13
Working with Text Data

This is the first of three chapters dedicated to extracting signals for algorithmic trading strategies from text data using **natural language processing** (**NLP**) and **machine learning** (**ML**).

Text data is very rich in content, yet unstructured in format, and hence requires more preprocessing so that an ML algorithm can extract the potential signal. The key challenge lies in converting text into a numerical format for use by an algorithm, while simultaneously expressing the semantics or meaning of the content. We will cover several techniques that capture nuances of language that are readily understandable to humans so that they can become an input for ML algorithms.

In this chapter, we introduce fundamental feature extraction techniques that focus on individual semantic units; that is, words or short groups of words called **tokens**. We will show how to represent documents as vectors of token counts by creating a document-term matrix that, in turn, serves as input for text classification and sentiment analysis. We will also introduce the Naive Bayes algorithm, which is popular for this purpose.

In the following two chapters, we build on these techniques and use ML algorithms such as topic modeling and word-vector embedding to capture information contained in a broader context.

In particular, in this chapter, we will cover the following:

- What the fundamental NLP workflow looks like
- How to build a multilingual feature extraction pipeline using `spaCy` and `TextBlob`
- How to perform NLP tasks such as **part-of-speech** (**POS**) tagging or named entity recognition
- How to convert tokens to numbers using the document-term matrix
- How to classify text using the Naive Bayes model
- How to perform sentiment analysis

The code samples for the following sections are in the GitHub repository for this chapter, and references are listed in the main `README` file.

How to extract features from text data

Text data can be extremely valuable given how much information humans communicate and store using natural language—the diverse set of data sources relevant to investment range from formal documents such as company statements, contracts, and patents, to news, opinion, and analyst research, and even to commentary and various types of social media posts and messages.

Numerous and diverse text data samples are available online to explore the use of NLP algorithms, many of which are listed among the references for this chapter.

To guide our journey through the techniques and Python libraries that most effectively support the realization of this goal, we will highlight NLP challenges, introduce critical elements of the NLP workflow, and illustrate applications of ML from text data to algorithmic trading.

Challenges of NLP

The conversion of unstructured text into a machine-readable format requires careful preprocessing to preserve valuable semantic aspects of the data. How humans derive meaning from, and comprehend the content of language, is not fully understood and improving language understanding by machines remains an area of very active research.

NLP is challenging because the effective use of text data for ML requires an understanding of the inner workings of language as well as knowledge about the world to which it refers. Key challenges include the following:

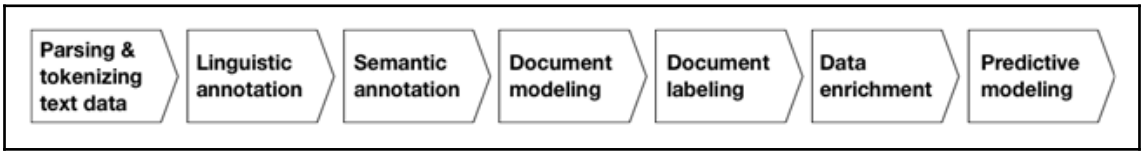
- Ambiguity due to polysemy; that is, a word or phrase can have different meanings depending on context (local high-school dropouts cut in half could be taken a couple of ways, for instance).
- Non-standard and evolving use of language, especially in social media.

- The use of idioms, such as throw in the towel.
- Tricky entity names, Where is A Bug's Life playing?
- Knowledge of the world—Mary and Sue are sisters versus Mary and Sue are mothers.

The NLP workflow

A key goal in using ML from text data for algorithmic trading is to extract signals from documents. A document is an individual sample from a relevant text data source, such as a company report, a headline or news article, or a tweet. A corpus, in turn, is a collection of documents (plural: *corpora*).

The following diagram lays out the key steps to convert documents into a dataset that can be used to train a supervised ML algorithm capable of making actionable predictions:



Fundamental techniques extract text features semantic units called **tokens**, and use linguistic rules and dictionaries to enrich these tokens with linguistic and semantic annotations. The **bag-of-words** (**BoW**) model uses token frequency to model documents as token vectors, which leads to the document-term matrix that is frequently used for text classification.

Advanced approaches use ML to refine features extracted by these fundamental techniques and produce more informative document models. These include topic models that reflect the joint usage of tokens across documents and word-vector models that capture the context of token usage.

We will review key decisions made at each step and related trade-offs in more detail before illustrating their implementation using the `spaCy` library in the next section. The following table summarizes the key tasks of an NLP pipeline:

Feature	Description
Tokenization	Segments text into words, punctuation marks, and so on.
POS tagging	Assigns word types to tokens, such as a verb or noun.
Dependency parsing	Labels syntactic token dependencies, such as *subject* <=> *object*.

Stemming and lemmatization	Assigns the base forms of words: *was => be*, *rats => rat*.
Sentence boundary detection	Finds and segments individual sentences.
Named entity recognition	Labels real-world objects, such as people, companies, and locations.
Similarity	Evaluates the similarity of words, text spans, and documents.

Parsing and tokenizing text data

A token is an instance of a characters that appears in a given document and should be considered a semantic unit for further processing. The vocabulary is a set of tokens contained in a corpus deemed relevant for further processing. A key trade-off in the following decisions is the accurate reflection of the text source at the expense of a larger vocabulary that may translate into more features and higher model complexity.

Basic choices in this regard concern the treatment of punctuation and capitalization, the use of spelling correction, and whether to exclude very frequent so-called **stop words** (such as *and* or *the*) as meaningless noise.

An additional decision is about the inclusion of groups of *n* individual tokens called **n-grams** as semantic units (an individual token is also called a **unigram**). An example of a 2-gram (or bi-gram) is New York, whereas New York City is a 3-gram (or tri-gram).

The goal is to create tokens that more accurately reflect the document's meaning. The decision can rely on dictionaries or a comparison of the relative frequencies of the individual and joint usage. Including n-grams will increase the number of features because the number of unique n-grams tends to be much higher than the number of unique unigrams and will likely add noise unless filtered for significance by frequency.

