8_Applying_Machine_Learning_To_Sentiment_Analysis

June 29, 2018

1 Applying Machine Learning to Sentiment Analysis

In this internet and social media age, people's opinions, reviews, and recommendations have become a valuable resource of political science and business. 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 a dataset of 50.000 movie reviews from the **Internet Movie Database (IMDb)** and build a predict 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

1.1 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 particualr 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. The movie review datasets 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 IMBDb, 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.

1.2 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. You can unpack the Gzip-compressed tarball archive directly in Python as follows:

```
In [1]: import tarfile

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

1.3 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)** package. PyPrind can be installed by executing the *pip install pyprind* command.

```
In [2]: import pyprind
        import pandas as pd
        import os

        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 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: 00:01:20
```

In the preceding code, we first initialized a new progress bar object *pbar* with 50.000 iterations, which is the number of documents we were going to read in. Using the nested *for* loops, we iterated over the *train* and *test* subdirectories in the main *aclImdb* directory and read the individual text files from the *pos* and *neg* subdirectories that we eventually appended to the *df* pandas *DataFrame*, together with an integer class label (1 = positive and 0 = negative).

Since the class labels in the assembled dataset are sorted, we will now shuffle *DataFrame* using the *permutation* function from the *np.random* submodule - this will be useful to split the dataset into training and test sets in later sections when we will stream the data from our local drive directly. For our own convenience, we will also store the assembled and shuffled movie review dataset as a CSV file:

```
In [3]: import numpy as np

        np.random.seed(0)
        df = df.reindex(np.random.permutation(df.index))
        df.to_csv('movie_data.csv', index=False, encoding='utf-8')
```

Since we are going to use this dataset later in this chapter, let's quickly confirm that we have successfully saved the data in the right format by reading in the CSV and printing an excerpt of the first three samples:

```
In [4]: df = pd.read_csv('movie_data.csv', encoding='utf-8')
        df.head(3)
```

```
Out[4]:                                                 review  sentiment
        0  This movie is the absolutely perfect way to ex...          1
        1  I don't usually like TV movies, I reckon that ...          0
        2  First of all, I am not a huge fan of contempor...          0
```

1.4 Introducing the bag-of-words model

You may remember 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 mosly 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.

1.5 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:

```
In [5]: 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 and the weather is sweet'])
        bag = count.fit_transform(docs)
```

By calling the *fit_transform* method on *CountVectorizer*, we constructed the vocabulary of the bag-of-words model and transformed the following three sentences into sparse feature vectors:

- 'The sun is shining'
- 'The weather is sweet'
- 'The sun is shining and the weather is sweet'

Now let's print the contents of the vocabulary to get a better understanding of the underlying concepts:

```
In [6]: print(count.vocabulary_)
```

```
{'the': 5, 'sun': 3, 'is': 1, 'shining': 2, 'weather': 6, 'sweet': 4, 'and': 0}
```

As we can see from executing the preceding command, the vocabulary is stored in a Python dictionary that maps the unique words to integer indices. Next, let's print the feature vectors that we just created:

```
In [7]: print(bag.toarray())
```

```
[[0 1 1 1 0 1 0]
 [0 1 0 0 1 1 1]
 [1 2 1 1 1 2 1]]
```

Each index position in the feature vectors shown here corresponds to the integer values that are stored as dictionary items in the *CountVectorizer* vocabulary. For example, the first feature at index position 0 resembles the count of the word *'and'*, which only occurs in the last document, and the word *'is'*, at index position 1 (the second feature in the document vectors), occurs in all three sentences. These values in the feature vectors are also called the **raw term frequencies (tf(t, d))** - the number of times a term *t* occurs in a document *d*.

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 item 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 other revealedthat n-grams of size 3 or 4 yield good performance in anti-spam filtering of email messages. 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)*.

1.6 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 do not 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:

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

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:

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

Here, n_d is the total number of documents, and $df(d,t)$ is the number of documents d that contain the term t. Note that adding the constant 1 to the denominator is optional and serves the purpose of assigning a non-zero value to terms that occur in all training samples; the *log* is used to ensure that low document frequencies are not given too much weight.

The scikit-learn library implements yet another transformer, the *TfidfTransformer* class, that takes the raw term frequencies from the *CountVectorizer* class as input and transforms them into tf-idfs:

```
In [8]: from sklearn.feature_extraction.text import TfidfTransformer

        tfidf = TfidfTransformer(use_idf=True, norm='l2',
                                 smooth_idf=True)
        np.set_printoptions(precision=2)
        print(tfidf.fit_transform(count.fit_transform(docs)).toarray())
```

