedding model)
  - — Customized deep learning–based feature representation[1]

Let's learn more about each of these methods.

2.1. Bag of words—word count

In natural language processing, a common technique for extracting features from text is to place all words that occur in the text in a bucket. This approach is called a *bag of words* model. It's referred to as a bag of words because any information about the structure of the sentence is lost. In this technique, we build a single matrix from a collection of texts, as shown in Figure 10-4, in which each row represents a token and each column represents a document or sentence in our corpus. The values of the matrix represent the count of the number of instances of the token appearing.

1 A customized deep learning–based feature representation model is built in case study 1 of this chapter.



	D1	D2	D3	D4	D5
Complexity	2		3	2	3
Algorithm	3			4	4
Entropy	1			2	
Traffic		2	3		
Network		1	4		

Figure 10-4. Bag of words

The `CountVectorizer` from sklearn provides a simple way to both tokenize a collection of text documents and encode new documents using that vocabulary. The `fit_transform` function learns the vocabulary from one or more documents and encodes each document in the word as a vector:

```
sentences = [
'The stock price of google jumps on the earning data today',
'Google plunge on China Data!'
]
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
print( vectorizer.fit_transform(sentences).todense() )
print( vectorizer.vocabulary_ )
```

`Output`

```
[[0 1 1 1 1 1 1 0 1 1 2 1]
 [1 1 0 1 0 0 1 1 0 0 0 0]]
{'the': 10, 'stock': 9, 'price': 8, 'of': 5, 'google': 3, 'jumps':\
 4, 'on': 6, 'earning': 2, 'data': 1, 'today': 11, 'plunge': 7,\
 'china': 0}
```

We can see an array version of the encoded vector showing a count of one occurrence for each word except *the* (index 10), which has an occurrence of two. Word counts are a good starting point, but they are very basic. One issue with simple counts is that some words like *the* will appear many times, and their large counts will not be very meaningful in the encoded vectors. These bag of words representations are sparse because the vocabularies are vast, and a given word or document would be represented by a large vector comprised mostly of zero values.

2.2. TF-IDF

An alternative is to calculate word frequencies, and by far the most popular method for that is *TF-IDF*, which stands for *Term Frequency–Inverse Document Frequency*:

Term Frequency
: This summarizes how often a given word appears within a document.

Inverse Document Frequency
: This downscales words that appear a lot across documents.

Put simply, TF-IDF is a word frequency score that tries to highlight words that are more interesting (i.e., frequent *within* a document, but not *across* documents). The *TfidfVectorizer* will tokenize documents, learn the vocabulary and the inverse document frequency weightings, and allow you to encode new documents:

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(max_features=1000, stop_words='english')
TFIDF = vectorizer.fit_transform(sentences)
print(vectorizer.get_feature_names()[-10:])
print(TFIDF.shape)
print(TFIDF.toarray())
```

`Output`

```
['china', 'data', 'earning', 'google', 'jumps', 'plunge', 'price', 'stock', \
'today']
(2, 9)
[[0.         0.29017021 0.4078241  0.29017021 0.4078241  0.
  0.4078241  0.4078241  0.4078241 ]
 [0.57615236 0.40993715 0.         0.40993715 0.         0.57615236
  0.         0.         0.        ]]
```

In the provided code snippet, a vocabulary of nine words is learned from the documents. Each word is assigned a unique integer index in the output vector. The sentences are encoded as a nine-element sparse array, and we can review the final scorings of each word with different values from the other words in the vocabulary.

2.3. Word embedding

A *word embedding* represents words and documents using a dense vector representation. In an embedding, words are represented by dense vectors in which a vector represents the projection of the word into a continuous vector space. The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used. The position of a word in the learned vector space is referred to as its *embedding*.

Some of the models of learning word embeddings from text include word2Vec, spaCy's pretrained word embedding model, and GloVe. In addition to these carefully designed methods, a word embedding can be learned as part of a deep learning

model. This can be a slower approach, but it tailors the model to a specific training dataset.

2.3.1. Pretrained model: Via spaCy

spaCy comes with built-in representation of text as vectors at different levels of word, sentence, and document. The underlying vector representations come from a word embedding model, which generally produces a dense, multidimensional semantic representation of words (as shown in the following example). The word embedding model includes 20,000 unique vectors with 300 dimensions. Using this vector representation, we can calculate similarities and dissimilarities between tokens, named entities, noun phrases, sentences, and documents.

The word embedding in spaCy is performed by first loading the model and then processing text. The vectors can be accessed directly using the `.vector` attribute of each processed token (i.e., word). The mean vector for the entire sentence is also calculated simply by using the vector, providing a very convenient input for machine learning models based on sentences:

```
doc = nlp("Apple orange cats dogs")
print("Vector representation of the sentence for first 10 features: \n", \
doc.vector[0:10])
```

```
Output:\

Vector representation of the sentence for first 10 features:
 [ -0.30732775 0.22351399 -0.110111   -0.367025   -0.13430001
   0.13790375 -0.24379876 -0.10736975  0.2715925   1.3117325 ]
```

The vector representation of the sentence for the first 10 features of the pretrained model is shown in the output.

2.3.2. Pretrained model: Word2Vec using gensim package

The Python-based implementation of the word2vec model using the gensim package is demonstrated here:

```
from gensim.models import Word2Vec

sentences = [
['The','stock','price', 'of', 'Google', 'increases'],
['Google','plunge',' on','China',' Data!']]

# train model
model = Word2Vec(sentences, min_count=1)

# summarize the loaded model
words = list(model.wv.vocab)
print(words)
print(model['Google'][1:5])
```

```
Output

    ['The', 'stock', 'price', 'of', 'Google', 'increases', 'plunge', ' on', 'China',\
    ' Data!']
    [-1.7868265e-03 -7.6242397e-04  6.0105987e-05  3.5568199e-03
    ]
```

The vector representation of the sentence for the first five features of the pretrained word2vec model is shown above.

3. Inference

As with other artificial intelligence tasks, an inference generated by an NLP application usually needs to be translated into a decision in order to be actionable. Inference falls under three machine learning categories covered in the previous chapters (i.e., supervised, unsupervised, and reinforcement learning). While the type of inference required depends on the business problem and the type of training data, the most commonly used algorithms are supervised and unsupervised.

One of the most frequently used supervised methodologies in NLP is the *Naive Bayes* model, as it can produce reasonable accuracy using simple assumptions. A more complex supervised methodology is using artificial neural network architectures. In past years, these architectures, such as recurrent neural networks (RNNs), have dominated NLP-based inference.

Most of the existing literature in NLP focuses on supervised learning. As such, unsupervised learning applications constitute a relatively less developed subdomain in which measuring *document similarity* is among the most common tasks. A popular unsupervised technique applied in NLP is *Latent Semantic Analysis* (LSA). LSA looks at relationships between a set of documents and the words they contain by producing a set of latent concepts related to the documents and terms. LSA has paved the way for a more sophisticated approach called *Latent Dirichlet Allocation* (LDA), under which documents are modeled as a finite mixture of topics. These topics in turn are modeled as a finite mixture over words in the vocabulary. LDA has been extensively used for *topic modeling*—a growing area of research in which NLP practitioners build probabilistic generative models to reveal likely topic attributions for words.

Since we have reviewed many supervised and unsupervised learning models in the previous chapters, we will provide details only on Naive Bayes and LDA models in the next sections. These are used extensively in NLP and were not covered in the previous chapters.

3.1. Supervised learning example—Naive Bayes

Naive Bayes is a family of algorithms based on applying *Bayes's theorem* with a strong (naive) assumption that every feature used to predict the category of a given sample is independent of the others. They are probabilistic classifiers and therefore will

calculate the probability of each category using Bayes's theorem. The category with the highest probability will be output.

In NLP, a Naive Bayes approach assumes that all word features are independent of each other given the class labels. Due to this simplifying assumption, Naive Bayes is very compatible with a bag-of-words word representation, and it has been demonstrated to be fast, reliable, and accurate in a number of NLP applications. Moreover, despite its simplifying assumptions, it is competitive with (and at times even outperforms) more complicated classifiers.

Let us look at the usage of Naive Bayes for the inference in a sentiment analysis problem. We take a dataframe in which there are two sentences with sentiments assigned to each. In the next step, we convert the sentences into a feature representation using `CountVectorizer`. The features and sentiments are used to train and test the model using Naive Bayes:

```
sentences = [
'The stock price of google jumps on the earning data today',
'Google plunge on China Data!']
sentiment = (1, 0)
data = pd.DataFrame({'Sentence':sentences,
        'sentiment':sentiment})

# feature extraction
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer().fit(data['Sentence'])
X_train_vectorized = vect.transform(data['Sentence'])

# Running naive bayes model
from sklearn.naive_bayes import MultinomialNB
clfrNB = MultinomialNB(alpha=0.1)
clfrNB.fit(X_train_vectorized, data['sentiment'])

#Testing the model
preds = clfrNB.predict(vect.transform(['Apple price plunge',\
 'Amazon price jumps']))
preds
```

`Output`

```
array([0, 1])
```

As we can see, the Naive Bayes trains the model fairly well from the two sentences. The model gives a sentiment of zero and one for the test sentences "Apple price plunge" and "Amazon price jumps," respectively, given the sentences used for training also had the keywords "plunge" and "jumps," with corresponding sentiment assignments.

3.2. Unsupervised learning example: LDA

LDA is extensively used for *topic modeling* because it tends to produce meaningful topics that humans can interpret, assigns topics to new documents, and is extensible. It works by first making a key assumption: documents are generated by first selecting *topics*, and then, for each topic, a set of *words*. The algorithm then reverse engineers this process to find the topics in a document.

In the following code snippet, we show an implementation of LDA for topic modeling. We take two sentences and convert the sentences into a feature representation using `CountVectorizer`. These features and the sentiments are used to train the model and produce two smaller matrices representing the topics:

```
sentences = [
'The stock price of google jumps on the earning data today',
'Google plunge on China Data!'
]

#Getting the bag of words
from sklearn.decomposition import LatentDirichletAllocation
vect=CountVectorizer(ngram_range=(1, 1),stop_words='english')
sentences_vec=vect.fit_transform(sentences)

#Running LDA on the bag of words.
from sklearn.feature_extraction.text import CountVectorizer
lda=LatentDirichletAllocation(n_components=3)
lda.fit_transform(sentences_vec)
```

`Output`