Linguistic annotation

Linguistic annotations include the application of **syntactic and grammatical rules** to identify the boundary of a sentence despite ambiguous punctuation, and a token's role in a sentence for POS tagging and dependency parsing. It also permits the identification of common root forms for stemming and lemmatization to group related words:

- **POS annotations:** It helps disambiguate tokens based on their function (this may be necessary when a verb and noun have the same form), which increases the vocabulary but may result in better accuracy.

- **Dependency parsing**: It identifies hierarchical relationships among tokens, is commonly used for translation, and is important for interactive applications that require more advanced language understanding, such as chatbots.
- **Stemming**: It uses simple rules to remove common endings, such as *s*, *ly*, *ing*, and *ed*, from a token and reduce it to its stem or root form.
- **Lemmatization**: It uses more sophisticated rules to derive the canonical root (lemma) of a word. It can detect irregular roots, such as better and best, and more effectively condenses vocabulary, but is slower than stemming. Both approaches simplify vocabulary at the expense of semantic nuances.

Semantic annotation

Named entity recognition (**NER**) aims to identify tokens that represent objects of interest, such as people, countries, or companies. It can be further developed into a **knowledge graph** that captures semantic and hierarchical relationships among such entities. It is a critical ingredient for applications that, for example, aim to predict the impact of news events or sentiment.

Labeling

Many NLP applications learn to predict outcomes from meaningful information extracted from text. Supervised learning requires labels to teach the algorithm the true input-output relationship. With text data, establishing this relationship may be challenging and may require explicit data modeling and collection.

Data modeling decisions include how to quantify sentiments implicit in a text document like an email, a transcribed interview, or a tweet, or which aspects of a research document or news report to assign to a specific outcome.

Use cases

The use of ML with text data for algorithmic trading relies on the extraction of meaningful information in the form of features that directly or indirectly predict future price movements. Applications range from the exploitation of the short-term market impact of news to the long-term fundamental analysis of the drivers of asset valuation. Examples include the following:

- The evaluation of product review sentiment to assess a company's competitive position or industry trends

- The detection of anomalies in credit contracts to predict the probability or impact of a default
- The prediction of news impact in terms of direction, magnitude, and affected entities

JP Morgan, for instance, developed a predictive model based on 250,000 analyst reports that outperformed several benchmark indices and produced uncorrelated signals relative to sentiment factors formed from consensus EPS and recommendation changes.

From text to tokens – the NLP pipeline

In this section, we will demonstrate how to construct an NLP pipeline using the open source Python library, `spaCy`. The `textacy` library builds on `spaCy` and provides easy access to `spaCy` attributes and additional functionality.

Refer to the `nlp_pipeline_with_spaCy` notebook for the following code samples, installation instructions, and additional details.

NLP pipeline with spaCy and textacy

`spaCy` is a widely used Python library with a comprehensive feature set for fast text processing in multiple languages. The usage of tokenization and annotation engines requires the installation of language models. The features we will use in this chapter only require small models; larger models also include word vectors that we will cover in `Chapter 15`, *Word Embeddings*.

Once installed and linked, we can instantiate a `spaCy` language model and then call it on a document. As a result, `spaCy` produces a `doc` object that tokenizes the text and processes it according to configurable pipeline components that, by default, consist of a tagger, a parser, and a named-entity recognizer:

```
nlp = spacy.load('en')
nlp.pipe_names
['tagger', 'parser', 'ner']
```

Let's illustrate the pipeline using a simple sentence:

```
sample_text = 'Apple is looking at buying U.K. startup for $1 billion'
doc = nlp(sample_text)
```

Parsing, tokenizing, and annotating a sentence

Parsed document content is iterable, and each element has numerous attributes produced by the processing pipeline. The following sample illustrates how to access the following attributes:

- `.text`: Original word text
- `.lemma_`: Word root
- `.pos_`: Basic POS tag
- `.tag_`: Detailed POS tag
- `.dep_`: Syntactic relationship or dependency between tokens
- `.shape_`: The shape of the word regarding capitalization, punctuation, or digits
- `.is alpha`: Check whether the token is alphanumeric
- `.is stop`: Check whether the token is on a list of common words for the given language

We iterate over each token and assign its attributes to a `pd.DataFrame`:

```
pd.DataFrame([[t.text, t.lemma_, t.pos_, t.tag_, t.dep_, t.shape_,
t.is_alpha, t.is_stop] for t in doc],
             columns=['text', 'lemma', 'pos', 'tag', 'dep', 'shape',
'is_alpha', 'is_stop'])
```

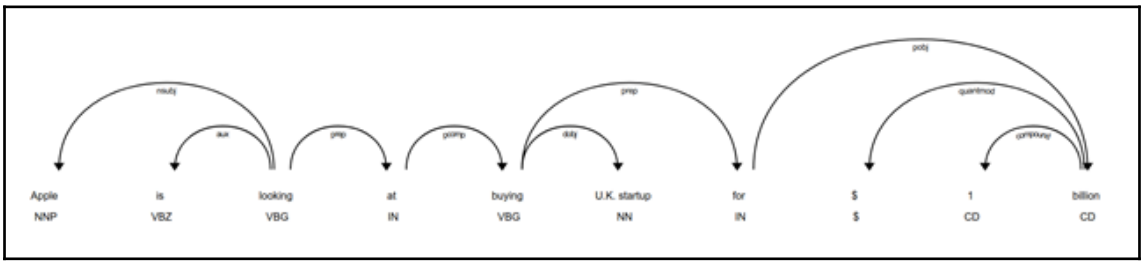
Which produces the following output:

text	lemma	pos	tag	dep	shape	is_alpha	is_stop
Apple	apple	PROPN	NNP	nsubj	Xxxxx	TRUE	FALSE
is	be	VERB	VBZ	aux	xx	TRUE	TRUE
looking	look	VERB	VBG	ROOT	xxxx	TRUE	FALSE
at	at	ADP	IN	prep	xx	TRUE	TRUE
buying	buy	VERB	VBG	pcomp	xxxx	TRUE	FALSE
U.K.	u.k.	PROPN	NNP	compound	X.X.	FALSE	FALSE
startup	startup	NOUN	NN	dobj	xxxx	TRUE	FALSE
for	for	ADP	IN	prep	xxx	TRUE	TRUE
$	$	SYM	$	quantmod	$	FALSE	FALSE
1	1	NUM	CD	compound	d	FALSE	FALSE
billion	billion	NUM	CD	pobj	xxxx	TRUE	FALSE