```
[[0.   0.43 0.56 0.56 0.   0.43 0.  ]
 [0.   0.43 0.   0.   0.56 0.43 0.56]
 [0.4  0.48 0.31 0.31 0.31 0.48 0.31]]
```

As we saw in the previous subsection, the word *'is'* had the largest term frequency in the third document, being the most frequently occurring word. However, after transforming the same feature vector into tf-idfs, we see that the work *'is'* s now associated with a relatively small tf-idf (0.48) in the third document, since it is also present in the first and second document and thus is unlikely to contain any useful discriminatory information.

However, if we would manually calculated the tf-idfs of the individual terms in our feature vector, we would notice that *TfidfTransformer* calculates the tf-idfs slightly different compared to the standard textbook equations that we defined previously.

While it is also more typical to normalize the raw terms frequencies before calculating the tf-idfs, *TfidfTranformer* class normalizes the tf-idfs directly. By default (norm = 'l2'), scikit-learn's *TfidfTransformer* applies the L2-normalization, which returns a vector of length 1 by dividing an un-normalized feature vector v by its L2-norm.

1.7 Cleaning text data

In the previous subsections, we learned about the 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:

```
In [9]: df.loc[0, 'review'][-50::]
```

```
Out[9]: 'your hearts forever :-).<br /><br />Regards Klavs.'
```

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 certai NLP context. 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:

```
In [10]: 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 advice 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 last regex and converted the text into lowercase characters.

Eventually, we added the temporarily stored *emoticons* to the end of the processed document string. Additionally, we remove the *nose* character (-) from the emoticons for consistency.

Although the addition of the emoticon characters to the end of the cleaned document string may not look like the most elegant approach, we shall note that the order of the words does not matter in our bag-of-words model if our vocabulary consists of only one-word tokens. But before we talk more about the splitting of documents into individual term, words, or tokens, let's confirm that our preprocessor works correctly:

```
In [11]: preprocessor(df.loc[0, 'review'][-50:])
```

```
Out[11]: 'your hearts forever regards klavs :)'
```

Lastly, since we will make use of the cleaned text data over and over again during the next sections, let us now apply our *preprocessor* function to all the movie reviews in our *DataFrame*:

```
In [12]: df['review'] = df['review'].apply(preprocessor)
```

1.8 Preprocessing 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:

```
In [13]: def tokenizer(text):
             return text.split()
         tokenizer('runners like running and thus they run')

Out[13]: ['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

In the context of tokenization, another useful tecnique is **word stemming**, which is the process of transforming a work into its root form. It allows us to map related words to the same steam. The original stemming algorithm was developed by Martin F. Porter in 1979 and is hence known as the **Porter stemmer** algorithm. The **Natural Language Toolkit (NLTK)** 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 *pip install nltk*.

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

```
In [14]: 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 to running and thus they run')

Out[14]: ['runner', 'like', 'to', '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'*.

The Porter stemming algorithm is probably the oldest and simplest stemming algorithm. Other popular stemming algorithms include the newer **Snowball stemmer** and the **Lancaster stemmer**, which is fater but also more aggressive then the Porter stemmer.

While stemming can create non-real words, such as *'thu'* (from *'thus'*), as shown in the previous example, a technique called **lemmatization** aims to obtain the canonical (grammatically correct) forms of individual words - the so-called **lemmas**. However, lemmatization is computationally more difficulty and expensive compared to stemming and, in practice, it has been observed that stemming and lemmatization have little impact on the performance of text classification.

Before we jump into the next section, where we will train a machine learning model using the bag-of-words model, let's briefly talk about another useful topic called **stop-word removal**. Stop-words are simply those words that are extremely common in all sorts of texts and probably bear no (or only little) useful information that can be used to distinguish between different classes of documents. Examples of stop-words are *is*, *and*, *has*, and *like*. Removing stop-words can be useful if we are working with raw or normalized term frequencies rather than tf-idfs, which are already downweighting frequently occurring words.

In order to remove stop-words from the movie reviews, we will use the set of 127 English stop-words that is available from the NLTK library, which can be obtained by calling the *nltk.download* function:

```
In [15]: import nltk

         nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /home/wesley/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
Out[15]: True
```

After we download the stop-words set, we can load and apply the English stop-word set as follows:

```
In [16]: from nltk.corpus import stopwords

         stop = stopwords.words('english')
         [w for w in tokenizer_porter('a runner likes running and runs a lot')[-10:]  if w not i
```

```
Out[16]: ['runner', 'like', 'run', 'run', 'lot']
```

1.9 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:

```
In [17]: 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:

```
In [18]: 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,
                           scoring='accuracy', cv=5, verbose=1,
                           n_jobs=-1)
gs_lr_tfidf.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 48 candidates, totalling 240 fits


[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed: 13.1min
[Parallel(n_jobs=-1)]: Done 184 tasks      | elapsed: 73.0min
[Parallel(n_jobs=-1)]: Done 240 out of 240 | elapsed: 98.9min finished
```