```
array([[0.04283242, 0.91209846, 0.04506912],
       [0.06793339, 0.07059533, 0.86147128]])
```

We will be using LDA for topic modeling in the third case study of this chapter and will discuss the concepts and interpretation in detail.

To review, in order to approach any NLP-based problem, we need to follow the preprocessing, feature extraction, and inference steps. Now, let's dive into the case studies.

Case Study 1: NLP and Sentiment Analysis–Based Trading Strategies

Natural language processing offers the ability to quantify text. One can begin to ask questions such as: How positive or negative is this news? and How can we quantify words?

Perhaps the most notable application of NLP is its use in algorithmic trading. NLP provides an efficient means of monitoring market sentiments. By applying

NLP-based sentiment analysis techniques to news articles, reports, social media, or other web content, one can effectively determine whether those sources have a positive or negative senitment score. Sentiment scores can be used as a directional signal to buy stocks with positive scores and sell stocks with negative ones.

Trading strategies based on text data are becoming more popular as the amount of unstructured data increases. In this case study we are going to look at how one can use NLP-based sentiments to build a trading strategy.

In this case study, we will focus on:

- Producing news sentiments using supervised and unsupervised algorithms.
- Enhancing sentiment analysis by using a deep learning model, such as LSTM.
- Comparison of different sentiment generation methodologies for the purpose of building a trading strategy.
- Using sentiments and word vectors effectively as features in a trading strategy.
- Collecting data from different sources and preprocessing it for sentiment analysis.
- Using NLP Python packages for sentiment analysis.
- Building a framework for backtesting results of a trading strategy using available Python packages.

This case study combines concepts presented in previous chapters. The overall model development steps of this case study are similar to the seven-step model development in prior case studies, with slight modifications.

Blueprint for Building a Trading Strategy Based on Sentiment Analysis

1. Problem definition

Our goal is to (1) use NLP to extract information from news headlines, (2) assign a sentiment to that information, and (3) use sentiment analysis to build a trading strategy.

The data used for this case study will be from the following sources:

News headlines data compiled from the RSS feeds of several news websites
: For the purpose of this study, we will look only at the headlines, not at the full text of the stories. Our dataset contains around 82,000 headlines from May 2011 through December 2018.[2]

Yahoo Finance website for stock data
: The return data for stocks used in this case study is derived from Yahoo Finance price data.

Kaggle
: We will use the labeled data of news sentiments for a classification-based sentiment analysis model. Note that this data may not be fully applicable to the case at hand and is used here for demonstration purposes.

Stock market lexicon
: *Lexicon* refers to the component of an NLP system that contains information (semantic, grammatical) about individual words or word strings. This is created based on stock market conversations in microblogging services.[3]

The key steps of this case study are outlined in Figure 10-5.

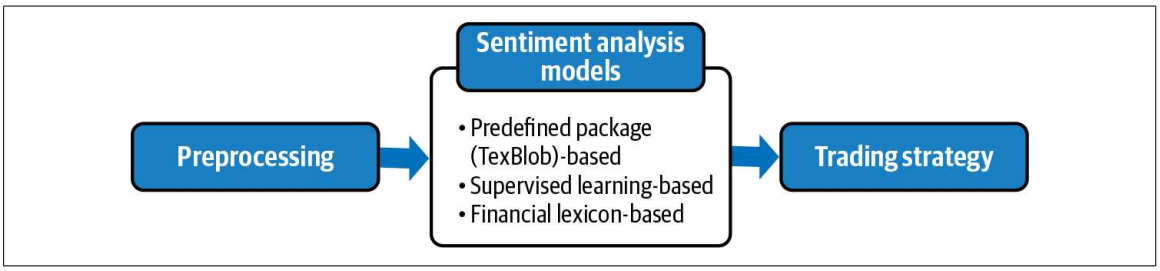


Figure 10-5. Steps in a sentiment analysis–based trading strategy

Once we are done with preprocessing, we will look at the different sentiment analysis models. The results from the sentiment analysis step are used to develop the trading strategy.

2 The news can be downloaded by a simple web-scraping program in Python using packages such as Beautiful Soup. Readers should talk to the website or follow its terms of service in order to use the news for commercial purpose.

3 The source of this lexicon is Nuno Oliveira, Paulo Cortez, and Nelson Areal, "Stock Market Sentiment Lexicon Acquisition Using Microblogging Data and Statistical Measures," *Decision Support Systems* 85 (March 2016): 62–73.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The first set of libraries to be loaded are the NLP-specific libraries discussed above. Refer to the Jupyter notebook of this case study for details of the other libraries.

```
from textblob import TextBlob
import spacy
import nltk
import warnings
from nltk.sentiment.vader import SentimentIntensityAnalyzer
nltk.download('vader_lexicon')
nlp = spacy.load("en_core_web_lg")
```

2.2. Loading the data. In this step, we load the stock price data from Yahoo Finance. We select 10 stocks for this case study. These stocks are some of the largest stocks in the S&P 500 by market share:

```
tickers = ['AAPL','MSFT','AMZN','GOOG','FB','WMT','JPM','TSLA','NFLX','ADBE']
start = '2010-01-01'
end = '2018-12-31'
df_ticker_return = pd.DataFrame()
for ticker in tickers:
    ticker_yf = yf.Ticker(ticker)
    if df_ticker_return.empty:
        df_ticker_return = ticker_yf.history(start = start, end = end)
        df_ticker_return['ticker']= ticker
    else:
        data_temp = ticker_yf.history(start = start, end = end)
        data_temp['ticker']= ticker
        df_ticker_return = df_ticker_return.append(data_temp)
df_ticker_return.to_csv(r'Data\Step3.2_ReturnData.csv')
```

```
df_ticker_return.head(2)
```

Date	Open	High	Low	Close	Volume	Dividends	Stock Splits	ticker
2010-01-04	26.40	26.53	26.27	26.47	123432400	0.0	0.0	AAPL
2010-01-05	26.54	26.66	26.37	26.51	150476200	0.0	0.0	AAPL

The data contains the price and volume data of the stocks along with their ticker name. In the next step, we look at the news data.

3. Data preparation

In this step, we load and preprocess the news data, followed by combining the news data with the stock return data. This combined dataset will be used for the model development.

3.1. Preprocessing news data. The news data is downloaded from the News RSS feed, and the file is available in JSON format. The JSON files for different dates are kept under a zipped folder. The data is downloaded using the standard web-scraping Python package Beautiful Soup, which is an open source framework. Let us look at the content of the downloaded JSON file:

```
z = zipfile.ZipFile("Data/Raw Headline Data.zip", "r")
testFile=z.namelist()[10]
fileData= z.open(testFile).read()
fileDataSample = json.loads(fileData)['content'][1:500]
fileDataSample
```

`Output`

```
'li class="n-box-item date-title" data-end="1305172799" data-start="1305086400"
data-txt="Tuesday, December 17, 2019">Wednesday, May 11,2011</li><li
class="n-box-item sa-box-item" data-id="76179" data-ts="1305149244"><div
class="media media-overflow-fix"><div class-"media-left"><a class="box-ticker"
href="/symbol/CSCO" target="blank">CSCO</a></div><div class="media-body"<h4
class="media-heading"><a href="/news/76179" sasource="on_the_move_news_
fidelity" target="_blank">Cisco (NASDAQ:CSCO): Pr'
```

We can see that the JSON format is not suitable for the algorithm. We need to get the news from the JSONs. Regex becomes the vital part of this step. Regex can find a pattern in the raw, messy text and perform actions accordingly. The following function parses HTML by using information encoded in the JSON file:

```
def jsonParser(json_data):
    xml_data = json_data['content']

    tree = etree.parse(StringIO(xml_data), parser=etree.HTMLParser())

    headlines = tree.xpath("//h4[contains(@class, 'media-heading')]/a/text()")
    assert len(headlines) == json_data['count']

    main_tickers = list(map(lambda x: x.replace('/symbol/', ''),\
           tree.xpath("//div[contains(@class, 'media-left')]//a/@href")))
    assert len(main_tickers) == json_data['count']
    final_headlines = [''.join(f.xpath('.//text()')) for f in\
           tree.xpath("//div[contains(@class, 'media-body')]/ul/li[1]")]
    if len(final_headlines) == 0:
        final_headlines = [''.join(f.xpath('.//text()')) for f in\
           tree.xpath("//div[contains(@class, 'media-body')]")]
        final_headlines = [f.replace(h, '').split('\xa0')[0].strip()\
                           for f,h in zip (final_headlines, headlines)]
    return main_tickers, final_headlines
```

Let us see how the output looks like after running the JSON parser:

```
jsonParser(json.loads(fileData))[1][1]
```

Output

```
'Cisco Systems (NASDAQ:CSCO) falls further into the red on FQ4
 guidance of $0.37-0.39 vs. $0.42 Street consensus. Sales seen flat
 to +2% vs. 8% Street view. CSCO recently -2.1%.'
```

As we can see, the output is converted into a more readable format after JSON parsing.

While evaluating the sentiment analysis models, we also analyze the relationship between the sentiments and subsequent stock performance. In order to understand the relationship, we use *event return*, which is the return that corresponds to the event. We do this because at times the news is reported late (i.e., after market participants are aware of the announcement) or after market close. Having a slightly wider window ensures that we capture the essence of the event. *Event return* is defined as:

$$R_{t-1} + R_t + R_{t+1}$$

where R_{t-1}, R_{t+1} are the returns before and after the news data, and R_t is the return on the day of the news (i.e., time t).

Let us extract the event return from the data:

```
#Computing the return
df_ticker_return['ret_curr'] = df_ticker_return['Close'].pct_change()
#Computing the event return
df_ticker_return['eventRet'] = df_ticker_return['ret_curr']\
 + df_ticker_return['ret_curr'].shift(-1) + df_ticker_return['ret_curr'].shift(1)
```

Now we have all the data in place. We will prepare a combined dataframe, which will have the news headlines mapped to the date, the returns (event return, current return, and next day's return), and stock ticker. This dataframe will be used for building the sentiment analysis model and the trading strategy:

```
combinedDataFrame = pd.merge(data_df_news, df_ticker_return, how='left', \
left_on=['date','ticker'], right_on=['date','ticker'])
combinedDataFrame = combinedDataFrame[combinedDataFrame['ticker'].isin(tickers)]
data_df = combinedDataFrame[['ticker','headline','date','eventRet','Close']]
data_df = data_df.dropna()
data_df.head(2)
```

Output

	ticker	headline	date	eventRet	Close
5	AMZN	Whole Foods (WFMI) –5.2% following a downgrade…	2011-05-02	0.017650	201.19
11	NFLX	Netflix (NFLX +1.1%) shares post early gains a…	2011-05-02	–0.013003	33.88

Let us look at the overall shape of the data:

```
print(data_df.shape, data_df.ticker.unique().shape)
```

```
Output
```

```
(2759, 5) (10,)
```

In this step, we prepared a clean dataframe that has ticker, headline, event return, return for a given day, and future return for 10 stock tickers, totaling 2,759 rows of data. Let us evaluate the models for sentiment analysis in the next step.

4. Evaluate models for sentiment analysis

In this section, we will go through the following three approaches of computing sentiments for the news:

- Predefined model—TextBlob package
- Tuned model—classification algorithms and LSTM
- Model based on financial lexicon

Let us go through the steps.

4.1. Predefined model—TextBlob package. The `TextBlob sentiment` function is a pretrained model based on the Naive Bayes classification algorithm. The function maps adjectives that are frequently found in movie reviews[4] to sentiment polarity scores ranging from –1 to +1 (negative to positive), converting a sentence to a numerical value. We apply this on all headline articles. An example of getting the sentiment for a news text is shown below:

```
text = "Bayer (OTCPK:BAYRY) started the week up 3.5% to €74/share in Frankfurt, \
touching their
highest level in 14 months, after the U.S. government said \
 a $25M glyphosate decision against the
company should be reversed."