We can visualize syntactic dependency in a browser or notebook using the following:

```
displacy.render(doc, style='dep', options=options, jupyter=True)
```

The result is a dependency tree:



Dependency tree

We can get additional insights into the meaning of attributes using `spacy.explain()`, as here:

```
spacy.explain("VBZ")
verb, 3rd person singular present
```

Batch-processing documents

We will now read a larger set of 2,225 BBC News articles (see GitHub for data source details) that belong to five categories and are stored in individual text files. We need to do the following:

1. Call the `.glob()` method of pathlib's `Path` object.
2. Iterate over the resulting list of paths.
3. Read all lines of the news article excluding the heading in the first line.
4. Append the cleaned result to a list:

```
files = Path('..', 'data', 'bbc').glob('**/*.txt')
bbc_articles = []
for i, file in enumerate(files):
    _, _, _, topic, file_name = file.parts
    with file.open(encoding='latin1') as f:
        lines = f.readlines()
        body = ' '.join([l.strip() for l in lines[1:]]).strip()
        bbc_articles.append(body)
len(bbc_articles)
2225
```

Sentence boundary detection

We will illustrate sentence detection by calling the NLP object on the first of the articles:

```
doc = nlp(bbc_articles[0])
type(doc)
spacy.tokens.doc.Doc
```

`spaCy` computes sentence boundaries from the syntactic parse tree so that punctuation and capitalization play an important but not decisive role. As a result, boundaries will coincide with clause boundaries, even for poorly punctuated text.

We can access parsed sentences using the `.sents` attribute:

```
sentences = [s for s in doc.sents]
sentences[:3]
[Voting is under way for the annual Bloggies which recognize the best web
blogs - online spaces where people publish their thoughts - of the year. ,
Nominations were announced on Sunday, but traffic to the official site was
so heavy that the website was temporarily closed because of too many
visitors.,
Weblogs have been nominated in 30 categories, from the top regional blog,
to the best-kept-secret blog.]
```

Named entity recognition

`spaCy` enables named entity recognition using the `.ent_type_ attribute`:

```
for t in sentences[0]:
    if t.ent_type_:
        print('{} | {} | {}'.format(t.text, t.ent_type_,
spacy.explain(t.ent_type_)))
annual | DATE | Absolute or relative dates or periods
the | DATE | Absolute or relative dates or periods
year | DATE | Absolute or relative dates or periods
```

`textacy` facilitates access to the named entities that appear in the first article:

```
from textacy.extract import named_entities
entities = [e.text for e in named_entities(doc)]
pd.Series(entities).value_counts()
year                          4
US                            2
South-East Asia Earthquake    2
annual                        2
Tsunami Blog                  2
```

N-grams

N-grams combine *N* consecutive tokens. N-grams can be useful for the BoW model because, depending on the textual context, treating something such as data scientist as a single token may be more meaningful than treating it as two distinct tokens: data and scientist.

`textacy` makes it easy to view the `ngrams` of a given length *n* occurring with at least `min_freq` times:

```
from textacy.extract import ngrams
pd.Series([n.text for n in ngrams(doc, n=2, min_freq=2)]).value_counts()
East Asia          2
Asia Earthquake    2
Tsunami Blog       2
annual Bloggies    2
```

spaCy's streaming API

To pass a larger number of documents through the processing pipeline, we can use `spaCy`'s streaming API as follows:

```
iter_texts = (bbc_articles[i] for i in range(len(bbc_articles)))
for i, doc in enumerate(nlp.pipe(iter_texts, batch_size=50, n_threads=8)):
      assert doc.is_parsed
```

Multi-language NLP

`spaCy` includes trained language models for English, German, Spanish, Portuguese, French, Italian, and Dutch, as well as a multi-language model for NER. Cross-language usage is straightforward since the API does not change.

We will illustrate the Spanish language model using a parallel corpus of TED Talk subtitles (see the GitHub repo for data source references). For this purpose, we instantiate both language models:

```
model = {}
for language in ['en', 'es']:
    model[language] = spacy.load(language)
```

We then read small corresponding text samples in each model:

```
text = {}
path = Path('../data/TED')
for language in ['en', 'es']:
    file_name = path / 'TED2013_sample.{}'.format(language)
    text[language] = file_name.read_text()
```

Sentence boundary detection uses the same logic but finds a different breakdown:

```
parsed, sentences = {}, {}
for language in ['en', 'es']:
    parsed[language] = model[language](text[language])
    sentences[language] = list(parsed[language].sents)
print('Sentences:', language, len(sentences[language]))
Sentences: en 19
Sentences: es 22
```

POS tagging also works in the same way:

```
pos = {}
for language in ['en', 'es']:
    pos[language] = pd.DataFrame([[t.text, t.pos_, spacy.explain(t.pos_)]
for t in sentences[language][0]],
    columns=['Token', 'POS Tag', 'Meaning'])
pd.concat([pos['en'], pos['es']], axis=1).head()
```

The result is the side-by-side token annotations for the English and Spanish documents:

Token	**POS Tag**	**Meaning**	**Token**	**POS Tag**	**Meaning**
There	ADV	adverb	Existe	VERB	verb
s	VERB	verb	una	DET	determiner
a	DET	determiner	estrecha	ADJ	adjective
tight	ADJ	adjective	y	CONJ	conjunction
and	CCONJ	coordinating conjunction	sorprendente	ADJ	adjective

The next section illustrates how to use parsed and annotated tokens to build a document-term matrix that can be used for text classification.

NLP with TextBlob

`TextBlob` is a Python library that provides a simple API for common NLP tasks and builds on the **Natural Language Toolkit** (**NLTK**) and the Pattern web mining libraries. `TextBlob` facilitates POS tagging, noun phrase extraction, sentiment analysis, classification, translation, and more.

