14
Text Data for Trading – Sentiment Analysis

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 but highly unstructured, so it requires more preprocessing to enable an ML algorithm to extract relevant information. A key challenge consists of converting text into a numerical format without losing its meaning. We will cover several techniques capable of capturing the nuances of language so that they can be used as input for ML algorithms.

In this chapter, we will 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 and then proceed to use it as input for **news 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 embeddings to capture the 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
- Performing NLP tasks such as **part-of-speech (POS)** tagging or named entity recognition
- Converting tokens to numbers using the document-term matrix
- Classifying text using the naive Bayes model
- How to perform sentiment analysis

> You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

ML with text data – from language to features

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 financial investments range from formal documents like company statements, contracts, and patents, to news, opinion, and analyst research or commentary, to various types of social media postings or messages.

Numerous and diverse text data samples are available online to explore the use of NLP algorithms, many of which are listed among the resources included in this chapter's `README` file on GitHub. For a comprehensive introduction, see Jurafsky and Martin (2008).

To realize the potential value of text data, we'll introduce the specialized NLP techniques and the most effective Python libraries, outline key challenges particular to working with language data, introduce critical elements of the NLP workflow, and highlight NLP applications relevant for algorithmic trading.

Key challenges of working with text data

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

NLP is particularly 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 having different meanings depending on context ("Local High School Dropouts Cut in Half")
- The nonstandard and **evolving use** of language, especially on social media
- The use of **idioms** like "throw in the towel"
- Tricky **entity names** like "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 for 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, for example, a company report, a headline, a news article, or a tweet. A corpus, in turn, is a collection of documents.

Figure 14.1 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:

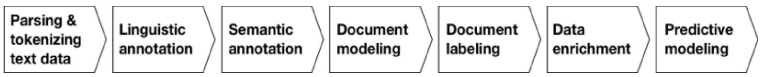

Figure 14.1: The NLP workflow

Fundamental techniques extract text features as isolated semantic units called tokens and use rules and dictionaries to annotate them with linguistic and semantic information. The bag-of-words model uses token frequency to model documents as token vectors, which leads to the document-term matrix that is frequently used for text classification, retrieval, or summarization.

Advanced approaches rely on ML to refine basic features such as tokens and produce richer document models. These include topic models that reflect the joint usage of tokens across documents and word-vector models that aim to capture the context of token usage.

We will review key decisions at each step of the workflow and the related tradeoffs 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	Segment text into words, punctuation marks, and so on.
Part-of-speech tagging	Assign word types to tokens, such as a verb or noun.
Dependency parsing	Label syntactic token dependencies, like subject <=> object.
Stemming and lemmatization	Assign the base forms of words: "was" => "be", "rats" => "rat".
Sentence boundary detection	Find and segment individual sentences.
Named entity recognition	Label "real-world" objects, such as people, companies, or locations.
Similarity	Evaluate the similarity of words, text spans, and documents.

Parsing and tokenizing text data – selecting the vocabulary

A token is an instance of a sequence of characters in a given document and is considered a semantic unit. The vocabulary is the set of tokens contained in a corpus deemed relevant for further processing; tokens not in the vocabulary will be ignored.

The **goal**, of course, is to extract tokens that most accurately reflect the document's meaning. The **key tradeoff** at this step is the choice of a larger vocabulary to better reflect the text source at the expense of more features and higher model complexity (discussed as the *curse of dimensionality* in *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*).

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.

In addition, we need to decide whether to include groupings of *n* individual tokens called ***n*-grams** as semantic units (an individual token is also called *unigram*). An example of a two-gram (or *bigram*) is "New York", whereas "New York City" is a three-gram (or *trigram*). The decision can rely on dictionaries or a comparison of the relative frequencies of the individual and joint usage. There are more unique combinations of tokens than unigrams, hence adding *n*-grams will drive up the number of features and risks adding noise unless filtered for by frequency.

Linguistic annotation – relationships among tokens

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 and relationships in a sentence for POS tagging and dependency parsing. It also permits the identification of common root forms for stemming and lemmatization to group together related words.

The following are some key concepts related to annotations:

- **Stemming** uses simple rules to remove common endings such as *s*, *ly*, *ing*, or *ed* from a token and reduce it to its stem or root form.
- **Lemmatization** uses more sophisticated rules to derive the canonical root (**lemma**) of a word. It can detect irregular common roots such as "better" and "best" and more effectively condenses the vocabulary but is slower than stemming. Both approaches simplify the vocabulary at the expense of semantic nuances.
- **POS** annotations help disambiguate tokens based on their function (for example, when a verb and noun have the same form), which increases the vocabulary but may capture meaningful distinctions.
- **Dependency parsing** identifies hierarchical relationships among tokens and is commonly used for translation. It is important for interactive applications that require more advanced language understanding, such as chatbots.