```
Out[18]: GridSearchCV(cv=5, error_score='raise',
               estimator=Pipeline(memory=None,
             steps=[('vect', TfidfVectorizer(analyzer='word', binary=False, decode_error='stric
                dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
                lowercase=False, max_df=1.0, max_features=None, min_df=1,
                ngram_range=(1, 1), norm='l2', preprocessor=None, smooth_idf=True,
          ...nalty='l2', random_state=0, solver='liblinear', tol=0.0001,
                  verbose=0, warm_start=False))]),
               fit_params=None, iid=True, n_jobs=-1,
               param_grid=[{'vect__ngram_range': [(1, 1)], 'vect__stop_words': [['i', 'me', 'my
               pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
               scoring='accuracy', verbose=1)
```

When we initialized the *GridSearchCV* object and its parameter grid using the preceding code, we restricted ourselves to a limited number of parameter combinations, since the number of feature vectors, as well as the large vocabulary, can make the grid search computationally quite expensive. Using a standard desktop computer, our grid search may take up to 40 minutes to complete.

In the previous code example, we replaced *CountVectorizer* and *TfidfTransformer* from the previous subsection with *TfidfVectorizer*, which combines the latter transformer objects. Our *param_grid* consisted of two parameter dictionaries. In the first dictionary, we used the *TfidfVectorizer* with its default settings (*use_idf=True*, *smooth_idf=True*, and *norm='l2'*) to calculate the tf-idfs; in the second dictionary, we set those parameters to *use_idf=False*, *smooth_idf=False*, and *norm=None* in order to train a model based on raw term frequencies. Furthermore, for the logistic regression classifier itself, we trained models using L2 and L1 regularization via the penalty parameter and compared different regularization strengths by defining a range of values for the inverse-regularization parameter C.

After the grid search has finished, we can print the best parameters set:

```
In [20]: print('Best parameter set: %s' %gs_lr_tfidf.best_params_)
```

```
Best parameter set: {'clf__C': 10.0, 'clf__penalty': 'l2', 'vect__ngram_range': (1, 1), 'vect__s
```

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 that uses L2-regularization with the regularization strength 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:

```
In [21]: print('CV Accuracy: %.3f' % (gs_lr_tfidf.best_score_))
         clf = gs_lr_tfidf.best_estimator_
         print('Test Accuracy: %.3f' % (clf.score(X_test, y_test)))
```

```
CV Accuracy: 0.897
Test Accuracy: 0.896
```

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 Naive Bayes classifier, which gained popularity in applications of email spam filtering. Naive Bayes classifiers are easy to implement, computationally efficient, and tend to perform particularly well on relatively small datasets compared to other algorithms.

2 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 common 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 techinique 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.

We introduced earlier the concept of **stochastic gradient descent**, which is an optimization algorithm that updates the model's weights using one sample at a time. In this section, we will make use of the *partial_fit* function of *SGDClassifier* in scikit-learn to stream the documents directly from our local drive, and train a logistic regression model using small mini-batches of documents.

First, we define a *tokenizer* function that cleans the unprocessed text data from the *movie_data.csv* file that we constructed at the beginning of this chapter and separate it into token words while removing stop words:

```
In [22]: import numpy as np
         import re
         from nltk.corpus import stopwords

         stop = stopwords.words('english')
```

```
def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text.lower())
    text = (re.sub('[\W]+', ' ', text.lower()) + ' '.join(emoticons).replace('-', ''))
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized
```

Next, we define a generator function *stream_docs* that reads in and return one document at a time:

```
In [23]: def stream_docs(path):
             with open(path, 'r', encoding='utf-8') as csv:
                 next(csv) # skip header
                 for line in csv:
                     text, label = line[:-3], int(line[-2])
                     yield text, label
```

To verify that our *stream_docs* functions works correctly, let's read in the first document from the *'movie_data.csv'* file, which should return a tuple consisting of the review text as well as the corresponding class label:

```
In [24]: next(stream_docs(path='movie_data.csv'))
```

```
Out[24]: ('"This movie is the absolutely perfect way to explain what a good movie is. It is a mo
          1)
```

We will now define a function, *get_minibatch*, that will take a document stream from the *stream_docs* function and return a particular number of documents specified by the *size* parameter:

```
In [25]: def get_minibatch(doc_stream, size):
             docs, y = [], []
             try:
                 for _ in range(size):
                     text, label = next(doc_stream)
                     docs.append(text)
                     y.append(label)
             except StopIteration:
                 return None, None
             return docs, y
```

Unfortunately, we cannot use *CountVectorizer* for out-of-core learning since it requires holding the complete vocabulary in memory. Also, *TfidfVectorizer* needs to keep all the feature vectors of the training dataset in memory to calculate the inverse documents frequencies. However, another useful vectorizer for text processing implemented in scikit-learn is *HashingVectorizer*. *HashingVectorizer* is data-independent and makes use of the hashing trick via the 32-bit *MurmurHash3* function.