To illustrate the use of `TextBlob`, we sample a BBC sports article with the headline *Robinson ready for difficult task*. Similarly to `spaCy` and other libraries, the first step is to pass the document through a pipeline represented by the `TextBlob` object to assign annotations required for various tasks (see the `nlp_with_textblob` notebook for this section):

```
from textblob import TextBlob
article = docs.sample(1).squeeze()
parsed_body = TextBlob(article.body)
```

Stemming

To perform stemming, we instantiate `SnowballStemmer` from the `nltk` library, call its `.stem()` method on each token and display modified tokens:

```
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer('english')
[(word, stemmer.stem(word)) for i, word in enumerate(parsed_body.words)
    if word.lower() != stemmer.stem(parsed_body.words[i])]
[('Andy', 'andi'),
('faces', 'face'),
('tenure', 'tenur'),
('tries', 'tri'),
('winning', 'win'),
```

Sentiment polarity and subjectivity

`TextBlob` provides polarity and subjectivity estimates for parsed documents using dictionaries provided by the Pattern library. These dictionaries map adjectives frequently found in product reviews to sentiment polarity scores, ranging from -1 to +1 (negative ↔ positive) and a similar subjectivity score (objective ↔ subjective).

The `.sentiment` attribute provides the average for each over the relevant tokens, whereas the `.sentiment_assessments` attribute lists the underlying values for each token (see notebook):

```
parsed_body.sentiment
Sentiment(polarity=0.088031914893617, subjectivity=0.46456433637284694)
```

From tokens to numbers – the document-term matrix

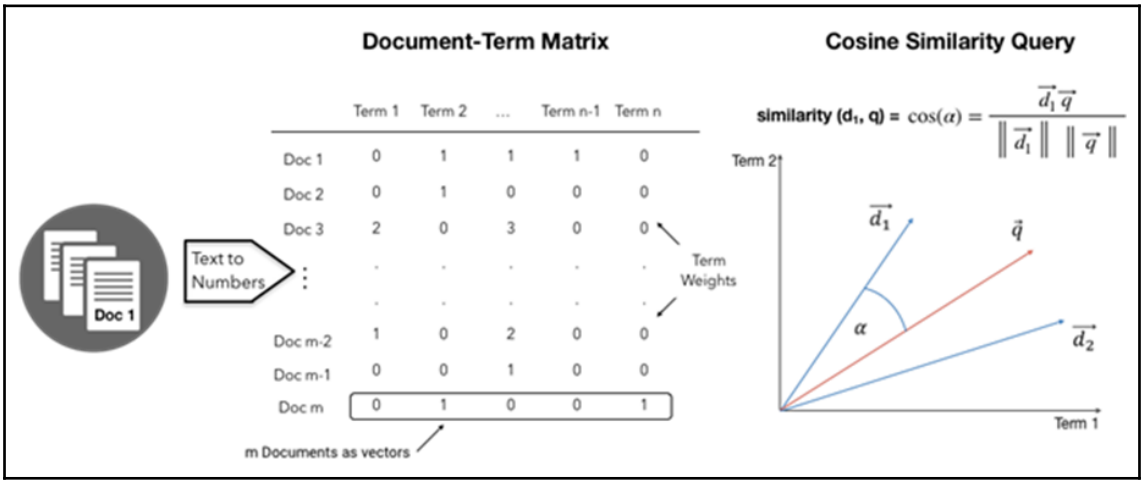
In this section, we first introduce how the BoW model converts text data into a numeric vector space representation that permits the comparison of documents using their distance. We then proceed to illustrate how to create a document-term matrix using the sklearn library.

The BoW model

The BoW model represents a document based on the frequency of the terms or tokens it contains. Each document becomes a vector with one entry for each token in the vocabulary that reflects the token's relevance to the document.

The document-term matrix is straightforward to compute given the vocabulary. However, it is also a crude simplification because it abstracts from word order and grammatical relationships. Nonetheless, it often achieves good results in text classification quickly and, thus, is a very useful starting point.

The following diagram (the one on the right) illustrates how this document model converts text data into a matrix with numerical entries, where each row corresponds to a document and each column to a token in the vocabulary. The resulting matrix is usually both very high-dimensional and sparse; that is, one that contains many zero entries because most documents only contain a small fraction of the overall vocabulary:



Resultant matrix

There are several ways to weigh a token's vector entry to capture its relevance to the document. We will illustrate how to use sklearn to use binary flags, which indicate presence or absence, counts, and weighted counts that account for differences in term frequencies across all documents; that is, in the corpus.

Measuring the similarity of documents

The representation of documents as word vectors assigns to each document a location in the vector space created by the vocabulary. Interpreting vector entries as Cartesian coordinates in this space, we can use the angle between two vectors to measure their similarity because vectors that point in the same direction contain the same terms with the same frequency weights.

The preceding diagram (the one on the right) illustrates—simplified in two dimensions—the calculation of the distance between a document represented by a vector d_1 and a query vector (either a set of search terms or another document) represented by the vector q.

Cosine similarity equals the cosine of the angle between the two vectors. It translates the size of the angle into a number in the range [0, 1] since all vector entries are non-negative token weights. A value of 1 implies that both documents are identical concerning their token weighs, whereas a value of 0 implies that two documents only contain distinct tokens.

As shown in the diagram, the cosine of the angle is equal to the dot product of the vectors; that is, the sum product of their coordinates, divided by the product of the lengths, measured by the Euclidean norms of each vector.

Document-term matrix with sklearn

The scikit-learn preprocessing module offers two tools to create a document-term matrix. `CountVectorizer` uses binary or absolute counts to measure the **term frequency** *tf(d, t)* for each document *d* and token *t*.

`TfidFVectorizer`, in contrast, weighs the (absolute) term frequency by the **inverse document frequency** (**idf**). As a result, a term that appears in more documents will receive a lower weight than a token with the same frequency for a given document but lower frequency across all documents. More specifically, using the default settings, *tf-idf(d, t)* entries for the document-term matrix are computed as *tf-idf(d, t) = tf(d, t) x idf(t)*:

$$\text{idf}(t) = \log \frac{1 + n_d}{1 + \text{df}(d, t)} + 1$$

Here n_d is the number of documents and *df(d, t)* the document frequency of term *t*. The resulting tf-idf vectors for each document are normalized with respect to their absolute or squared totals (see the `sklearn` documentation for details). The tf-idf measure was originally used in information retrieval to rank search engine results and has subsequently proven useful for text classification or clustering.