Semantic annotation – from entities to knowledge graphs

Named-entity recognition (**NER**) aims at identifying tokens that represent objects of interest, such as persons, 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 at predicting the impact of news events on sentiment.

Labeling – assigning outcomes for predictive modeling

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

Examples include decisions on how to quantify the sentiment implicit in a text document such as an email, transcribed interview, or tweet with respect to a new domain, or which aspects of a research document or news report should be assigned a specific outcome.

Applications

The use of ML with text data for trading relies on extracting meaningful information in the form of features that help predict future price movements. Applications range from the exploitation of the short-term market impact of news to the longer-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 notebook `nlp_pipeline_with_spaCy` for the following code samples, installation instruction, 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 the tokenization and annotation engines requires the installation of language models. The features we will use in this chapter only require the small models;

the larger models also include word vectors that we will cover in *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*.

With the library installed and linked, we can instantiate a spaCy language model and then apply it to the document. The result is 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

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

- `.text` : The original word text
- `.lemma_` : The word root
- `.pos_` : A basic POS tag
- `.tag_` : The detailed POS tag
- `.dep_` : The syntactic relationship or dependency between tokens
- `.shape_` : The shape of the word, in terms of capitalization, punctuation, and digits
- `.is alpha` : Checks whether the token is alphanumeric
- `.is stop` : Checks 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'])
```

This produces the following result:

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 the syntactic dependency in a browser or notebook using the following:

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

The preceding code allows us to obtain a dependency tree like the following one:

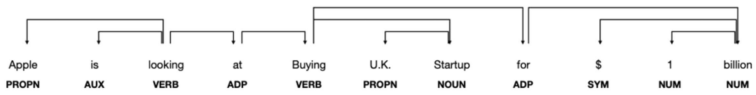

Figure 14.2: spaCy dependency tree

We can get additional insight into the meaning of attributes using `spacy.explain()`, such as the following, for example:

```
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 the data source details) that belong to five categories and are stored in individual text files. We do the following:

1. Call the `.glob()` method of the `pathlib` module'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(sorted(list(files))):
    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 the parsed sentences using the `.sents` attribute:

```
sentences = [s for s in doc.sents]
sentences[:3]
[Quarterly profits at US media giant TimeWarner jumped 76% to $1.13bn (Â£600m) for the three month
 The firm, which is now one of the biggest investors in Google, benefited from sales of high-speed
 TimeWarner said fourth quarter sales rose 2% to $11.1bn from $10.9bn.]
```

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_)))
Quarterly | DATE | Absolute or relative dates or periods
US | GPE | Countries, cities, states
TimeWarner | ORG | Companies, agencies, institutions, etc.
```

Textacy makes access to the named entities that appear in the first article easy:

```
entities = [e.text for e in entities(doc)]
pd.Series(entities).value_counts().head()
TimeWarner        7
```

```
AOL               5
fourth quarter    3
year-earlier      2
one               2
```

N-grams

N-grams combine *n* consecutive tokens. This can be useful for the bag-of-words model because, depending on the textual context, treating (for example) "data scientist" as a single token may be more meaningful than the two distinct tokens "data" and "scientist".

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

```
pd.Series([n.text for n in ngrams(doc, n=2, min_freq=2)]).value_counts()
fourth quarter     3
quarter profits    2
Time Warner        2
company said       2
AOL Europe         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 named-entity recognition. 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 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 22
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()
```

This produces the following table:

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 the 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, and translation, among others.

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

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

Stemming

To perform stemming, we instantiate the `SnowballStemmer` from the NTLK library, call its `.stem()` method on each token, and display tokens that were modified as a result:

```
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])]
[('Manchester', 'manchest'),
 ('United', 'unit'),
 ('reduced', 'reduc'),
 ('points', 'point'),
 ('scrappy', 'scrappi')
```

Sentiment polarity and subjectivity

TextBlob provides polarity and subjectivity estimates for parsed documents using dictionaries provided by the Pattern library. These dictionaries lexicon-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 score over the relevant tokens, whereas the `.sentiment_assessments` attribute lists the underlying values for each token (see the notebook):

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

Counting tokens – the document-term matrix

In this section, we first introduce how the bag-of-words model converts text data into a numeric vector space representations. The goal is to approximate document similarity by their distance in that space. We then proceed to illustrate how to create a document-term matrix using the sklearn library.

The bag-of-words model

The bag-of-words 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.

Creating the document-term matrix

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, provides a very useful starting point.

The left panel of *Figure 14.3* 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, it contains many zero entries because most documents only contain a small fraction of the overall vocabulary.

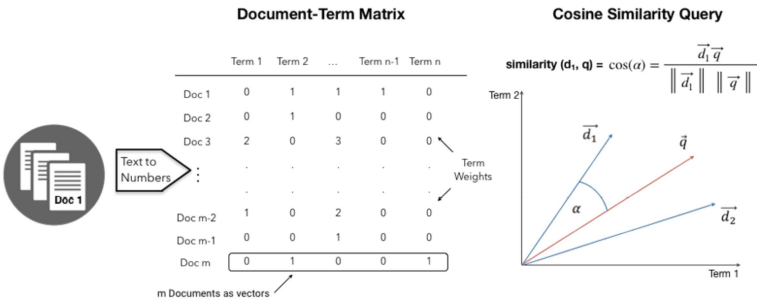

Figure 14.3: Document-term matrix and cosine similarity

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 that indicate presence or absence, counts, and weighted counts that account for differences in term frequencies across all documents 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 the 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 right panel of the preceding figure 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.

The **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 with respect to their token weights, whereas a value of 0 implies that the two documents only contain distinct tokens.

As shown in the figure, 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 scikit-learn

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**) *tf*(*d*, *t*) for each document *d* and token *t*.

`TfidfVectorizer`, by 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 with a lower frequency across all documents. More specifically, using the default settings, the *tf-idf*(*d*, *t*) entries for the document-term matrix are computed as *tf-idf(d, t) = tf(d, t)* x *idf(t)* with:

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

where 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 and 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` : Uses a built-in or user-provided list of (frequent) words to exclude
- `ngram_range` : Includes *n*-grams in a range of *n* defined by a tuple of (n_{min}, n_{max})
- `lowercase` : Converts characters accordingly (the default is `True`)
- `min_df` / `max_df` : Ignores words that appear in less/more (`int`) or are present in a smaller/larger share of documents (if `float` [0.0,1.0])

- `max_features` : Limits the number of tokens in vocabulary accordingly
- `binary` : Sets non-zero counts to 1 (`True`)

See the notebook `document_term_matrix` 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 `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 a small share (<0.7 percent) of the 445,870 non-zero entries in the 2,225 (document) rows and 29,275 (token) columns.

Visualizing the vocabulary distribution

The visualization in *Figure 14.4* shows that requiring tokens to appear in at least 1 percent and less than 50 percent of documents restricts the vocabulary to around 10 percent of the almost 30,000 tokens.

This leaves a mode of slightly over 100 unique tokens per document, as shown in the left panel of the following plot. The right panel shows the document frequency histogram for the remaining tokens:

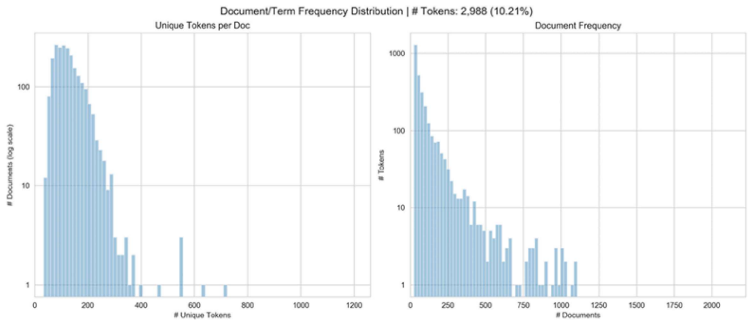

Figure 14.4: The distributions of unique tokens and number of tokens per document

Finding the most similar documents

`CountVectorizer` result lets us find the most similar documents using the `pdist()` functions 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]
(6, 245)
```

Articles 6 and 245 are closest by cosine similarity, due to the fact that they share 38 tokens out of a combined vocabulary of 303 (see notebook). The following table summarizes these two articles and demonstrates the limited ability of similarity measures based on word counts to identify deeper semantic similarity:

	Article 6	Article 245
Topic	Business	Business
Heading	Jobs growth still slow in the US	Ebbers 'aware' of WorldCom fraud
Body	The US created fewer jobs than expected in January, but a fall in jobseekers pushed the unemployment rate to its lowest level in three years. According to Labor Department figures, US firms added only 146,000 jobs in January.	Former WorldCom boss Bernie Ebbers was directly involved in the $11bn financial fraud at the firm, his closest associate has told a US court. Giving evidence in the criminal trial of Mr Ebbers, ex-finance chief Scott Sullivan implicated his colleague in the accounting scandal at the firm.

