8

Applying Machine Learning to Sentiment Analysis

In this internet and social media age, people's opinions, reviews, and recommendations have become a valuable resource for political science and businesses. Thanks to modern technologies, we are now able to collect and analyze such data most efficiently. In this chapter, we will delve into a subfield of **Natural Language Processing** (**NLP**) called **sentiment analysis** and learn how to use machine learning algorithms to classify documents based on their polarity: the attitude of the writer. In particular, we are going to work with a dataset of 50,000 movie reviews from the **Internet Movie Database** (**IMDb**) and build a predictor that can distinguish between positive and negative reviews.

The topics that we will cover in the following sections include the following:

- Cleaning and preparing text data
- Building feature vectors from text documents
- Training a machine learning model to classify positive and negative movie reviews
- Working with large text datasets using out-of-core learning
- Inferring topics from document collections for categorization

Preparing the IMDb movie review data for text processing

Sentiment analysis, sometimes also called **opinion mining**, is a popular subdiscipline of the broader field of NLP; it is concerned with analyzing the polarity of documents. A popular task in sentiment analysis is the classification of documents based on the expressed opinions or emotions of the authors with regard to a particular topic.

In this chapter, we will be working with a large dataset of movie reviews from the IMDb that has been collected by Maas and others (*Learning Word Vectors for Sentiment Analysis*, *A. L. Maas*, *R. E. Daly*, *P. T. Pham*, *D. Huang*, *A. Y. Ng*, and *C. Potts*, *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, Association for Computational Linguistics, *June 2011*). The movie review dataset consists of 50,000 polar movie reviews that are labeled as either positive or negative; here, positive means that a movie was rated with more than six stars on IMDb, and negative means that a movie was rated with fewer than five stars on IMDb. In the following sections, we will download the dataset, preprocess it into a useable format for machine learning tools, and extract meaningful information from a subset of these movie reviews to build a machine learning model that can predict whether a certain reviewer liked or disliked a movie.

Obtaining the movie review dataset

A compressed archive of the movie review dataset (84.1 MB) can be downloaded from `http://ai.stanford.edu/~amaas/data/sentiment/` as a Gzip-compressed tarball archive:

- If you are working with Linux or macOS, you can open a new Terminal window, `cd` into the download directory, and execute `tar -zxf aclImdb_v1.tar.gz` to decompress the dataset.
- If you are working with Windows, you can download a free archiver such as 7Zip (`http://www.7-zip.org`) to extract the files from the download archive.
- Alternatively, you can directly unpack the Gzip-compressed tarball archive directly in Python as follows:

```
>>> import tarfile
>>> with tarfile.open('aclImdb_v1.tar.gz', 'r:gz') as tar:
...     tar.extractall()
```

Preprocessing the movie dataset into more convenient format

Having successfully extracted the dataset, we will now assemble the individual text documents from the decompressed download archive into a single CSV file. In the following code section, we will be reading the movie reviews into a pandas `DataFrame` object, which can take up to 10 minutes on a standard desktop computer. To visualize the progress and estimated time until completion, we will use the **Python Progress Indicator** (**PyPrind**, `https://pypi.python.org/pypi/PyPrind/`) package that I developed several years ago for such purposes. PyPrind can be installed by executing the `pip install pyprind` command.

```
>>> import pyprind
>>> import pandas as pd
>>> import os

>>> # change the `basepath` to the directory of the
>>> # unzipped movie dataset

>>> basepath = 'aclImdb'
>>>
>>> labels = {'pos': 1, 'neg': 0}
>>> pbar = pyprind.ProgBar(50000)
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
...     for l in ('pos', 'neg'):
...         path = os.path.join(basepath, s, l)
...         for file in sorted(os.listdir(path)):
...             with open(os.path.join(path, file),
...                       'r', encoding='utf-8') as infile:
...                 txt = infile.read()
...             df = df.append([[txt, labels[l]]],
...                            ignore_index=True)
...             pbar.update()
>>> df.columns = ['review', 'sentiment']
0%                            100%
[##############################] | ETA: 00:00:00
Total time elapsed: 00:03:37
```

Introducing the bag-of-words model

You may remember from *Chapter 4, Building Good Training Sets – Data Preprocessing*, that we have to convert categorical data, such as text or words, into a numerical form before we can pass it on to a machine learning algorithm. In this section, we will introduce the **bag-of-words**, which allows us to represent text as numerical feature vectors. The idea behind the bag-of-words model is quite simple and can be summarized as follows:

1. We create a vocabulary of unique tokens—for example, words—from the entire set of documents.
2. We construct a feature vector from each document that contains the counts of how often each word occurs in the particular document.