Both tools use the same interface and perform tokenization and further optional preprocessing of a list of documents before vectorizing the text by generating token counts to populate the document-term matrix.

Key parameters that affect the size of the vocabulary include the following:

- `stop_words`: Use a built-in or provide a list of (frequent) words to exclude
- `ngram_range`: Include n-grams in a range for n defined by a tuple of (n_{min}, n_{max})
- `lowercase`: Convert characters accordingly (default is `True`)
- `min_df` / `max_df`: Ignore words that appear in less / more (`int`) or a smaller/larger share of documents (if `float` [0.0,1.0])
- `max_features`: Limit the number of tokens in a vocabulary accordingly
- `binary`: Set non-zero counts to 1 `True`

See the `document_term_matrix` notebook for the following code samples and additional details. We are again using the 2,225 BBC News articles for illustration.

Using CountVectorizer

The notebook contains an interactive visualization that explores the impact of the `min_df` and `max_df` settings on the size of the vocabulary. We read the articles into a DataFrame, set the `CountVectorizer` to produce binary flags and use all tokens, and call its `.fit_transform()` method to produce a document-term matrix:

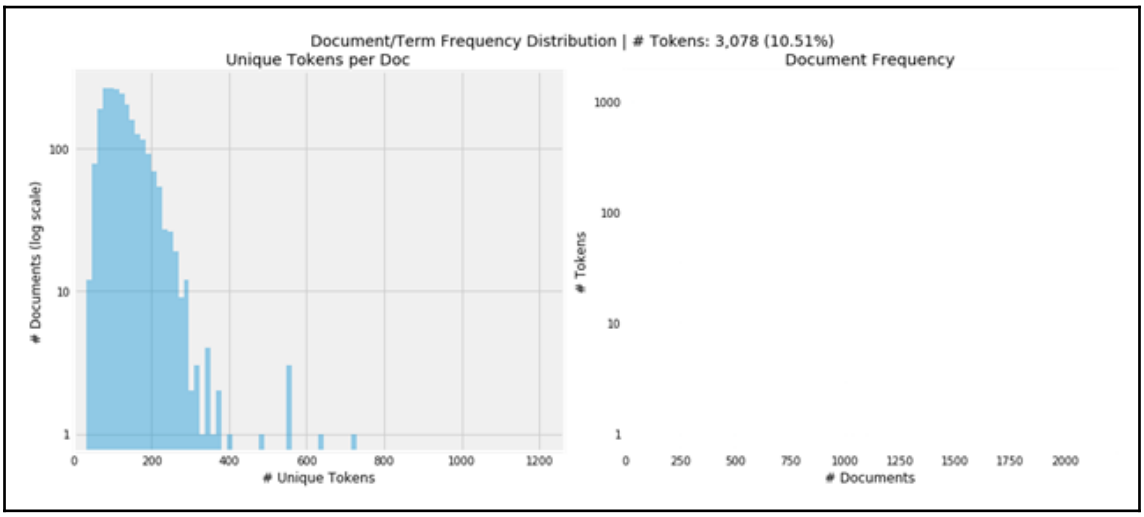
```
binary_vectorizer = CountVectorizer(max_df=1.0,
                                    min_df=1,
                                    binary=True)

binary_dtm = binary_vectorizer.fit_transform(docs.body)
<2225x29275 sparse matrix of type '<class 'numpy.int64'>'
   with 445870 stored elements in Compressed Sparse Row format>
```

The output is a `scipy.sparse` matrix in row format that efficiently stores of the small share (<0.7%) of `445870` non-zero entries in the `2225` (document) rows and `29275` (token) columns.

Visualizing vocabulary distribution

The visualization shows that requiring tokens to appear in at least 1% and fewer than 50% of documents restricts the vocabulary to around 10% of the almost 30,000 tokens.

This leaves a mode of slightly over 100 unique tokens per document (left panel), and the right panel shows the document frequency histogram for the remaining tokens:



Documents/Term frequency distribution

Finding the most similar documents

The `CountVectorizer` result lets us find the most similar documents using the `pdist()` function for pairwise distances provided by the `scipy.spatial.distance` module. It returns a condensed distance matrix with entries corresponding to the upper triangle of a square matrix. We use `np.triu_indices()` to translate the index that minimizes the distance to the row and column indices that in turn correspond to the closest token vectors:

```
m = binary_dtm.todense() # pdist does not accept sparse format
pairwise_distances = pdist(m, metric='cosine')
closest = np.argmin(pairwise_distances) # index that minimizes distance
rows, cols = np.triu_indices(n_docs) # get row-col indices
rows[closest], cols[closest]
(11, 75)
```

Articles number `11` and `75` are closest by cosine similarity because they share 58 tokens (see notebook):

Topic	tech	tech
Heading	Software watching while you work	BT program to beat dialer scams
Body	Software that can not only monitor every keystroke and action performed at a PC but can also be used as legally binding evidence of wrong-doing has been unveiled. Worries about cyber-crime and sabotage have prompted many employers to consider monitoring employees.	BT is introducing two initiatives to help beat rogue dialer scams, which can cost dial-up net users thousands. From May, dial-up net users will be able to download free software to stop computers using numbers not on a user's pre-approved list.

Both `CountVectorizer` and `TfidFVectorizer` can be used with `spaCy`; for example, to perform lemmatization and exclude certain characters during tokenization, we use the following:

```
nlp = spacy.load('en')
def tokenizer(doc):
    return [w.lemma_ for w in nlp(doc)
                if not w.is_punct | w.is_space]
vectorizer = CountVectorizer(tokenizer=tokenizer, binary=True)
doc_term_matrix = vectorizer.fit_transform(docs.body)
```

See the notebook for additional details and more examples.