Both `CountVectorizer` and `TfidfVectorizer` can be used with spaCy, for example, to perform lemmatization and exclude certain characters during tokenization:

```
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 detail and more examples.

TfidfTransformer and TfidfVectorizer

`TfidfTransformer` computes the TF-IDF weights from a document-term matrix of token counts like the one produced by `CountVectorizer`.

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

The 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 before:

```
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
```

The 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` : Adds one to document frequency, as if an extra document contained every token in the vocabulary, to prevent zero divisions
- `sublinear_tf` : Applies sublinear tf scaling, that is, replaces 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
```

Summarizing news articles using TfidfVectorizer

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

Key lessons instead of lessons learned

The large number of techniques and options to process natural language for use in ML models corresponds to the complex nature of this highly unstructured data source. The engineering of good language features is both challenging and rewarding, and arguably the most important step in unlocking the semantic value hidden in text data.

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

NLP for trading

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 techniques to news articles, product reviews, and Twitter data and teach 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 notebook `news_text_classification`.

The naive Bayes classifier

The naive Bayes algorithm is very popular for text classification because its 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 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 (for example, 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 example, that an email is spam as opposed to benign "ham") given another event (for example, that the email contains certain words) as follows:

$$\underbrace{P(\text{is spam} \mid \text{has words})}_{\text{Posterior}} = \frac{\overbrace{P(\text{has words} \mid \text{is spam})}^{\text{Likelihood}} \overbrace{P(\text{is spam})}^{\text{Prior}}}{\underbrace{P(\text{has words})}_{\text{Evidence}}} \%0$$

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

- The **prior** probability that an email is spam
- The **likelihood** of encountering these words 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 the words that did or did not appear jointly in different documents that prevent the evaluation required to compute a probability table and assign a value to the likelihood.

The conditional independence assumption

The key assumption to make the model both tractable and earn it the name *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:

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

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 that the mail is spam, that is, $P(\text{send} \mid \text{money, now, spam}) = P(\text{send} \mid \text{spam})$. As a result, we can simplify the likelihood function:

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

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 sum, the advantages of the naive Bayes model are fast training and prediction because the number of parameters is proportional to 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 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.

Classifying news articles

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

```
<class 'pandas.core.frame.DataFrame'>
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
```

To train and evaluate a multinomial naive Bayes classifier, we split the data into the default 75:25 train-test set ratio, ensuring that the 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 the multiclass predictions using `accuracy` to find the default classifier achieved an accuracy of almost 98 percent:

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

Sentiment analysis with Twitter and Yelp data

Sentiment analysis is one of the most popular uses of NLP and ML 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 (as does the TextBlob library) or models trained on outcomes for a specific domain. The latter is often preferable because it permits more targeted labeling, for example, by tying text features to subsequent price changes rather than indirect sentiment scores.

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

Binary sentiment classification with 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 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 has over 77.5 percent accuracy:

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

Comparison with TextBlob sentiment scores

We also obtain TextBlob sentiment scores for the tweets and note (see the left panel in *Figure 14.5*) that positive test tweets receive a significantly higher sentiment estimate. We then use the `MultinomialNB` model's `.predict_proba()` method to compute predicted probabilities and compare both models using the respective area **under the curve**, or **AUC**, that we introduced in *Chapter 6*, *The Machine Learning Process* (see the right panel in *Figure 14.5*).

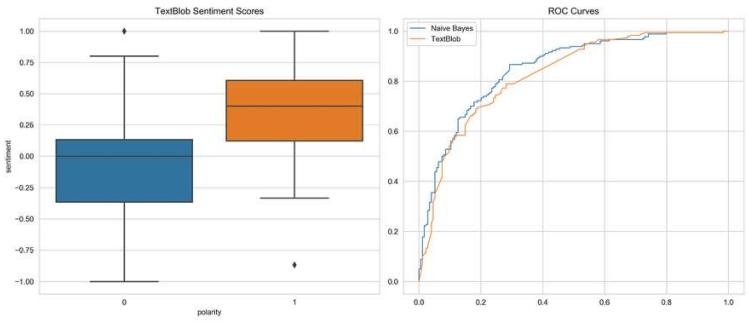

Figure 14.5: Accuracy of custom versus generic sentiment scores

The custom naive Bayes model outperforms TextBlob in this case, achieving a test AUC of 0.848 compared to 0.825 for TextBlob.