Since the unique words in each document represent only a small subset of all the words in the bag-of-words vocabulary, the feature vectors will mostly consist of zeros, which is why we call them **sparse**. Do not worry if this sounds too abstract; in the following subsections, we will walk through the process of creating a simple bag-of-words model step-by-step.

Transforming words into feature vectors

To construct a bag-of-words model based on the word counts in the respective documents, we can use the `CountVectorizer` class implemented in scikit-learn. As we will see in the following code section, `CountVectorizer` takes an array of text data, which can be documents or sentences, and constructs the bag-of-words model for us:

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer()
>>> docs = np.array([
...         'The sun is shining',
...         'The weather is sweet',
...         'The sun is shining, the weather is sweet,'
...         'and one and one is two'])
>>> bag = count.fit_transform(docs)
```

The sequence of items in the bag-of-words model that we just created is also called the **1-gram** or **unigram** model—each item or token in the vocabulary represents a single word. More generally, the contiguous sequences of items in NLP—words, letters, or symbols—are also called **n-grams**. The choice of the number *n* in the n-gram model depends on the particular application; for example, a study by Kanaris and others revealed that n-grams of size 3 and 4 yield good performances in anti-spam filtering of email messages (*Words versus character n-grams for anti-spam filtering*, *Ioannis Kanaris*, *Konstantinos Kanaris*, *Ioannis Houvardas*, and *Efstathios Stamatatos*, *International Journal on Artificial Intelligence Tools*, *World Scientific Publishing Company*, 16(06): 1047-1067, *2007*). To summarize the concept of the n-gram representation, the 1-gram and 2-gram representations of our first document "the sun is shining" would be constructed as follows:

- 1-gram: "the", "sun", "is", "shining"
- 2-gram: "the sun", "sun is", "is shining"

The `CountVectorizer` class in scikit-learn allows us to use different n-gram models via its `ngram_range` parameter. While a 1-gram representation is used by default, we could switch to a 2-gram representation by initializing a new `CountVectorizer` instance with `ngram_range=(2,2)`.

Assessing word relevancy via term frequency-inverse document frequency

When we are analyzing text data, we often encounter words that occur across multiple documents from both classes. These frequently occurring words typically don't contain useful or discriminatory information. In this subsection, we will learn about a useful technique called **term frequency-inverse document frequency** (**tf-idf**) that can be used to downweight these frequently occurring words in the feature vectors. The tf-idf can be defined as the product of the term frequency and the inverse document frequency:

$$\text{tf-idf}(t,d) = tf(t,d) \times \text{idf}(t,d)$$

Here the *tf(t, d)* is the term frequency that we introduced in the previous section, and *idf(t, d)* is the inverse document frequency and can be calculated as follows:

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

As we can see, the results now match the results returned by scikit-learn's `TfidfTransformer`, and since we now understand how tf-idfs are calculated, let's proceed to the next section and apply those concepts to the movie review dataset.

Cleaning text data

In the previous subsections, we learned about the bag-of-words model, term bag-of-words model, term frequencies, and tf-idfs. However, the first important step—before we build our bag-of-words model—is to clean the text data by stripping it of all unwanted characters. To illustrate why this is important, let's display the last 50 characters from the first document in the reshuffled movie review dataset:

```
>>> df.loc[0, 'review'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

As we can see here, the text contains HTML markup as well as punctuation and other non-letter characters. While HTML markup does not contain much useful semantics, punctuation marks can represent useful, additional information in certain NLP contexts. However, for simplicity, we will now remove all punctuation marks except for emoticon characters such as :) since those are certainly useful for sentiment analysis. To accomplish this task, we will use Python's **regular expression** (**regex**) library, `re`, as shown here:

```
>>> import re
>>> def preprocessor(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
...                            text)
...     text = (re.sub('[\W]+', ' ', text.lower()) +
...             ' '.join(emoticons).replace('-', ''))
...     return text
```

Via the first regex `<[^>]*>` in the preceding code section, we tried to remove all of the HTML markup from the movie reviews. Although many programmers generally advise against the use of regex to parse HTML, this regex should be sufficient to *clean* this particular dataset. After we removed the HTML markup, we used a slightly more complex regex to find emoticons, which we temporarily stored as `emoticons`. Next, we removed all non-word characters from the text via the regex `[\W]+` and converted the text into lowercase characters.