TfidFTransformer and TfidFVectorizer

`TfidfTransfomer` computes tf-idf weights from a document-term matrix of token counts, such as the one produced by the `CountVectorizer`.

`TfidfVectorizer` performs both computations in a single step. It adds a few parameters to the `CountVectorizer` API that controls smoothing behavior.

TFIDF computation works as follows for a small text sample:

```
sample_docs = ['call you tomorrow',
               'Call me a taxi',
               'please call me... PLEASE!']
```

We compute the term frequency as we just did:

```
vectorizer = CountVectorizer()
tf_dtm = vectorizer.fit_transform(sample_docs).todense()
tokens = vectorizer.get_feature_names()
term_frequency = pd.DataFrame(data=tf_dtm,
                              columns=tokens)

  call  me  please  taxi  tomorrow  you
0    1   0       0     0         1    1
1    1   1       0     1         0    0
2    1   1       2     0         0    0
```

Document frequency is the number of documents containing the token:

```
vectorizer = CountVectorizer(binary=True)
df_dtm = vectorizer.fit_transform(sample_docs).todense().sum(axis=0)
document_frequency = pd.DataFrame(data=df_dtm,
                                  columns=tokens)
   call  me  please  taxi  tomorrow  you
0    3   2       1     1         1    1
```

The tf-idf weights are the ratio of these values:

```
tfidf = pd.DataFrame(data=tf_dtm/df_dtm, columns=tokens)
   call   me  please  taxi  tomorrow  you
0  0.33 0.00    0.00  0.00      1.00 1.00
1  0.33 0.50    0.00  1.00      0.00 0.00
2  0.33 0.50    2.00  0.00      0.00 0.00
```

The effect of smoothing

To avoid zero division, `TfidfVectorizer` uses smoothing for document and term frequencies:

- `smooth_idf`: Add `1` to document frequency, as if an extra document contained every token in the vocabulary, to prevent zero divisions
- `sublinear_tf`: Apply sublinear `tf` scaling; in other words, replace `tf` with `1 + log(tf)`

In combination with normed weights, the results differ slightly:

```
vect = TfidfVectorizer(smooth_idf=True,
                       norm='l2',  # squared weights sum to 1 by
                                     document
                       sublinear_tf=False,  # if True, use 1+log(tf)
                       binary=False)
pd.DataFrame(vect.fit_transform(sample_docs).todense(),
             columns=vect.get_feature_names())
```

```
   call   me  please  taxi  tomorrow  you
0  0.39 0.00    0.00  0.00      0.65 0.65
1  0.43 0.55    0.00  0.72      0.00 0.00
2  0.27 0.34    0.90  0.00      0.00 0.00
```

How to summarize news articles using TfidFVectorizer

Due to their ability to assign meaningful token weights, TFIDF vectors are also used to summarize text data. For instance, Reddit's `autotldr` function is based on a similar algorithm. See the notebook for an example using the BBC articles.

Text Preprocessing - review

The large number of techniques to process natural language for its use in machine learning models that we introduced in this section is necessary to address the complex nature of this highly unstructured data source. The engineering of good language features is both challenging and rewarding and is arguably the most important step in unlocking the semantic value hidden in text data.

In practice, experience helps us select transformations that remove noise rather than the signal, but it will likely remain necessary to cross-validate and compare the performance of different combinations of preprocessing choices.

Text classification and sentiment analysis

Once text data has been converted into numerical features using the NLP techniques discussed in the previous sections, text classification works just like any other classification task.

In this section, we will apply these preprocessing technique to news articles, product reviews, and Twitter data and teach you about various classifiers to predict discrete news categories, review scores, and sentiment polarity.

First, we will introduce the Naive Bayes model, a probabilistic classification algorithm that works well with the text features produced by a bag-of-words model.

The code samples for this section are in the `text_classification` notebook.

The Naive Bayes classifier

The Naive Bayes algorithm is very popular for text classification because low computational cost and memory requirements facilitate training on very large, high-dimensional datasets. Its predictive performance can compete with more complex models, provides a good baseline, and is best known for successful spam detection.

The model relies on Bayes' theorem (see `Chapter 9`, *Bayesian Machine Learning*) and the assumption that the various features are independent of each other given the outcome class. In other words, for a given outcome, knowing the value of one feature (such as the presence of a token in a document) does not provide any information about the value of another feature.

Bayes' theorem refresher

Bayes' theorem expresses the conditional probability of one event (for instance, that an email is spam as opposed to benign ham) given another event (for example, that the email contains certain words), as follows:

$$\underbrace{P(\text{is spam}|\text{has words}}_{\text{Posterior}} = \frac{\overbrace{P(\text{has words} \mid \text{is spam})}^{\text{Likelihood}}\overbrace{P(\text{is spam})}^{\text{Prior}}}{\underbrace{P(\text{has words})}_{\text{Evidence}}}$$

The **posterior** probability that an email is in fact spam, given it contains certain words, depends on the interplay of three factors:

- The **prior** probability that an email is spam
- The **likelihood** of encountering these word in a spam email
- The **evidence**; that is, the probability of seeing these words in an email

To compute the posterior, we can ignore the evidence because it is the same for all outcomes (spam versus ham), and the unconditional prior may be easy to compute.

However, the likelihood poses insurmountable challenges for a reasonably sized vocabulary and a real-world corpus of emails. The reason is the combinatorial explosion of words that did or did not appear jointly in different documents and that prevent the evaluation required to compute a probability table and assign a value to the likelihood.

The conditional independence assumption

The assumption that is making the model both tractable and justifiably calling it Naive is that the features are independent conditional on the outcome. To illustrate, let's classify an email with the three words *Send money now* so that Bayes' theorem becomes the following:

$$P(spam \mid send\ money\ now) = \frac{P(send\ money\ now \mid spam) \times P(spam)}{P(send\ money\ now)}$$

Formally, the assumption that the three words are conditionally independent means that the probability of observing *send* is not affected by the presence of the other terms given the mail is spam; in other words, *P(send | money, now, spam) = P(send | spam)*. As a result, we can simplify the likelihood function:

$$P(spam \mid send\ money\ now) = \frac{P(send \mid spam) \times P(money \mid spam) \times P(now \mid spam) \times P(spam)}{P(send\ money\ now)}$$

Using the naive conditional independence assumption, each term in the numerator is straightforward to compute as relative frequencies from the training data. The denominator is constant across classes and can be ignored when posterior probabilities need to be compared rather than calibrated. The prior probability becomes less relevant as the number of factors—that is, features—increases.