TextBlob(text).sentiment.polarity
```

```
Output
```

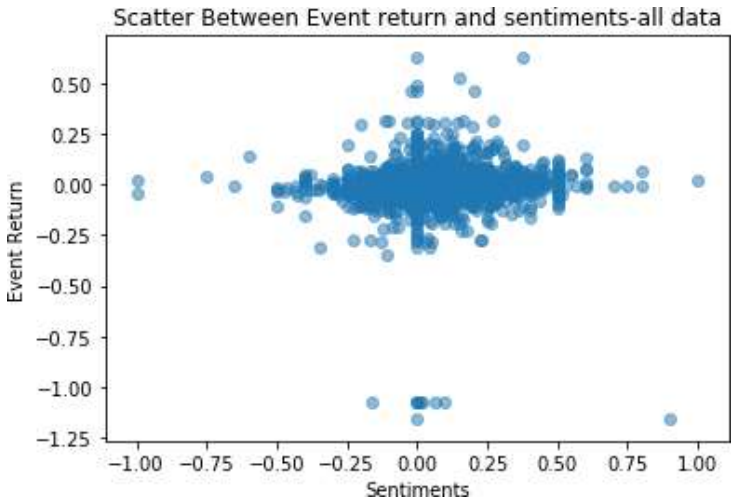
```
0.5
```

4 We also train a sentiment analysis model on the financial data in the subsequent section and compare the results against the TextBlob model.

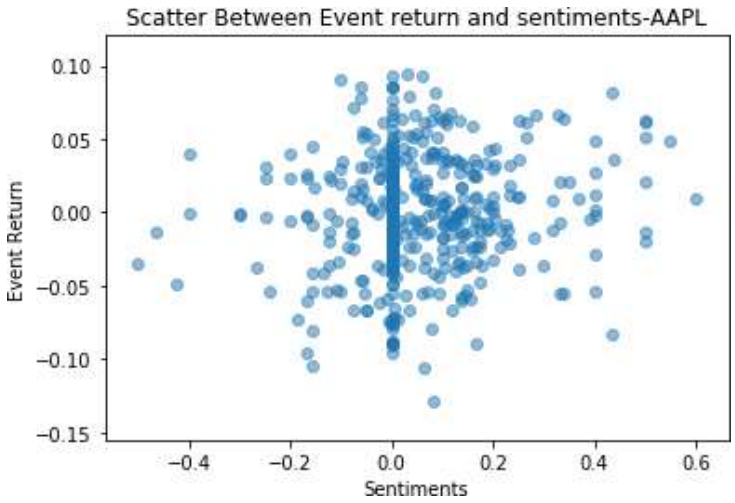
The sentiment for the statement is 0.5. We apply this on all headlines we have in the data:

```
data_df['sentiment_textblob'] = [TextBlob(s).sentiment.polarity for s in \
data_df['headline']]
```

Let us inspect the scatterplot of the sentiments and returns to examine the correlation between the two for all 10 stocks.



A plot for a single stock (APPL) is also shown in the following chart (see the code in the Jupyter notebook in the GitHub repository for this book for more details on the code):



From the scatterplots, we can see that there is not a strong relationship between the news and the sentiments. The correlation between return and sentiments is positive (4.27%), which means that news with positive sentiments leads to positive return and is expected. However, the correlation is not very high. Even looking at the overall

scatterplot, we see the majority of the sentiments concentrated around zero. This raises the question of whether a sentiment score trained on movie reviews is appropriate for stock prices. The `sentiment_assessments` attribute lists the underlying values for each token and can help us understand the reason for the overall sentiment of a sentence:

```
text = "Bayer (OTCPK:BAYRY) started the week up 3.5% to €74/share\
in Frankfurt, touching their highest level in 14 months, after the\
U.S. government said a $25M glyphosate decision against the company\
should be reversed."
TextBlob(text).sentiment_assessments
```

`Output`

```
Sentiment(polarity=0.5, subjectivity=0.5, assessments=[(['touching'], 0.5, 0.5, \
None)])
```

We see that the statement has a positive sentiment of 0.5, but it appears the word "touching" gave rise to the positive sentiment. More intuitive words, such as "high," do not. This example shows that the context of the training data is important for the sentiment score to be meaningful. There are many predefined packages and functions available for sentiment analysis, but it is important to be careful and have a thorough understanding of the problem's context before using a function or an algorithm for sentiment analysis.

For this case study, we may need sentiments trained on the financial news. Let us take a look at that in the next step.

4.2. Supervised learning—classification algorithms and LSTM

In this step, we develop a customized model for sentiment analysis based on available labeled data. The label data for this is obtained from the Kaggle website:

```
sentiments_data = pd.read_csv(r'Data\LabelledNewsData.csv', \
encoding="ISO-8859-1")
sentiments_data.head(1)
```

`Output`

	datetime	headline	ticker	sentiment
0	1/16/2020 5:25	$MMM fell on hard times but could be set to re…	MMM	0
1	1/11/2020 6:43	Wolfe Research Upgrades 3M $MMM to ¡§Peer Perf…	MMM	1

The data has headlines for the news across 30 different stocks, totaling 9,470 rows, and has sentiments labeled zero and one. We perform the classification steps using the classification model development template presented in Chapter 6.

In order to run a supervised learning model, we first need to convert the news headlines into a feature representation. For this exercise, the underlying vector

representations come from a *spaCy word embedding model*, which generally produces a dense, multidimensional semantic representation of words (as shown in the example below). The word embedding model includes 20,000 unique vectors with 300 dimensions. We apply this on all headlines in the data processed in the previous step:

```
all_vectors = pd.np.array([pd.np.array([token.vector for token in nlp(s) ]).\
mean(axis=0)*pd.np.ones((300))\
 for s in sentiments_data['headline']])
```

Now that we have prepared the independent variable, we train the classification model in a similar manner as discussed in Chapter 6. We have the sentiments label zero or one as the dependent variable. We first divide the data into training and test sets and run the key classification models (i.e., logistic regression, CART, SVM, random forest, and artificial neural network).

We will also include LSTM, which is an RNN-based model,[5] in the list of models considered. An RNN-based model performs well for NLP, because it stores the information for current features as well neighboring ones for prediction. It maintains a memory based on past information, which enables the model to predict the current output conditioned on long distance features and looks at the words in the context of the entire sentence, rather than simply looking at the individual words.

For us to be able to feed the data into our LSTM model, all input documents must have the same length. We use the Keras `tokenizer` function to tokenize the strings and then use `texts_to_sequences` to make sequences of words. More details can be found on the Keras website. We will limit the maximum review length to *max_words* by truncating longer reviews and pad shorter reviews with a null value (0). We can accomplish this using the `pad_sequences` function, also in Keras. The third parameter is the *input_length* (set to 50), which is the length of each comment sequence:

```
### Create sequence
vocabulary_size = 20000
tokenizer = Tokenizer(num_words= vocabulary_size)
tokenizer.fit_on_texts(sentiments_data['headline'])
sequences = tokenizer.texts_to_sequences(sentiments_data['headline'])
X_LSTM = pad_sequences(sequences, maxlen=50)
```

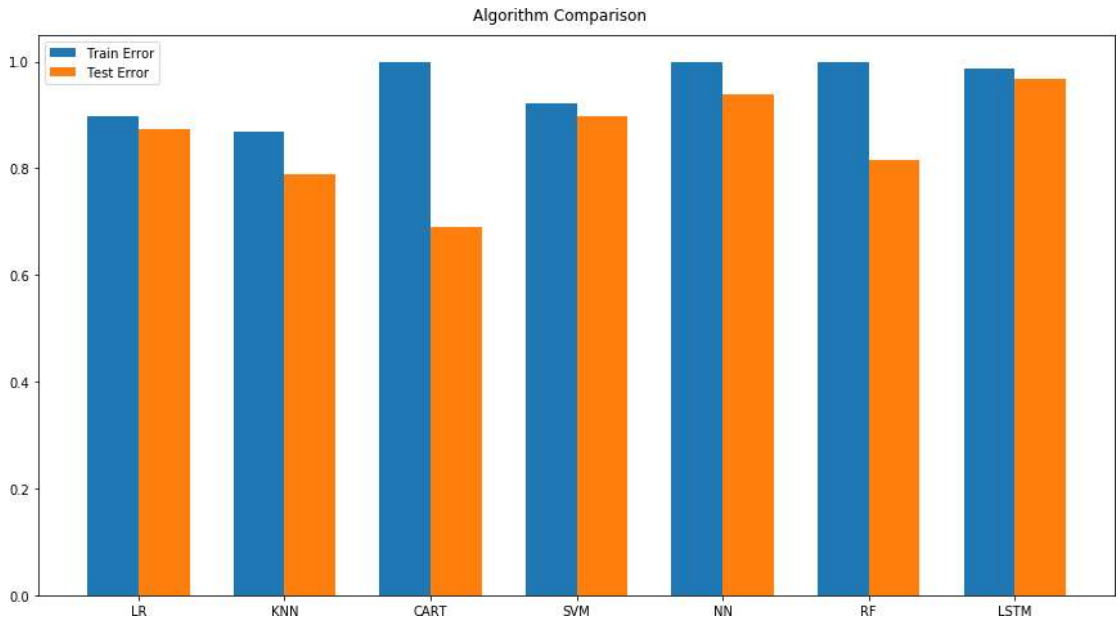
In the following code snippet, we use the Keras library to build an artificial neural network classifier based on an underlying LSTM model. The network starts with an *embedding* layer. This layer lets the system expand each token to a larger vector, allowing the network to represent a word in a meaningful way. The layer takes 20,000 as the first argument (i.e., the size of our vocabulary) and 300 as the second input parameter (i.e., the dimension of the embedding). Finally, given that this is a classification problem and the output needs to be labeled as zero or one, the

5 Refer to Chapter 5 for more details on RNN models.

`KerasClassifier` function is used as a wrapper over the LSTM model to produce a binary (zero or one) output:

```
from keras.wrappers.scikit_learn import KerasClassifier
def create_model(input_length=50):
    model = Sequential()
    model.add(Embedding(20000, 300, input_length=50))
    model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
    metrics=['accuracy'])
    return model
model_LSTM = KerasClassifier(build_fn=create_model, epochs=3, verbose=1, \
  validation_split=0.4)
model_LSTM.fit(X_train_LSTM, Y_train_LSTM)
```

The comparison of all the machine learning models is as follows:



As expected, the LSTM model has the best performance in the test set (accuracy of 96.7%) as compared to all other models. The performance of the ANN, with a training set accuracy of 99% and a test set accuracy of 93.8%, is comparable to the LSTM-based model. The performances of random forest (RF), SVM, and logistic regression (LR) are reasonable as well. CART and KNN do not perform as well as other models. CART shows high overfitting. Let us use the LSTM model for the computation of the sentiments in the data in the following steps.

4.3. Unsupervised—model based on a financial lexicon

In this case study, we update the VADER lexicon with words and sentiments from a lexicon adapted to stock market conversations in microblogging services:

Lexicons
: Special dictionaries or vocabularies that have been created for analyzing sentiments. Most lexicons have a list of positive and negative *polar* words with some score associated with them. Using various techniques, such as the position of words, the surrounding words, context, parts of speech, and phrases, scores are assigned to the text documents for which we want to compute the sentiment. After aggregating these scores, we get the final sentiment:

VADER (Valence Aware Dictionary for Sentiment Reasoning)
: A prebuilt sentiment analysis model included in the NLTK package. It can give both positive and negative polarity scores as well as the strength of the emotion of a text sample. It is rule-based and relies heavily on human-rated texts. These are words or any textual form of communication labeled according to their semantic orientation as either positive or negative.

This lexical resource was automatically created using diverse statistical measures and a large set of labeled messages from StockTwits, which is a social media platform designed for sharing ideas among investors, traders, and entrepreneurs.[6] The sentiments are between –1 and 1, similar to the sentiments from TextBlob. In the following code snippet, we train the model based on the financial sentiments:

```
# stock market lexicon
sia = SentimentIntensityAnalyzer()
stock_lex = pd.read_csv('Data/lexicon_data/stock_lex.csv')
stock_lex['sentiment'] = (stock_lex['Aff_Score'] + stock_lex['Neg_Score'])/2
stock_lex = dict(zip(stock_lex.Item, stock_lex.sentiment))
stock_lex = {k:v for k,v in stock_lex.items() if len(k.split(' '))==1}
stock_lex_scaled = {}
for k, v in stock_lex.items():
    if v > 0:
        stock_lex_scaled[k] = v / max(stock_lex.values()) * 4
    else:
        stock_lex_scaled[k] = v / min(stock_lex.values()) * -4

final_lex = {}
final_lex.update(stock_lex_scaled)
final_lex.update(sia.lexicon)
sia.lexicon = final_lex
```

6 The source of this lexicon is Nuno Oliveira, Paulo Cortez, and Nelson Areal, "Stock Market Sentiment Lexicon Acquisition Using Microblogging Data and Statistical Measures," *Decision Support Systems* 85 (March 2016): 62–73.

Let us check the sentiment of a news item:

```
text = "AAPL is trading higher after reporting its October sales\
rose 12.6% M/M. It has seen a 20%+ jump in orders"

sia.polarity_scores(text)['compound']
```

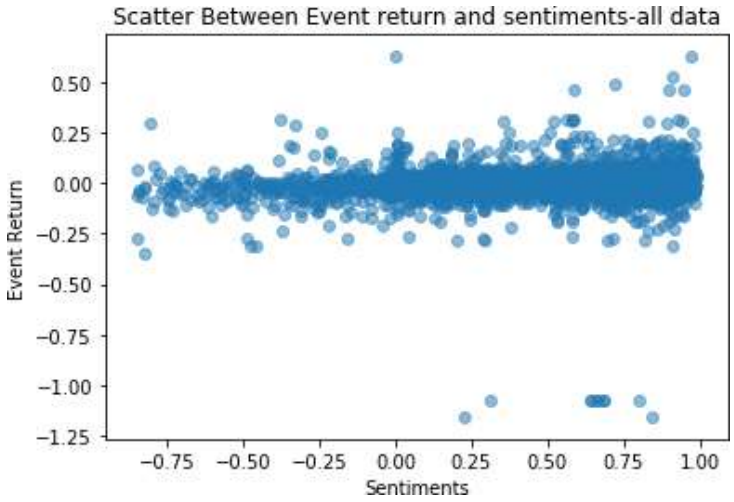
Output

```
0.4535
```

We get the sentiments for all the news headlines based in our dataset:

```
vader_sentiments = pd.np.array([sia.polarity_scores(s)['compound']\
 for s in data_df['headline']])
```

Let us look at the relationship between the returns and sentiments, which is computed using the lexicon-based methodology for the entire dataset.



There are not many instances of high returns for lower sentiment scores, but the data may not be very clear. We will look deeper into the comparison of different types of sentiment analysis in the next section.

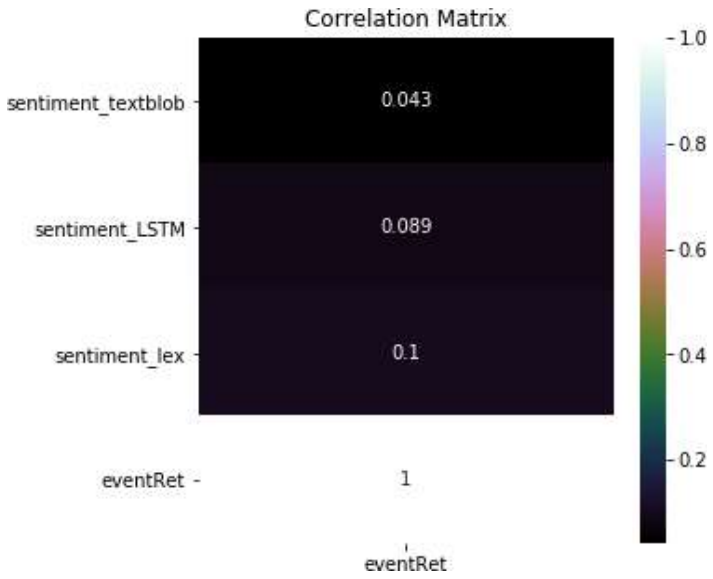
4.4. Exploratory data analysis and comparison

In this section, we compare the sentiments computed using the different techniques presented above. Let us look at the sample headlines and the sentiments from three different methodologies, followed by a visual analysis:

	ticker	headline	sentiment_textblob	sentiment_LSTM	sentiment_lex
4620	TSM	TSMC (TSM +1.8%) is trading higher after reporting its October sales rose 12.6% M/M. DigiTimes adds TSMC has seen a 20%+ jump in orders from QCOM, NVDA, SPRD, and Mediatek. The numbers suggest TSMC could beat its Q4 guidance (though December tends to be weak), and that chip demand could be stabilizing after getting hit hard by inventory corrections. (earlier) (UMC sales)	0.036667	1	0.5478

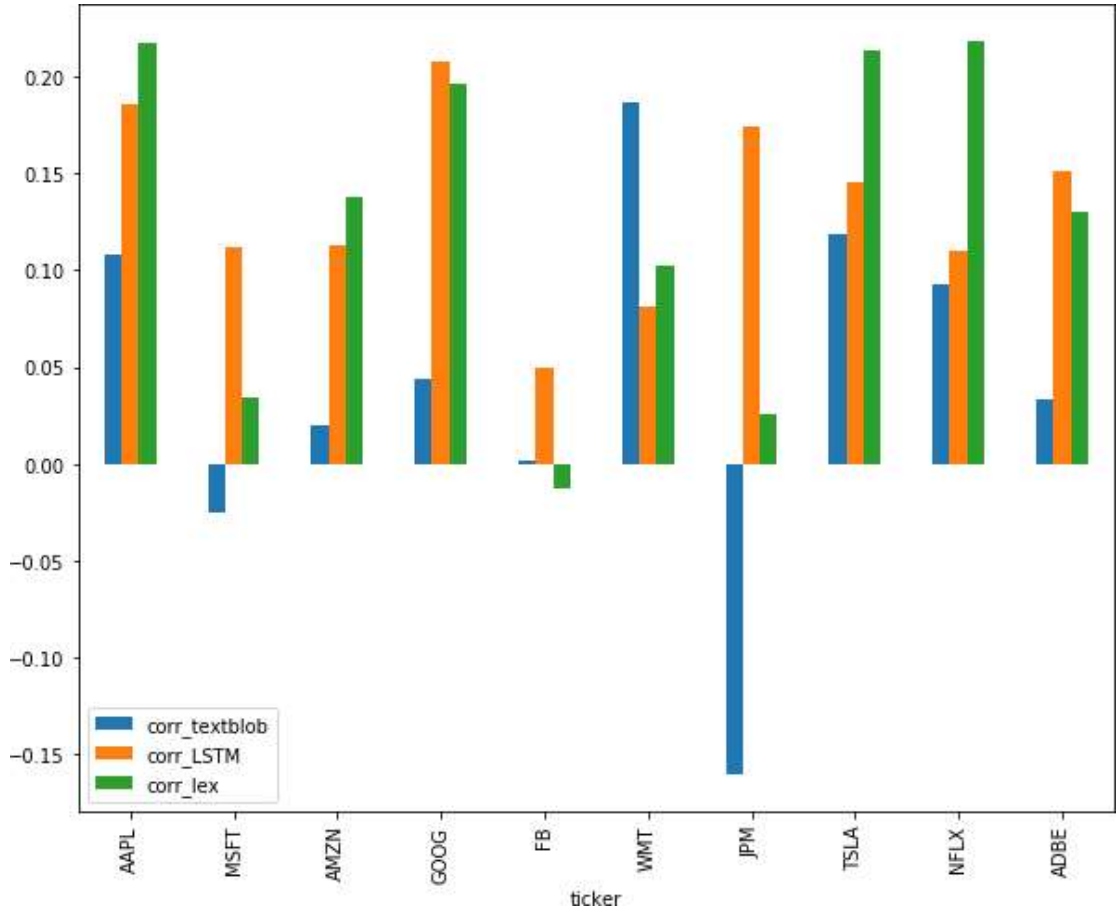
Looking at one of the headlines, the sentiment from this sentence is positive. However, the TextBlob sentiment result is smaller in magnitude, suggesting that the sentiment is more neutral. This points back to the previous assumption that the model trained on movie sentiments likely will not be accurate for stock sentiments. The classification-based model correctly suggests the sentiment is positive, but it is binary. `Sentiment_lex` has a more intuitive output with a significantly positive sentiment.

Let us review the correlation of all the sentiments from different methodologies versus returns:



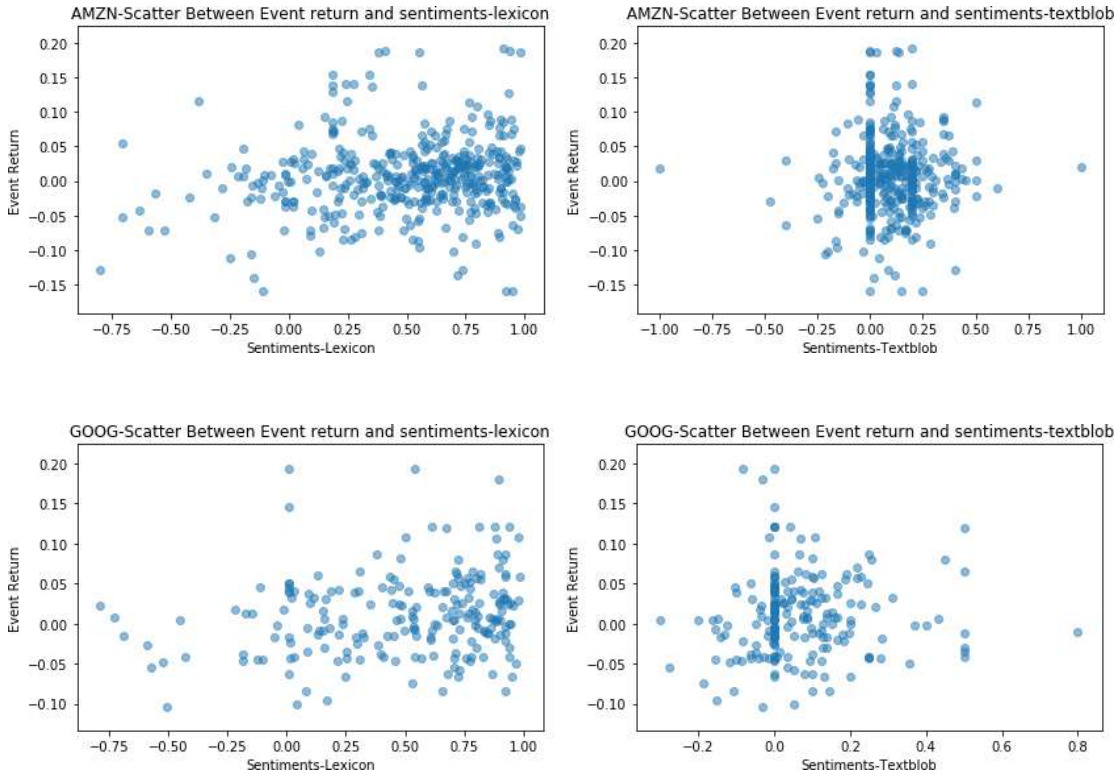
All sentiments have positive relationships with the returns, which is intuitive and expected. The sentiments from the lexicon methodology are highest, which means the stock's event return can be predicted the best using the lexicon methodology. Recall that this methodology leverages financial terms in the model. The LSTM-based method also performs better than the TextBlob approach, but the performance is slightly worse compared to the lexicon-based methodology.

Let us look at the performance of the methodology at the ticker level. We chose a few tickers with the highest market cap for the analysis:



Looking at the chart, the correlation from the lexicon methodology is highest across all stock tickers, which corroborates the conclusion from the previous analysis. It means the returns can be predicted the best using the lexicon methodology. The TextBlob-based sentiments show unintuitive results in some cases, such as with JPM.

Let us look at the scatterplot for lexicon versus TextBlob methodologies for AMZN and GOOG. We will set the LSTM-based method aside since the binary sentiments will not be meaningful in the scatterplot:



The lexicon-based sentiments on the left show a positive relationship between the sentiments and returns. Some of the points with the highest returns are associated with the most positive news. Also, the scatterplot is more uniformly distributed in the case of lexicon as compared to TextBlob. The sentiments for TextBlob are concentrated around zero, probably because the model is not able to categorize financial sentiments well. For the trading strategy, we will be using the lexicon-based sentiments, as these are the most appropriate based on the analysis in this section. The LSTM-based sentiments are good as well, but they are labeled either zero or one. The more granular lexicon-based sentiments are preferred.

5. Models evaluation—building a trading strategy

The sentiment data can be used in several ways for building a trading strategy. The sentiments can be used as a stand-alone signal to decide buy, sell, or hold actions. The sentiment score or the word vectors can also be used to predict the return or price of a stock. That prediction can be used to build a trading strategy.

In this section, we demonstrate a trading strategy in which we buy or sell a stock based on the following approach:

- Buy a stock when the change in sentiment score (current sentiment score/previous sentiment score) is greater than 0.5. Sell a stock when the change in sentiment score is less than –0.5. The sentiment score used here is based on the lexicon-based sentiments computed in the previous step.
- In addition to the sentiments, we use moving average (based on the last 15 days) while making a buy or sell decision.
- Trades (i.e., buy or sell) are in 100 shares. The initial amount available for trading is set to $100,000.

The strategy threshold, the lot size, and the initial capital can be tweaked depending on the performance of the strategy.

5.1. Setting up a strategy. To set up the trading strategy, we use *backtrader*, which is a convenient Python-based framework for implementing and backtesting trading strategies. Backtrader allows us to write reusable trading strategies, indicators, and analyzers instead of having to spend time building infrastructure. We use the Quickstart code in the backtrader documentation as a base and adapt it to our sentiment-based trading strategy.

The following code snippet summarizes the buy and sell logic for the strategy. Refer to the Jupyter notebook of this case study for the detailed implementation:

```
# buy if current close more than simple moving average (sma)
# AND sentiment increased by >= 0.5
if self.dataclose[0] > self.sma[0] and self.sentiment - prev_sentiment >= 0.5:
  self.order = self.buy()

# sell if current close less than simple moving average(sma)
# AND sentiment decreased by >= 0.5
if self.dataclose[0] < self.sma[0] and self.sentiment - prev_sentiment <= -0.5:
  self.order = self.sell()
```

5.2. Results for individual stocks. First, we run our strategy on GOOG and look at the results:

```
ticker = 'GOOG'
run_strategy(ticker, start = '2012-01-01', end = '2018-12-12')
```

The output shows the trading log for some of the days and the final return:

`Output`

```
Starting Portfolio Value: 100000.00
2013-01-10, Previous Sentiment 0.08, New Sentiment 0.80 BUY CREATE, 369.36
2014-07-17, Previous Sentiment 0.73, New Sentiment -0.22 SELL CREATE, 572.16
2014-07-18, OPERATION PROFIT, GROSS 22177.00, NET 22177.00
2014-07-18, Previous Sentiment -0.22, New Sentiment 0.77 BUY CREATE, 593.45
2014-09-12, Previous Sentiment 0.66, New Sentiment -0.05 SELL CREATE, 574.04
```

```
2014-09-15, OPERATION PROFIT, GROSS -1876.00, NET -1876.00
2015-07-17, Previous Sentiment 0.01, New Sentiment 0.90 BUY CREATE, 672.93
.
.
.
2018-12-11, Ending Value 149719.00
```

We analyze the backtesting result in the following plot produced by the backtrader package. Refer to the Jupyter notebook of this case study for the detailed version of this chart.

The results show an overall profit of $49,719. The chart is a typical chart[7] produced by the backtrader package and is divided into four panels:

Top panel
: The top panel is the *cash value observer*. It keeps track of the cash and the total portolio value during the life of the backtesting run. In this run, we started with $100,000 and ended with $149,719.

Second panel
: This panel is the *trade observer*. It shows the realized profit/loss of each trade. A trade is defined as opening a position and taking the position back to zero

7 Refer to the plotting section of the backtrader website for more details on the backtrader's charts and the panels.

(directly or crossing over from long to short or short to long). Looking at this panel, five out of eight trades are profitable for the strategy.

Third panel
: This panel is *buy sell observer*. It indicates where buy and sell operations have taken place. In general, we see that the buy action takes place when the stock price is increasing, and the sell action takes place when the stock price has started declining.

Bottom panel
: This panel shows the sentiment score, varying between –1 and 1.

Now we choose one of the days (2015-07-17) when a buy action was triggered and analyze the news for Google on that and the previous day:

```
GOOG_ticker= data_df[data_df['ticker'].isin([ticker])]
New= list(GOOG_ticker[GOOG_ticker['date'] ==  '2015-07-17']['headline'])
Old= list(GOOG_ticker[GOOG_ticker['date'] ==  '2015-07-16']['headline'])
print("Current News:",New,"\n\n","Previous News:", Old)
```

`Output`

```
Current News: ["Axiom Securities has upgraded Google (GOOG +13.4%, GOOGL +14.8%)
to Buy following the company's Q2 beat and investor-pleasing comments about
spending discipline, potential capital returns, and YouTube/mobile growth. MKM
has launched coverage at Buy, and plenty of other firms have hiked their targets.
Google's market cap is now above $450B."]

Previous News: ["While Google's (GOOG, GOOGL) Q2 revenue slightly missed
estimates when factoring traffic acquisitions costs (TAC), its ex-TAC revenue of
$14.35B was slightly above a $14.3B consensus. The reason: TAC fell to 21% of ad
revenue from Q1's 22% and Q2 2014's 23%. That also, of course, helped EPS beat
estimates.", 'Google (NASDAQ:GOOG): QC2 EPS of $6.99 beats by $0.28.']
```

Clearly, the news on the selected day mentions the upgrade of Google, a piece of positive news. The previous day mentions the revenue missing estimates, which is negative news. Hence, there was a significant change of the news sentiment on the selected day, resulting in a buy action triggered by the trading algorithm.

Next, we run the strategy for FB:

```
ticker = 'FB'
run_strategy(ticker, start = '2012-01-01', end = '2018-12-12')
```

`Output`

```
Start Portfolio value: 100000.00
Final Portfolio Value: 108041.00
Profit: 8041.00
```

The details of the backtesting results of the strategy are as follows:

Top panel
: The cash value panel shows an overall profit of $8,041.

Second panel
: The trade observer panel shows that six out of seven actions were profitable.

Third panel
: The buy/sell observer shows that in general the buy (sell) action took place when the stock price was increasing (decreasing).

Bottom panel
: It shows a high number of positive sentiments for FB around the 2013–2014 period.

5.3. Results for multiple stocks. In the previous step, we executed the trading strategy on individual stocks. Here, we run it on all 10 stocks for which we computed the sentiments:

```
results_tickers = {}
for ticker in tickers:
    results_tickers[ticker] = run_strategy(ticker, start = '2012-01-01', \
    end = '2018-12-12')
pd.DataFrame.from_dict(results_tickers).set_index(\
  [pd.Index(["PerUnitStartPrice", StrategyProfit'])])
```

Output

	AAPL	MSFT	AMZN	GOOG	FB	WMT	JPM	TSLA	NFLX	ADBE
PerUnitStartPrice	50.86	21.96	179.03	331.46	38.23	48.78	27.31	28.08	10.32	28.57
StrategyProfit	3735.00	4067.00	75377.00	49719.00	8041.00	1152.00	2014.00	15755.00	25181.00	17027.00

The strategy performs quite well and yields an overall profit for all the stocks. As mentioned before, the buy and sell actions are performed in a lot size of 100. Hence, the dollar amount used is proportional to the stock price. We see the highest nominal profit from AMZN and GOOG, which is primarily attributed to the high dollar amounts invested for these stocks given their high stock price. Other than overall profit, several other metrics, such as Sharpe ratio and maximum drawdown, can be used to analyze the performance.

5.4. Varying the strategy time period

In the previous analysis, we used the time period from 2011 to 2018 for our backtesting. In this step, to further analyze the effectiveness of our strategy, we vary the time period of the backtesting and analyze the results. First, we run the strategy for all the stocks for the time period between 2012 and 2014:

```
results_tickers = {}
for ticker in tickers:
    results_tickers[ticker] = run_strategy(ticker, start = '2012-01-01', \
    end = '2014-12-31')
```

Output

	AAPL	MSFT	AMZN	GOOG	FB	WMT	JPM	TSLA	NFLX	ADBE
StockPriceBeginning	50.86	21.96	179.03	331.46	38.23	48.78	27.31	28.08	10.32	28.57
StrategyProfit	2794.00	617.00	-2873.00	23191.00	3528.00	-313.00	2472.00	11994.00	2712.00	3367.00

The strategy yields an overall profit for all the stocks except AMZN and WMT. Now we run the strategy between 2016 and 2018:

```
results_tickers = {}
for ticker in tickers:
    results_tickers[ticker] = run_strategy(ticker, start = '2016-01-01', \
    end = '2018-12-31')
```

Output

	AAPL	MSFT	AMZN	GOOG	FB	WMT	JPM	TSLA	NFLX	ADBE
PerUnitStartPrice	97.95	50.26	636.99	741.84	102.22	54.97	55.84	223.41	109.96	91.97
StrategyProfit	-262.00	3324.00	67454.00	31430.00	648.00	657.00	0.00	10886.00	25020.00	12551.00

We see a good performance of the sentiment-based strategy across all the stocks except AAPL, and we can conclude that it performs quite well on different time periods. The strategy can be adjusted by modifying the trading rules or lot sizes. Additional metrics can also be used to understand the performance of the strategy. The sentiments can also be used along with the other features, such as correlated variables and technical indicators for prediction.

Conclusion

In this case study, we looked at various ways in which unstructured data can be converted to structured data and then used for analysis and prediction using tools for NLP. We have demonstrated three different approaches, including deep learning models to develop a model for computing the sentiments. We performed a comparison of the models and concluded that one of the most important steps in training the model for sentiment analysis is using a domain-specific vocabulary.

We also used a pretrained English model by spaCy to convert a sentence into sentiments and used the sentiments as signals to develop a trading strategy. The initial results suggested that the model trained on a financial lexicon–based sentiment could prove to be a viable model for a trading strategy. Additional improvements to this can be made by using more complex pretrained sentiment analysis models, such as BERT by Google, or different pretrained NLP models available in open source platforms. Existing NLP libraries fill in some of the preprocessing and encoding steps to allow us to focus on the inference step.

We could build on the trading strategy by including more correlated variables, technical indicators, or even improved sentiment analysis by using more sophisticated preprocessing steps and models based on more relevant financial text data.

Case Study 2: Chatbot Digital Assistant

Chatbots are computer programs that maintain a conversation with a user in natural language. They can understand the user's intent and send responses based on an organization's business rules and data. These chatbots use deep learning and NLP to process language, enabling them to understand human speech.

Chatbots are increasingly being implemented across many domains for financial services. Banking bots enable consumers to check their balance, transfer money, pay bills, and more. Brokering bots enable consumers to find investment options, make investments, and track balances. Customer support bots provide instant responses, dramatically increasing customer satisfaction. News bots deliver personalized current events information, while enterprise bots enable employees to check leave balance, file expenses, check their inventory balance, and approve transactions. In addition to

automating the process of assisting customers and employees, chatbots can help financial institutions gain information about their customers. The bot phenomenon has the potential to cause broad disruption in many areas within the finance sector.

Depending on the way bots are programmed, we can categorize chatbots into two variants:

Rule-based
: This variety of chatbots is trained according to rules. These chatbots do not learn through interactions and may sometimes fail to answer complex queries outside of the defined rules.

Self-learning
: This variety of bots relies on ML and AI technologies to converse with users. Self-learning chatbots are further divided into *retrieval-based* and *generative*:

  Retrieval-based
  : These chatbot are trained to rank the best response from a finite set of predefined responses.

  Generative
  : These chatbots are not built with predefined responses. Instead, they are trained using a large number of previous conversations. They require a very large amount of conversational data to train.

In this case study, we will prototype a self-learning chatbot that can answer financial questions.

This case study focuses on:

- Building a customized logic and rules parser using NLP for a chatbot.
- Understanding the data preparation required for building a chatbot.
- Understanding the basic building blocks of a chatbot.
- Leveraging available Python packages and corpuses to train a chatbot in a few lines of code.

Blueprint for Creating a Custom Chatbot Using NLP

1. Problem definition

The goal of this case study is to build a basic prototype of a conversational chatbot powered by NLP. The primary purpose of this chatbot is to help a user retrieve a financial ratio about a particular company. Such chatbots are designed to quickly retrieve the details about a stock or an instrument that may help the user make a trading decision.

In addition to retrieving a financial ratio, the chatbot could also engage in casual conversations with a user, perform basic mathematical calculations, and provide answers to questions from a list used to train it. We intend to use Python packages and functions for chatbot creation and to customize several components of the chatbot architecture to adapt to our requirements.

The chatbot prototype created in this case study is designed to understand user inputs and intention and retrieve the information they are seeking. It is a small prototype that could be enhanced for use as an information retrieval bot in banking, brokering, or customer support.

2. Getting started—loading the libraries

For this case study, we will use two text-based libraries: spaCy and ChatterBot. spaCy has been previously introduced; ChatterBot is a Python library used to create simple chatbots with minimal programming required.

An untrained instance of ChatterBot starts off with no knowledge of how to communicate. Each time a user enters a statement, the library saves the input and response text. As ChatterBot receives more inputs, the number of responses it can offer and the accuracy of those responses increase. The program selects the response by searching for the closest matching known statement to the input. It then returns the most likely response to that statement based on how frequently each response is issued by the people the bot communicates with.

2.1. Load libraries. We import spaCy using the following Python code:

```
import spacy #Custom NER model.
from spacy.util import minibatch, compounding
```

The ChatterBot library has the modules `LogicAdapter`, `ChatterBotCorpusTrainer`, and `ListTrainer`. These modules are used by our bot in order to construct responses to user queries. We begin by importing the following:

```
from chatterbot import ChatBot
from chatterbot.logic import LogicAdapter
from chatterbot.trainers import ChatterBotCorpusTrainer
from chatterbot.trainers import ListTrainer
```

Other libraries used in this exercise are as follows:

```
import random
from itertools import product
```

Before we move to the customized chatbot, let us develop a chatbot using the default features of the ChatterBot package.

3. Training a default chatbot

ChatterBot and many other chatbot packages come with a data utility module that can be used to train chatbots. Here are the ChatterBot components we will be using:

Logic adapters
: Logic adapters determine the logic for how ChatterBot selects a response to a given input statement. It is possible to enter any number of logic adapters for your bot to use. In the example below, we are using two inbuilt adapters: *BestMatch*, which returns the best known responses, and *MathematicalEvaluation*, which performs mathematical operations.

Preprocessors
: ChatterBot's preprocessors are simple functions that modify the input statement a chatbot receives before the statement gets processed by the logic adapter. The preprocessors can be customized to perform different preprocessing steps, such as tokenization and lemmatization, in order to have clean and processed data. In the example below, the default preprocessor for cleaning white spaces, `clean_whitespace`, is used.

Corpus training
: ChatterBot comes with a corpus data and utility module that makes it easy to quickly train the bot to communicate. We use the already existing corpuses *english, english.greetings*, and *english.conversations* for training the chatbot.

List training
: Just like the corpus training, we train the chatbot with the conversations that can be used for training using *ListTrainer*. In the example below, we have trained the chatbot using some sample commands. The chatbot can be trained using a significant amount of conversation data.

```
chatB = ChatBot("Trader",
                preprocessors=['chatterbot.preprocessors.clean_whitespace'],
                logic_adapters=['chatterbot.logic.BestMatch',
                                'chatterbot.logic.MathematicalEvaluation'])

# Corpus Training
trainerCorpus = ChatterBotCorpusTrainer(chatB)

# Train based on English Corpus
trainerCorpus.train(
    "chatterbot.corpus.english"
)
# Train based on english greetings corpus
trainerCorpus.train("chatterbot.corpus.english.greetings")

# Train based on the english conversations corpus
trainerCorpus.train("chatterbot.corpus.english.conversations")

trainerConversation = ListTrainer(chatB)
# Train based on conversations

# List training
trainerConversation.train([
    'Help!',
    'Please go to google.com',
    'What is Bitcoin?',
    'It is a decentralized digital currency'
])

# You can train with a second list of data to add response variations
trainerConversation.train([
    'What is Bitcoin?',
    'Bitcoin is a cryptocurrency.'
])
```

Once the chatbot is trained, we can test the trained chatbot by having the following conversation:

```
>Hi
How are you doing?

>I am doing well.
That is good to hear

>What is 78964 plus 5970
78964 plus 5970 = 84934

>what is a dollar
dollar: unit of currency in the united states.

>What is Bitcoin?
It is a decentralized digital currency
```

```
>Help!
Please go to google.com

>Tell me a joke
Did you hear the one about the mountain goats in the andes? It was "ba a a a d".

>What is Bitcoin?
Bitcoin is a cryptocurrency.
```

In this example, we see a chatbot that gives an intuitive reply in response to the input. The first two responses are due to the training on the English greetings and English conversation corpuses. Additionally, the responses to *Tell me a joke* and *what is a dollar* are due to the training on the English corpus. The computation in the fourth line is the result of the chatbot being trained on the `MathematicalEvaluation` logical adapter. The responses to *Help!* and *What is Bitcoin?* are the result of the customized list trainers. Additionally, we see two different replies to *What is Bitcoin?*, given that we trained it using the list trainers.

Next, we move on to creating a chatbot designed to use a customized logical adapter to give financial ratios.

4. Data preparation: Customized chatbot

We want our chatbot to be able to recognize and group subtly different inquiries. For example, one might want to ask about the company *Apple Inc.* by simply referring to it as *Apple*, and we would want to map it to a ticker—*AAPL*, in this case. Constructing commonly used phrases in order to refer to firms can be built by using a dictionary as follows:

```
companies = {
    'AAPL':  ['Apple', 'Apple Inc'],
    'BAC': ['BAML', 'BofA', 'Bank of America'],
    'C': ['Citi', 'Citibank'],
    'DAL': ['Delta', 'Delta Airlines']
}
```

Similarly, we want to build a map for financial ratios:

```
ratios = {
    'return-on-equity-ttm': ['ROE', 'Return on Equity'],
    'cash-from-operations-quarterly': ['CFO', 'Cash Flow from Operations'],
    'pe-ratio-ttm': ['PE', 'Price to equity', 'pe ratio'],
    'revenue-ttm': ['Sales', 'Revenue'],
}
```

The keys of this dictionary can be used to map to an internal system or API. Finally, we want the user to be able to request the phrase in multiple formats. Saying *Get me the [RATIO] for [COMPANY]* should be treated similarly to *What is the [RATIO] for [COMPANY]?* We build these sentence templates for our model to train on by building a list as follows:

```
string_templates = ['Get me the {ratio} for {company}',
                    'What is the {ratio} for {company}?',
                    'Tell me the {ratio} for {company}',
                   ]
```

4.1. Data construction. We begin constructing our model by creating *reverse dictionaries*:

```
companies_rev = {}
for k, v in companies.items():
  for ve in v:
      companies_rev[ve] = k
  ratios_rev = {}
  for k, v in ratios.items():
        for ve in v:
             ratios_rev[ve] = k
  companies_list = list(companies_rev.keys())
  ratios_list = list(ratios_rev.keys())
```

Next, we create sample statements for our model. We build a function that gives us a random sentence structure, inquiring about a random financial ratio for a random company. We will be creating a custom named entity recognition_ model in the spaCy framework. This requires training the model to pick up the word or phrase in a sample sentence. To train the spaCy model, we need to provide it with an example, such as *(Get me the ROE for Citi, {"entities": [(11, 14, RATIO), (19, 23, COMPANY)]}).*

4.2. Training data. The first part of the training example is the sentence. The second is a dictionary that consists of entities and the starting and ending index of the label:

```
N_training_samples = 100
def get_training_sample(string_templates, ratios_list, companies_list):
  string_template=string_templates[random.randint(0, len(string_templates)-1)]
      ratio = ratios_list[random.randint(0, len(ratios_list)-1)]
      company = companies_list[random.randint(0, len(companies_list)-1)]
      sent = string_template.format(ratio=ratio,company=company)
      ents = {"entities": [(sent.index(ratio), sent.index(ratio)+\
  len(ratio), 'RATIO'),
                    (sent.index(company), sent.index(company)+len(company), \
                    'COMPANY')]}
       return (sent, ents)
```

Let us define the training data:

```
TRAIN_DATA = [
get_training_sample(string_templates, ratios_list, companies_list) \
for i in range(N_training_samples)
]
```

5. Model creation and training. Once we have the training data, we construct a *blank* model in spaCy. spaCy's models are statistical, and every decision they make—for example, which part-of-speech tag to assign, or whether a word is a named entity—is a prediction. This prediction is based on the examples the model has seen during training. To train a model, you first need training data—examples of text and the labels you want the model to predict. This could be a part-of-speech tag, a named entity, or any other information. The model is then shown the unlabeled text and makes a prediction. Because we know the correct answer, we can give the model feedback on its prediction in the form of an *error gradient* of the loss function. This calculates the difference between the training example and the expected output, as shown in Figure 10-6. The greater the difference, the more significant the gradient, and the more updates we need to make to our model.

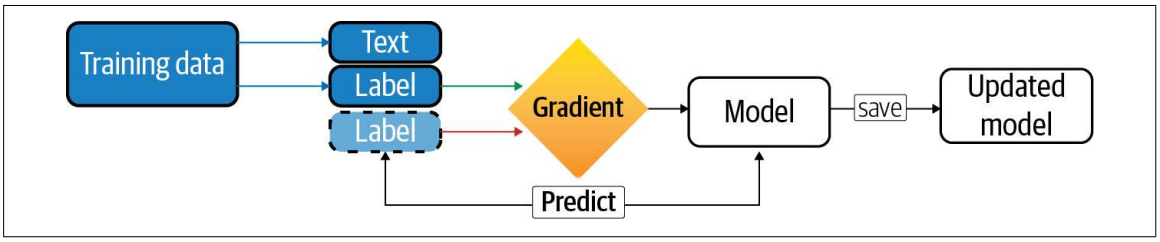


Figure 10-6. Machine learning–based training in spaCy

```
nlp = spacy.blank("en")
```

Next, we create an NER pipeline to our model:

```
ner = nlp.create_pipe("ner")
nlp.add_pipe(ner)
```

Then we add the training labels that we use:

```
ner.add_label('RATIO')
ner.add_label('COMPANY')
```

5.1. Model optimization function

Now we start optimization of our models:

```
optimizer = nlp.begin_training()
move_names = list(ner.move_names)
pipe_exceptions = ["ner", "trf_wordpiecer", "trf_tok2vec"]
other_pipes = [pipe for pipe in nlp.pipe_names if pipe not in pipe_exceptions]
with nlp.disable_pipes(*other_pipes):  # only train NER
     sizes = compounding(1.0, 4.0, 1.001)
     # batch up the examples using spaCy's minibatch
     for itn in range(30):
        random.shuffle(TRAIN_DATA)
        batches = minibatch(TRAIN_DATA, size=sizes)
        losses = {}
```

```
        for batch in batches:
            texts, annotations = zip(*batch)
            nlp.update(texts, annotations, sgd=optimizer,
            drop=0.35, losses=losses)
        print("Losses", losses)
```

Training the NER model is akin to updating the weights for each token. The most important step is to use a good optimizer. The more examples of our training data that we provide spaCy, the better it will be at recognizing generalized results.

5.2. Custom logic adapter

Next, we build our custom logic adapter:

```
from chatterbot.conversation import Statement
class FinancialRatioAdapter(LogicAdapter):
     def __init__(self, chatbot, **kwargs):
          super(FinancialRatioAdapter, self).__init__(chatbot, **kwargs)
     def process(self, statement, additional_response_selection_parameters):
        user_input = statement.text
        doc = nlp(user_input)
        company = None
        ratio = None
        confidence = 0
        # We need exactly 1 company and one ratio
        if len(doc.ents) == 2:
         for ent in doc.ents:
             if ent.label_ == "RATIO":
                  ratio = ent.text
                 if ratio in ratios_rev:
                      confidence += 0.5
             if ent.label_ == "COMPANY":
                  company = ent.text
                  if company in companies_rev:
                       confidence += 0.5
        if confidence > 0.99: (its found a ratio and company)
         outtext = '''https://www.zacks.com/stock/chart\
                /{comanpy}/fundamental/{ratio} '''.format(ratio=ratios_rev[ratio]\
                  , company=companies_rev[company])
         confidence = 1
        else:
         outtext = 'Sorry! Could not figure out what the user wants'
         confidence = 0
        output_statement = Statement(text=outtext)
        output_statement.confidence = confidence
        return output_statement
```

With this custom logic adapter, our chatbot will take each input statement and try to recognize a *RATIO* and/or *COMPANY* using our NER model. If the model finds exactly one *COMPANY* and exactly one *RATIO*, it constructs a URL to guide the user.

5.3. Model usage—training and testing

Now we begin using our chatbot by using the following import:

```
from chatterbot import ChatBot
```

We construct our chatbot by adding the `FinancialRatioAdapter` logical adapter that we created above to the chatbot. Although the following code snippet only shows us adding the `FinancialRatioAdapter`, note that other logical adapters, lists, and corpuses used in the prior training of the chatbot are also included. Please refer to the Jupyter notebook of the case study for more details.

```
chatbot = ChatBot(
       "My ChatterBot",
       logic_adapters=[
        'financial_ratio_adapter.FinancialRatioAdapter'
    ]
)
```

Now we test our chatbot using the following statements:

```
converse()

>What is ROE for Citibank?
https://www.zacks.com/stock/chart/C/fundamental/return-on-equity-ttm

>Tell me PE for Delta?
https://www.zacks.com/stock/chart/DAL/fundamental/pe-ratio-ttm

>What is Bitcoin?
It is a decentralized digital currency

>Help!
Please go to google.com

>What is 786940 plus 75869
786940 plus 75869 = 862809

>Do you like dogs?
Sorry! Could not figure out what the user wants
```

As shown above, the custom logic adapter for our chatbot finds a RATIO and/or COMPANY in the sentence using our NLP model. If an exact pair is detected, the model constructs a URL to guide the user to the answer. Additionally, other logical adapters, such as mathematical evaluation, work as expected.

Conclusion

Overall, this case study provides an introduction to a number of aspects of chatbot development.

Using the ChatterBot library in Python allows us to build a simple interface to resolve user inputs. To train a blank model, one must have a substantial training dataset. In this case study, we looked at patterns available to us and used them to generate training samples. Getting the right amount of training data is usually the hardest part of constructing a custom chatbot.

This case study is a demo project, and significant enhancements can be made to each component to extend it to a wide variety of tasks. Additional preprocessing steps can be added to have cleaner data to work with. To generate a response from our bot for input questions, the logic can be refined further to incorporate better similarity measures and embeddings. The chatbot can be trained on a bigger dataset using more advanced ML techniques. A series of custom logic adapters can be used to construct a more sophisticated ChatterBot. This can be generalized to more interesting tasks, such as retrieving information from a database or asking for more input from the user.

Case Study 3: Document Summarization

Document summarization refers to the selection of the most important points and topics in a document and arranging them in a comprehensive manner. As discussed earlier, analysts at banks and other financial service organizations pore over, analyze, and attempt to quantify qualitative data from news, reports, and documents. Document summarization using NLP can provide in-depth support in this analyzing and interpretation. When tailored to financial documents, such as earning reports and financial news, it can help analysts quickly derive key topics and market signals from content. Document summarization can also be used to improve reporting efforts and can provide timely updates on key matters.

In NLP, *topic models* (such as LDA, introduced earlier in the chapter) are the most frequently used tools for the extraction of sophisticated, interpretable text features. These models can surface key topics, themes, or signals from large collections of documents and can be effectively used for document summarization.

In this case study, we will focus on:

- Implementing the LDA model for topic modeling.
- Understanding the necessary data preparation (i.e., converting a PDF for an NLP-related problem).
- Topic visualization.

Blueprint for Using NLP for Document Summarization

1. Problem definition

The goal of this case study is to effectively discover common topics from earnings call transcripts of publicly traded companies using LDA. A core advantage of this technique compared to other approaches, is that no prior knowledge of the topics is needed.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. For this case study, we will extract the text from a PDF. Hence, the Python library *pdf-miner* is used for processing the PDF files into a text format. Libraries for feature extraction and topic modeling are also loaded. The libraries for the visualization will be loaded later in the case study:

```
Libraries for pdf conversion
```

```
from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from pdfminer.pdfpage import PDFPage
import re
from io import StringIO
```

```
Libraries for feature extraction and topic modeling
```

```
from sklearn.feature_extraction.text import CountVectorizer,TfidfVectorizer
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.stop_words import ENGLISH_STOP_WORDS
```

```
Other libraries
```

```
import numpy as np
import pandas as pd
```

3. Data preparation

The `convert_pdf_to_txt` function defined below pulls out all characters from a PDF document except the images. The function simply takes in the PDF document, extracts all characters from the document, and outputs the extracted text as a Python list of strings:

```
def convert_pdf_to_txt(path):
    rsrcmgr = PDFResourceManager()
```

```
    retstr = StringIO()
    laparams = LAParams()
    device = TextConverter(rsrcmgr, retstr, laparams=laparams)
    fp = open(path, 'rb')
    interpreter = PDFPageInterpreter(rsrcmgr, device)
    password = ""
    maxpages = 0
    caching = True
    pagenos=set()

    for page in PDFPage.get_pages(fp, pagenos,\
            maxpages=maxpages, password=password,caching=caching,\
            check_extractable=True):
        interpreter.process_page(page)

    text = retstr.getvalue()

    fp.close()
    device.close()
    retstr.close()
    return text
```

In the next step, the PDF is converted to text using the above function and saved in a text file:

```
Document=convert_pdf_to_txt('10K.pdf')
f=open('Finance10k.txt','w')
f.write(Document)
f.close()
with open('Finance10k.txt') as f:
    clean_cont = f.read().splitlines()
```

Let us look at the raw document:

```
clean_cont[1:15]
```

Output

```
[' ',
 '',
 'SECURITIES AND EXCHANGE COMMISSION',
 ' ',
 '',
 'Washington, D.C. 20549',
 ' ',
 '',
 '\xa0',
 'FORM ',
 '\xa0',
 '',
 'QUARTERLY REPORT PURSUANT TO SECTION 13 OR 15(d) OF',
 ' ']
```

The text extracted from the PDF document contains uninformative characters that need to be removed. These characters reduce the effectiveness of our models as they provide unnecessary count ratios. The following function uses a series of regular expression (*regex*) searches as well as list comprehension to replace uninformative characters with a blank space:

```
doc=[i.replace('\xe2\x80\x9c', '') for i in clean_cont ]
doc=[i.replace('\xe2\x80\x9d', '') for i in doc ]
doc=[i.replace('\xe2\x80\x99s', '') for i in doc ]

docs = [x for x in doc if x != ' ']
docss = [x for x in docs if x != '']
financedoc=[re.sub("[^a-zA-Z]+", " ", s) for s in docss]
```

4. Model construction and training

The `CountVectorizer` function from the sklearn module is used with minimal parameter tuning to represent the clean document as a *document term matrix*. This is performed because our modeling requires that strings be represented as integers. The `CountVectorizer` shows the number of times a word occurs in the list after the removal of stop words. The document term matrix was formatted into a Pandas dataframe in order to inspect the dataset. This dataframe shows the word-occurrence count of each term in the document:

```
vect=CountVectorizer(ngram_range=(1, 1),stop_words='english')
fin=vect.fit_transform(financedoc)
```

In the next step, the document term matrix will be used as the input data to the LDA algorithm for topic modeling. The algorithm was fitted to isolate five distinct topic contexts, as shown by the following code. This value can be adjusted depending on the level of granularity one intends to obtain from the modeling:

```
lda=LatentDirichletAllocation(n_components=5)
lda.fit_transform(fin)
lda_dtf=lda.fit_transform(fin)

sorting=np.argsort(lda.components_)[:, ::-1]
features=np.array(vect.get_feature_names())
```

The following code uses the *mglearn* library to display the top 10 words within each specific topic model:

```
import mglearn
mglearn.tools.print_topics(topics=range(5), feature_names=features,
sorting=sorting, topics_per_chunk=5, n_words=10)
```

`Output`

```
topic 1       topic 2       topic 3       topic 4       topic 5
--------      --------      --------      --------      --------
assets        quarter       loans         securities    value
```

```
balance       million       mortgage      rate          total
losses        risk          loan          investment    income
credit        capital       commercial    contracts     net
period        months        total         credit        fair
derivatives   financial     real          market        billion
liabilities   management    estate        federal       equity
derivative    billion       securities    stock         september
allowance     ended         consumer      debt          december
average       september     backed        sales         table
```

Each topic in the table is expected to represent a broader theme. However, given that we trained the model on only a single document, the themes across the topics may not be very distinct from each other.

Looking at the broader theme, topic 2 discusses quarters, months, and currency units related to asset valuation. Topic 3 reveals information on income from real estate, mortgages, and related instrument. Topic 5 also has terms related to asset valuation. The first topic references balance sheet items and derivatives. Topic 4 is slightly similar to topic 1 and has words related to an investment process.

In terms of overall theme, topics 2 and 5 are quite distinct from the others. There may also be some similarity between topics 1 and 4, based on the top words. In the next section, we will try to understand the separation between these topics using the Python library *pyLDAvis*.

5. Visualization of topics

In this section, we visualize the topics using different techniques.

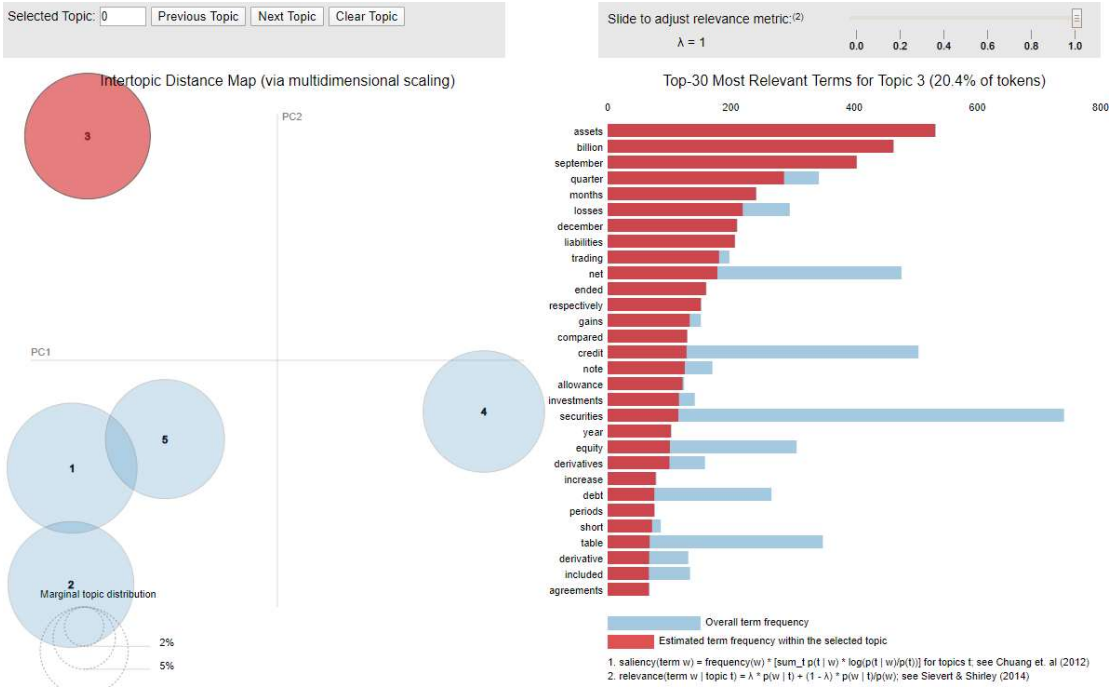
5.1. Topic visualization. *Topic visualization* facilitates the evaluation of topic quality using human judgment. *pyLDAvis* is a library that displays the global relationships between topics while also facilitating their semantic evaluation by inspecting the terms most closely associated with each topic and, inversely, the topics associated with each term. It also addresses the challenge in which frequently used terms in a document tend to dominate the distribution over words that define a topic.

Below, the *pyLDAvis_* library is used to visualize the topic models:

```
from __future__ import  print_function
import pyLDAvis
import pyLDAvis.sklearn

zit=pyLDAvis.sklearn.prepare(lda,fin,vect)
pyLDAvis.show(zit)
```

Output



We notice that topics 2 and 5 are quite distant from each other. This is what we observed in the section above from the overall theme and list of words under these topics. Topics 1 and 4 are quite close, which validates our observation above. Such close topics should be analyzed more intricately and might be combined if needed. The relevance of the terms under each topic, as shown in the right panel of the chart, can also be used to understand the differences. Topics 3 and 4 are relatively close as well, although topic 3 is quite distant from the others.

5.2. Word cloud. In this step, a *word cloud* is generated for the entire document to note the most recurrent terms in the document:

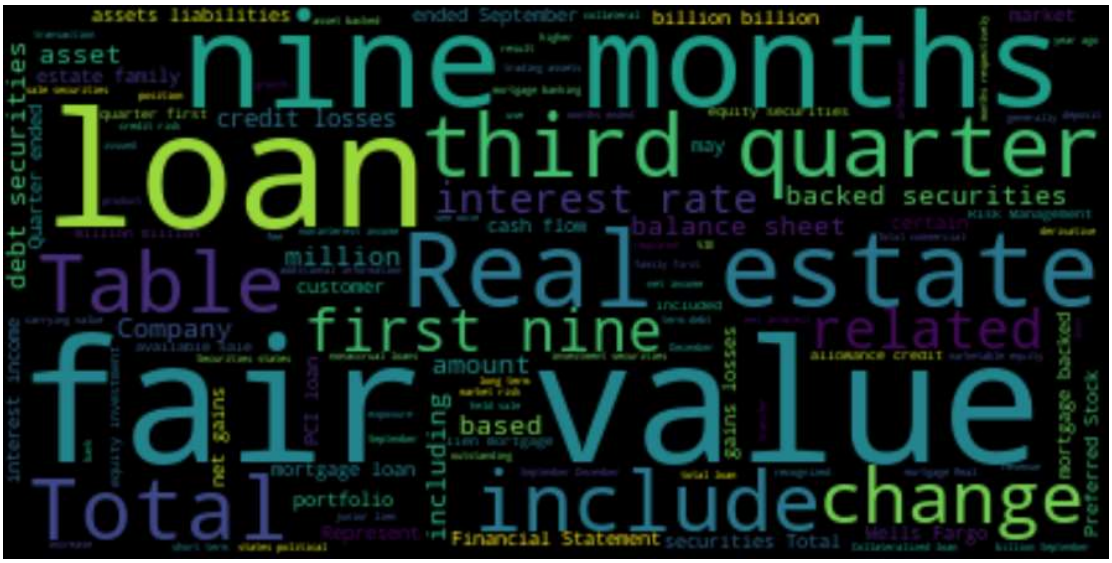
```
#Loading the additional packages for word cloud
from os import path
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from wordcloud import WordCloud,STOPWORDS

#Loading the document and generating the word cloud
d = path.dirname(__name__)
text = open(path.join(d, 'Finance10k.txt')).read()

stopwords = set(STOPWORDS)
wc = WordCloud(background_color="black", max_words=2000, stopwords=stopwords)
wc.generate(text)
```

```
    plt.figure(figsize=(16,13))
    plt.imshow(wc, interpolation='bilinear')
    plt.axis("off")
    plt.show()
```

Output



The word cloud generally agrees with the results from the topic modeling, as recurrent words, such as *loan*, *real estate*, *third quarter*, and *fair value*, are larger and bolder.

By integrating the information from the steps above, we may come up with the list of topics represented by the document. For the document in our case study, we see that words like *third quarter*, *first nine*, and *nine months* occur quite frequently. In the word list, there are several topics related to balance sheet items. So the document might be a third-quarter financial balance sheet with all credit and assets values in that quarter.

Conclusion

In this case study, we explored the use of topic modeling to gain insights into the content of a document. We demonstrated the use of the LDA model, which extracts plausible topics and allows us to gain a high-level understanding of large amounts of text in an automated way.

We performed extraction of the text from a document in PDF format and performed further data preprocessing. The results, alongside the visualizations, demonstrated that the topics are intuitive and meaningful.

Overall, the case study shows how machine learning and NLP can be applied across many domains—such as investment analysis, asset modeling, risk management, and regulatory compliance—to summarize documents, news, and reports in order to significantly reduce manual processing. Given this ability to quickly access and verify relevant, filtered information, analysts may be able to provide more comprehensive and informative reports on which management can base their decisions.

Chapter Summary

The field of NLP has made significant progress, resulting in technologies that have and will continue to revolutionize how financial institutions operate. In the near term, we are likely to see an increase in NLP-based technologies across different domains of finance, including asset management, risk management, and process automation. The adoption and understanding of NLP methodologies and related infrastructure are very important for financial institutions.

Overall, the concepts in Python, machine learning, and finance presented in this chapter through the case studies can be used as a blueprint for any other NLP-based problem in finance.

Exercises

- Using the concepts from case study 1, use NLP-based techniques to develop a trading strategy using Twitter data.
- In case study 1, use the word2vec word embedding method to generate the word vectors and incorporate it into the trading strategy.
- Using the concepts from case study 2, test a few more logical adapters to the chatbot.
- Using the concepts from case study 3, perform topic modeling on a set of financial news articles for a given day and retrieve the key themes of the day.