Multiclass sentiment analysis with Yelp business reviews

Finally, we apply sentiment analysis to the significantly larger Yelp business review dataset with five outcome classes (see the notebook `sentiment_analysis_yelp` for code and additional details).

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 notebook for details). The following figure shows the number of reviews and the average number of stars per year, as well as the star distribution across all reviews.

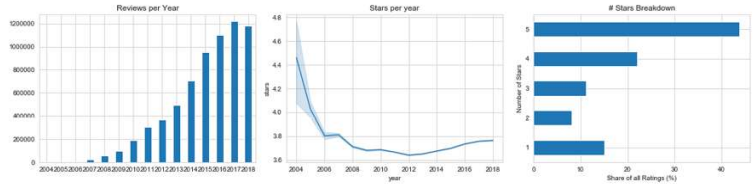

Figure 14.6: Basic exploratory analysis of Yelp reviews

We will train various models on a 10 percent sample of the data through 2017 and use the 2018 reviews as the test set. In addition to the text features resulting from the review texts, we will also use other information submitted with the review about the given user.

Combining text and numerical features

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

The vectorizers produce `scipy.sparse` matrices. To combine the 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.uint))
train_dtm_numeric = sparse.hstack((train_dtm, train_numeric))
```

Benchmark accuracy

Using the most frequent number of stars (=5) to predict the test set achieves an accuracy close to 52 percent:

```
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 percent accuracy on the test set, a 24.4 percent improvement over the benchmark:

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

Training with the combination of text and other features improves the test accuracy to 0.671.

Logistic regression

In *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*, we introduced binary logistic regression. sklearn also implements a multiclass model with a multinomial and a one-versus-all training option, where the latter trains a binary model for each class while considering all other classes as the negative class. The multinomial option is much faster and more accurate than the one-versus-all implementation.

We evaluate a range of values for the regularization parameter `C` to identify the best performing model, using the `lbfgs` solver as follows (see the sklearn documentation for details):

```
def evaluate_model(model, X_train, X_test, name, store=False):
    start = time()
    model.fit(X_train, train.stars)
    runtime[name] = time() - start
    predictions[name] = model.predict(X_test)
    accuracy[result] = accuracy_score(test.stars, predictions[result])
    if store:
        joblib.dump(model, f'results/{result}.joblib')
Cs = np.logspace(-5, 5, 11)
for C in Cs:
    model = LogisticRegression(C=C, multi_class='multinomial', solver='lbfgs')
    evaluate_model(model, train_dtm, test_dtm, result, store=True)
```

Figure 14.7 shows the plots of the validation results.

Multiclass gradient boosting with LightGBM

For comparison, we also train a LightGBM gradient boosting tree ensemble with default settings and a `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])
```

Predictive performance

Figure 14.7 displays the accuracy of each model for the combined data. The right panel plots the validation performance for the logistic regression models for both datasets and different levels of regularization.

Multinomial logistic regression performs best with a test accuracy slightly above 74 percent. Naive Bayes performs significantly worse. The default LightGBM settings did not improve over the linear model with an accuracy of 0.736. However, we could tune the hyperparameters of the gradient boosting model and may well see performance improvements that put it at least on par with logistic regression. Either way, the result serves as a reminder not to discount simple, regularized models as they may deliver not only good results, but also do so quickly.

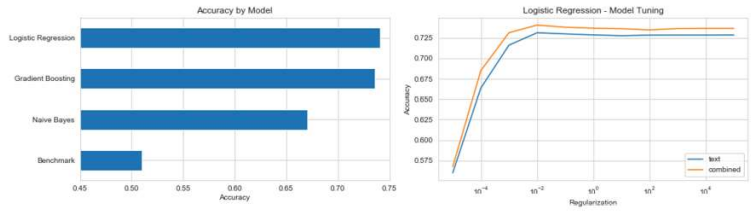

Figure 14.7: Test performance on combined data (all models, left) and for logistic regression with varying regularization

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 ML 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 build 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 the vectorized text data for classification and sentiment analysis.

We have two more chapters on alternative text data. In the next chapter, we will learn how to summarize texts using unsupervised learning to identify latent topics. Then, in *Chapter 16*, *Word Embeddings for Earnings Calls and SEC Filings*, we will learn how to represent words as vectors that reflect the context of word usage, a technique that has been used very successfully to provide richer text features for various classification tasks.