In summary, the advantages of the Naive Bayes model are fast training and prediction because the number of parameters is linear in the number of features, and their estimation has a closed-form solution (based on training data frequencies) rather than expensive iterative optimization. It is also intuitive and somewhat interpretable, does not require hyperparameter tuning, and is relatively robust to irrelevant features given a sufficient signal.

However, when the independence assumption does not hold, and text classification depends on combinations of features or features are correlated, the model will perform poorly.

News article classification

We start with an illustration of the Naive Bayes model for news article classification using the BBC articles that we read as before to obtain a DataFrame with 2,225 articles from five categories:

```
RangeIndex: 2225 entries, 0 to 2224
Data columns (total 3 columns):
topic 2225 non-null object
heading 2225 non-null object
body 2225 non-null object
```

Training and evaluating multinomial Naive Bayes classifier

We split the data into the default 75:25 train-test sets, ensuring that test set classes closely mirror the train set:

```
y = pd.factorize(docs.topic)[0] # create integer class values
X = docs.body
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1,
stratify=y)
```

We proceed to learn the vocabulary from the training set and transform both datasets using `CountVectorizer` with default settings to obtain almost 26,000 features:

```
vectorizer = CountVectorizer()
X_train_dtm = vectorizer.fit_transform(X_train)
X_test_dtm = vectorizer.transform(X_test)
X_train_dtm.shape, X_test_dtm.shape
((1668, 25919), (557, 25919))
```

Training and prediction follow the standard `sklearn` fit/predict interface:

```
nb = MultinomialNB()
nb.fit(X_train_dtm, y_train)
y_pred_class = nb.predict(X_test_dtm)
```

We evaluate multiclass predictions using `accuracy` and find that the default classifier achieved almost 98%:

```
accuracy_score(y_test, y_pred_class)
0.97666068222621
```

Sentiment analysis

Sentiment analysis is one of the most popular uses of NLP and machine learning for trading because positive or negative perspectives on assets or other price drivers are likely to impact returns.

Generally, modeling approaches to sentiment analysis rely on dictionaries, such as the `TextBlob` library, or models that are trained on outcomes for a specific domain. The latter is preferable because it permits more targeted labeling; for instance, by tying text features to subsequent price changes rather than indirect sentiment scores.

We will illustrate machine learning for sentiment analysis using a Twitter dataset with binary polarity labels, and a large Yelp business review dataset with a five-point outcome scale.

Twitter data

We use a dataset that contains 1.6 million training and 350 test tweets from 2009 with algorithmically assigned binary positive and negative sentiment scores that are fairly evenly split (see the relevant notebook for more detailed data exploration).

Multinomial Naive Bayes

We create a document-term matrix with 934 tokens as follows:

```
vectorizer = CountVectorizer(min_df=.001, max_df=.8, stop_words='english')
train_dtm = vectorizer.fit_transform(train.text)
<1566668x934 sparse matrix of type '<class 'numpy.int64'>'
   with 6332930 stored elements in Compressed Sparse Row format>
```

We then train the `MultinomialNB` classifier as before and predict the test set:

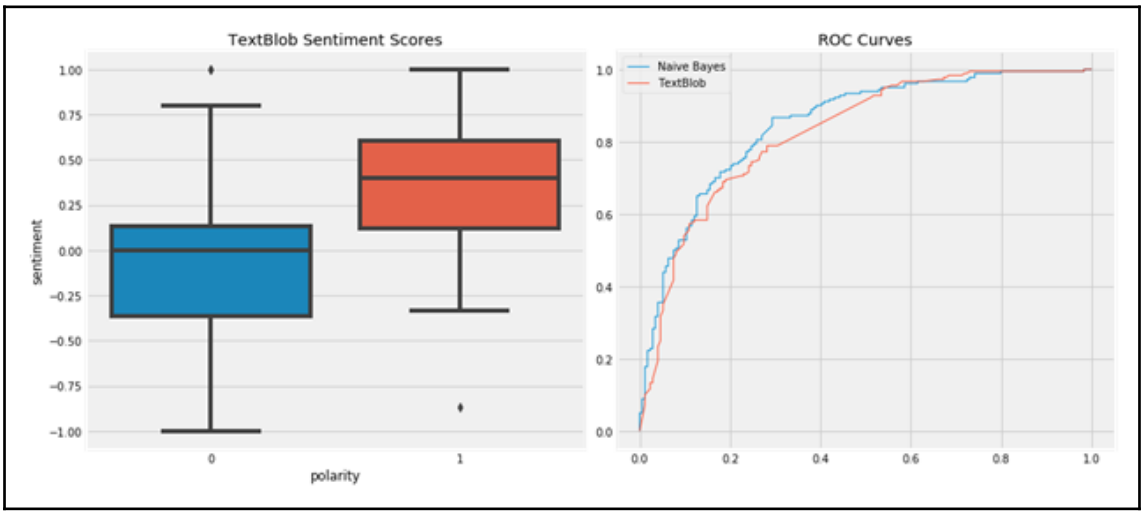
```
nb = MultinomialNB()
nb.fit(train_dtm, train.polarity)
predicted_polarity = nb.predict(test_dtm)
```

The result is over 77.5% accuracy:

```
accuracy_score(test.polarity, y_pred_class)
0.7768361581920904
```

Comparison with TextBlob sentiment scores

We also obtain `TextBlob` sentiment scores for tweets and note (see the following left-hand diagram) that positive test tweets receive a significantly higher sentiment estimate. We then use the `MultinomialNB` model and the `.predict_proba()` method to compute predicted probabilities and compare both models using the respective Area Under the Curve (see the following right-hand diagram):