Processing documents into tokens

After successfully preparing the movie review dataset, we now need to think about how to split the text corpora into individual elements. One way to *tokenize* documents is to split them into individual words by splitting the cleaned documents at its whitespace characters:

```
>>> def tokenizer(text):
...     return text.split()
>>> tokenizer('runners like running and thus they run')
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

In the context of tokenization, another useful technique is **word stemming**, which is the process of transforming a word into its root form. It allows us to map related words to the same stem. The original stemming algorithm was developed by Martin F. Porter in 1979 and is hence known as the **Porter stemmer** algorithm (*An algorithm for suffix stripping, Martin F. Porter, Program: Electronic Library and Information Systems*, 14(3): 130–137, *1980*). The **Natural Language Toolkit** (**NLTK**, `http://www.nltk.org`) for Python implements the Porter stemming algorithm, which we will use in the following code section. In order to install the NLTK, you can simply execute `conda install nltk` or `pip install nltk`.

Although the NLTK is not the focus of the chapter, I highly recommend that you visit the NLTK website as well as read the official NLTK book, which is freely available at `http://www.nltk.org/book/`, if you are interested in more advanced applications in NLP.

The following code shows how to use the Porter stemming algorithm:

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter = PorterStemmer()
>>> def tokenizer_porter(text):
...     return [porter.stem(word) for word in text.split()]
>>> tokenizer_porter('runners like running and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

Using the `PorterStemmer` from the `nltk` package, we modified our `tokenizer` function to reduce words to their root form, which was illustrated by the simple preceding example where the word `'running'` was *stemmed* to its root form `'run'`.

Training a logistic regression model for document classification

In this section, we will train a logistic regression model to classify the movie reviews into *positive* and *negative* reviews. First, we will divide the `DataFrame` of cleaned text documents into 25,000 documents for training and 25,000 documents for testing:

```
>>> X_train = df.loc[:25000, 'review'].values
>>> y_train = df.loc[:25000, 'sentiment'].values
>>> X_test = df.loc[25000:, 'review'].values
>>> y_test = df.loc[25000:, 'sentiment'].values
```

Next, we will use a `GridSearchCV` object to find the optimal set of parameters for our logistic regression model using 5-fold stratified cross-validation:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.feature_extraction.text import TfidfVectorizer

>>> tfidf = TfidfVectorizer(strip_accents=None,
...                         lowercase=False,
...                         preprocessor=None)
>>> param_grid = [{'vect__ngram_range': [(1,1)],
...               'vect__stop_words': [stop, None],
...               'vect__tokenizer': [tokenizer,
...                                   tokenizer_porter],
...               'clf__penalty': ['l1', 'l2'],
...               'clf__C': [1.0, 10.0, 100.0]},
...              {'vect__ngram_range': [(1,1)],
...               'vect__stop_words': [stop, None],
...               'vect__tokenizer': [tokenizer,
...                                   tokenizer_porter],
...               'vect__use_idf':[False],
...               'vect__norm':[None],
...               'clf__penalty': ['l1', 'l2'],
...               'clf__C': [1.0, 10.0, 100.0]}
...              ]
>>> lr_tfidf = Pipeline([('vect', tfidf),
...                      ('clf',
...                       LogisticRegression(random_state=0))])
>>> gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
```

As we can see in the preceding output, we obtained the best grid search results using the regular `tokenizer` without Porter stemming, no stop-word library, and tf-idfs in combination with a logistic regression classifier that uses L2-regularization with the regularization strength C of `10.0`.

Using the best model from this grid search, let's print the average 5-fold cross-validation accuracy scores on the training set and the classification accuracy on the test dataset:

```
>>> print('CV Accuracy: %.3f'
...       % gs_lr_tfidf.best_score_)
CV Accuracy: 0.892
>>> clf = gs_lr_tfidf.best_estimator_
>>> print('Test Accuracy: %.3f'
...     % clf.score(X_test, y_test))
Test Accuracy: 0.899
```

The results reveal that our machine learning model can predict whether a movie review is positive or negative with 90 percent accuracy.

A still very popular classifier for text classification is the Naïve Bayes classifier, which gained popularity in applications of email spam filtering. Naïve Bayes classifiers are easy to implement, computationally efficient, and tend to perform particularly well on relatively small datasets compared to other algorithms. Although we don't discuss Naïve Bayes classifiers in this book, the interested reader can find my article about Naïve text classification that I made freely available on *arXiv* (*Naive Bayes and Text Classification I – Introduction and Theory, S. Raschka, Computing Research Repository (CoRR)*, abs/1410.5329, *2014*, `http://arxiv.org/pdf/1410.5329v3.pdf`).

Working with bigger data – online algorithms and out-of-core learning

If you executed the code examples in the previous section, you may have noticed that it could be computationally quite expensive to construct the feature vectors for the 50,000 movie review dataset during grid search. In many real-world applications, it is not uncommon to work with even larger datasets that can exceed our computer's memory. Since not everyone has access to supercomputer facilities, we will now apply a technique called **out-of-core learning**, which allows us to work with such large datasets by fitting the classifier incrementally on smaller batches of the dataset.

If you are planning to continue directly with *Chapter 9*, *Embedding a Machine Learning Model into a Web Application*, I recommend you keep the current Python session open. In the next chapter, we will use the model that we just trained to learn how to save it to disk for later use and embed it into a web application.