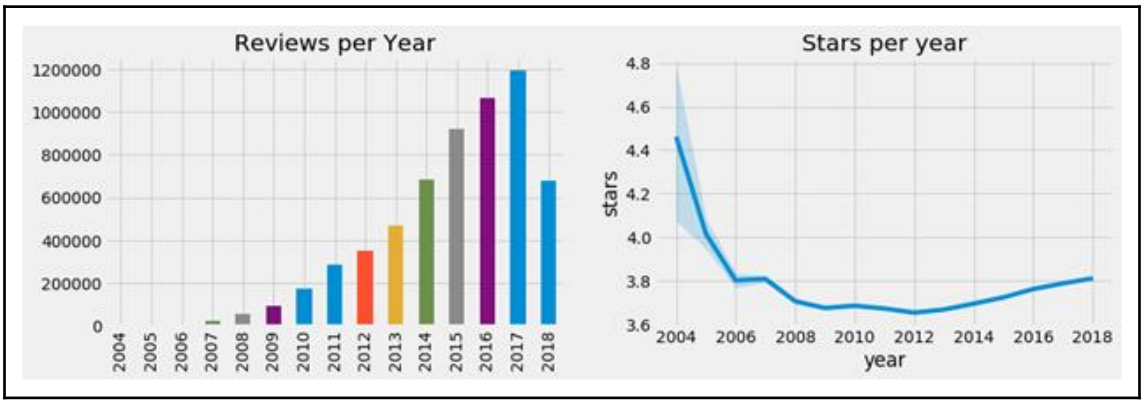
TextBlob sentiment scores

The Naive Bayes model outperforms `TextBlob` in this case.

Business reviews – the Yelp dataset challenge

Finally, we apply sentiment analysis to the significantly larger Yelp business review dataset with five outcome classes. The data consists of several files with information on the business, the user, the review, and other aspects that Yelp provides to encourage data science innovation.

We will use around six million reviews produced over the 2010-2018 period (see the relevant notebook for details). The following diagrams show the number of reviews and the average number of stars per year:



Graphs representing number of reviews and the average number of stars per year

In addition to the text features resulting from the review texts, we will also use other information submitted with the review or about the user.

We will train various models on data through 2017 and use 2018 as the test set.

Benchmark accuracy

Using the most frequent number of stars (=5) to predict the test set, we achieve an accuracy close to 52%:

```
test['predicted'] = train.stars.mode().iloc[0]
accuracy_score(test.stars, test.predicted)
0.5196950594793454
```

Multinomial Naive Bayes model

Next, we train a Naive Bayes classifier using a document-term matrix produced by `CountVectorizer` with default settings:

```
nb = MultinomialNB()
nb.fit(train_dtm,train.stars)
predicted_stars = nb.predict(test_dtm)
```

The prediction produces 64.7% accuracy on the test set, a 24.4% improvement over the benchmark:

```
accuracy_score(test.stars, predicted_stars)
0.6465164206691094
```

One-versus-all logistic regression

We proceed to train a one-versus-all logistic regression that trains one model per class, while treating the remaining classes as the negative class, and predicts probabilities for each class using the different models.

Using only text features, we train and evaluate the model as follows:

```
logreg = LogisticRegression(C=1e9)
logreg.fit(X=train_dtm, y=train.stars)
y_pred_class = logreg.predict(test_dtm)
```

The model achieves significantly higher accuracy at 73.6%:

```
accuracy_score(test.stars, y_pred_class)
0.7360498864740219
```

Combining text and numerical features

The dataset contains various numerical features (see the relevant notebook for implementation details).

Vectorizers produce `scipy.sparse` matrices. To combine vectorized text data with other features, we need to first convert these to sparse matrices as well; many sklearn objects and other libraries, such as LightGBM, can handle these very memory-efficient data structures. Converting the sparse matrix to a dense NumPy array risks memory overflow.

Most variables are categorical, so we use one-hot encoding since we have a fairly large dataset to accommodate the increase in features.

We convert the encoded numerical features and combine them with the document-term matrix:

```
train_numeric = sparse.csr_matrix(train_dummies.astype(np.int8))
train_dtm_numeric = sparse.hstack((train_dtm, train_numeric))
```

Multinomial logistic regression

Logistic regression also provides a multinomial training option that is faster and more accurate than the one-versus-all implementation. We use the `lbfgs` solver (see the sklearn documentation linked on GitHub for details):

```
multi_logreg = LogisticRegression(C=1e9, multi_class='multinomial',
                                  solver='lbfgs')
multi_logreg.fit(train_dtm_numeric.astype(float), train.stars)
y_pred_class = multi_logreg.predict(test_dtm_numeric.astype(float))
```

This model improves the performance to 74.6% accuracy:

```
accuracy_score(test.stars, y_pred_class)
0.7464488070176475
```

In this case, tuning the regularization parameter `C` did not lead to very significant improvements (see the notebook).

Gradient-boosting machine

For illustration purposes, we also train a LightGBM gradient-boosting tree ensemble with default settings and the `multiclass` objective:

```
param = {'objective':'multiclass', 'num_class': 5}
booster = lgb.train(params=param,
                    train_set=lgb_train,
                    num_boost_round=500,
                    early_stopping_rounds=20,
                    valid_sets=[lgb_train, lgb_test])
```

The basic settings do not improve on multinomial logistic regression, but further parameter tuning remains an unused option:

```
y_pred_class = booster.predict(test_dtm_numeric.astype(float))
accuracy_score(test.stars, y_pred_class.argmax(1) + 1)
0.738665855696524
```

Summary

In this chapter, we explored numerous techniques and options to process unstructured data with the goal of extracting semantically meaningful, numerical features for use in machine learning models.

We covered the basic tokenization and annotation pipeline and illustrated its implementation for multiple languages using spaCy and TextBlob. We built on these results to create a document model based on the bag-of-words model to represent documents as numerical vectors. We learned how to refine the preprocessing pipeline and then used vectorized text data for classification and sentiment analysis.

In the remaining two chapters on alternative text data, we will learn how to summarize text using unsupervised learning to identify latent topics (in the next chapter) and examine techniques to represent words as vectors that reflect the context of word usage and have been used very successfully to proceed richer text features for various classification tasks.