A more modern alternative to the bag-of-words model is **word2vec**, an algorithm that Google released in 2013 (*Efficient Estimation of Word Representations in Vector Space*, *T. Mikolov*, *K. Chen*, *G. Corrado*, and *J. Dean*, arXiv preprint arXiv:1301.3781, *2013*). The word2vec algorithm is an unsupervised learning algorithm based on neural networks that attempts to automatically learn the relationship between words. The idea behind word2vec is to put words that have similar meanings into similar clusters, and via clever vector-spacing, the model can reproduce certain words using simple vector math, for example, *king – man + woman = queen*.

The original C-implementation with useful links to the relevant papers and alternative implementations can be found at `https://code.google.com/p/word2vec/`.

Topic modeling with Latent Dirichlet Allocation

Topic modeling describes the broad task of assigning topics to unlabelled text documents. For example, a typical application would be the categorization of documents in a large text corpus of newspaper articles where we don't know on which specific page or category they appear in. In applications of topic modeling, we then aim to assign category labels to those articles—for example, sports, finance, world news, politics, local news, and so forth. Thus, in the context of the broad categories of machine learning that we discussed in *Chapter 1*, *Giving Computers the Ability to Learn from Data*, we can consider topic modeling as a clustering task, a subcategory of unsupervised learning.

In this section, we will introduce a popular technique for topic modeling called **Latent Dirichlet Allocation** (**LDA**). However, note that while Latent Dirichlet Allocation is often abbreviated as LDA, it is not to be confused with Linear discriminant analysis, a supervised dimensionality reduction technique that we introduced in *Chapter 5*, *Compressing Data via Dimensionality Reduction*.

LDA is different from the supervised learning approach that we took in this chapter to classify movie reviews as positive and negative. Thus, if you are interested in embedding scikit-learn models into a web application via the Flask framework using the movie reviewer as an example, please feel free to jump to the next chapter and revisit this standalone section on topic modeling later on.

Decomposing text documents with LDA

Since the mathematics behind LDA is quite involved and requires knowledge about Bayesian inference, we will approach this topic from a practitioner's perspective and interpret LDA using layman's terms. However, the interested reader can read more about LDA in the following research paper: *Latent Dirichlet Allocation*, *David M. Blei*, *Andrew Y. Ng*, and *Michael I. Jordan*, *Journal of Machine Learning Research 3*, pages: 993-1022, *Jan 2003*.

LDA is a generative probabilistic model that tries to find groups of words that appear frequently together across different documents. These frequently appearing words represent our topics, assuming that each document is a mixture of different words. The input to an LDA is the bag-of-words model we discussed earlier in this chapter. Given a bag-of-words matrix as input, LDA decomposes it into two new matrices:

- A document to topic matrix
- A word to topic matrix

LDA decomposes the bag-of-words matrix in such a way that if we multiply those two matrices together, we would be able to reproduce the input, the bag-of-words matrix, with the lowest possible error. In practice, we are interested in those topics that LDA found in the bag-of-words matrix. The only downside may be that we must define the number of topics beforehand—the number of topics is a hyperparameter of LDA that has to be specified manually.

LDA with scikit-learn

In this subsection, we will use the `LatentDirichletAllocation` class implemented in scikit-learn to decompose the movie review dataset and categorize it into different topics. In the following example, we restrict the analysis to 10 different topics, but readers are encouraged to experiment with the hyperparameters of the algorithm to explore the topics that can be found in this dataset further.

```
<br /><br />Horror movie time, Japanese style. Uzumaki/Spiral was a
total freakfest from start to finish. A fun freakfest at that, but at
times it was a tad too reliant on kitsch rather than the horror. The
story is difficult to summarize succinctly: a carefree, normal teenage
girl starts coming fac ...
```

Using the preceding code example, we printed the first 300 characters from the top three horror movies, and we can see that the reviews—even though we don't know which exact movie they belong to—sound like reviews of horror movies (however, one might argue that `Horror movie #2` could also be a good fit for topic category 1: *Generally bad movies*).

Summary

In this chapter, we learned how to use machine learning algorithms to classify text documents based on their polarity, which is a basic task in sentiment analysis in the field of NLP. Not only did we learn how to encode a document as a feature vector using the bag-of-words model, but we also learned how to weight the term frequency by relevance using tf-idf.

Working with text data can be computationally quite expensive due to the large feature vectors that are created during this process; in the last section, we learned how to utilize out-of-core or incremental learning to train a machine learning algorithm without loading the whole dataset into a computer's memory.

Lastly, we introduced the concept of topic modeling using LDA to categorize the movie reviews into different categories in unsupervised fashion.

In the next chapter, we will use our document classifier and learn how to embed it into a web application.