```
In [26]: from sklearn.feature_extraction.text import HashingVectorizer
         from sklearn.linear_model import SGDClassifier

         vect = HashingVectorizer(decode_error='ignore',
                                  n_features=2**21,
                                  preprocessor=None,
                                  tokenizer=tokenizer)
         clf = SGDClassifier(loss='log', random_state=1, n_iter=1)
         doc_stream = stream_docs(path='movie_data.csv')
```

Using the preceding code, we initialized *HashingVectorizer* with our tokenizer function and set the number of features to *2^21*. Furthermore, we reinitialized a logistic regression classifier by setting the *loss* parameter of the *SGDClassifier* to *'log'* - note that by choosing a large number of features in the *HashingVectorizer*, we reduce the chance of causing hash collisions, but we also increase the number of coefficients in our logistic regression model. Now comes the really interesting part. Having set up all the complementary functions, we can now start the out-of-core learning using the following code:

```
In [28]: import pyprind

         pbar = pyprind.ProgBar(45)
         classes = np.array([0, 1])
         for _ in range(45):
             X_train, y_train = get_minibatch(doc_stream, size=1000)
             if not X_train:
                 break
             X_train = vect.transform(X_train)
             clf.partial_fit(X_train, y_train, classes=classes)
             pbar.update()
```

```
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [#                            ] 100% | ETA: 00:00:23/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [##                           ] 100% | ETA: 00:00:23/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [###                          ] 100% | ETA: 00:00:22/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [####                         ] 100% | ETA: 00:00:20/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [#####                        ] 100% | ETA: 00:00:20/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
```

```
0% [######                        ] 100% | ETA: 00:00:19/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [#######                       ] 100% | ETA: 00:00:18/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [########                      ] 100% | ETA: 00:00:18/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [#########                     ] 100% | ETA: 00:00:17/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [##########                    ] 100% | ETA: 00:00:15/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [###########                   ] 100% | ETA: 00:00:15/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [############                  ] 100% | ETA: 00:00:14/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [#############                 ] 100% | ETA: 00:00:13/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [##############                ] 100% | ETA: 00:00:13/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [###############               ] 100% | ETA: 00:00:11/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [################              ] 100% | ETA: 00:00:10/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [#################             ] 100% | ETA: 00:00:10/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [##################            ] 100% | ETA: 00:00:09/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [###################           ] 100% | ETA: 00:00:08/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [####################          ] 100% | ETA: 00:00:08/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [#####################         ] 100% | ETA: 00:00:07/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
```

```
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [#######################        ] 100% | ETA: 00:00:05/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [#######################        ] 100% | ETA: 00:00:05/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [#########################      ] 100% | ETA: 00:00:04/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [##########################     ] 100% | ETA: 00:00:03/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [###########################    ] 100% | ETA: 00:00:03/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [############################   ] 100% | ETA: 00:00:02/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
0% [#############################  ] 100% | ETA: 00:00:01/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [############################## ] 100% | ETA: 00:00:00/usr/lib/python3.6/site-packages/sklearn
  DeprecationWarning)
0% [##############################] 100% | ETA: 00:00:00
Total time elapsed: 00:00:24
```

Again, we made use of PyPrind package in order to estimate the progress of our learning algorithm. We initialized the progress bar object with 45 iterations and, in the following *for* loop, we iterated over 45 mini-batches of documents were each mini-batch consists of 1.000 documents. Having completed the incremental learning process, we will use the last 5.000 documents to evaluate the performance of our model:

```
In [29]: X_test, y_test = get_minibatch(doc_stream, size=5000)
         X_test = vect.transform(X_test)
         print('Accuracy: %.3f' % clf.score(X_test, y_test))
```

```
Accuracy: 0.872
```

As we can see, the accuracy of the model is approximately 88 percent, slight below the accuracy that we achieved in the previous section using the grid search for hyperparameter tuning. However, out-of-core learning is very memory efficient and took less than a minute to complete. Finally, we can use the last 5.000 documents to update our model:

```
In [30]: clf = clf.partial_fit(X_test, y_test)
```

```
/usr/lib/python3.6/site-packages/sklearn/linear_model/stochastic_gradient.py:117: DeprecationWar
  DeprecationWarning)
```

If you are planning to continue directly to the next chapter, 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 the disk for later use and embed it into a web application.

3 Topic modeling with Latent Dirichlet Allocation

Topic modeling describes the broad task of assigning topis 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 do not 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, 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 if often abbreviated as LDA, it is not to be confused with Linear Discriminant Analysis, a supervised dimensionality reduction technique.

3.1 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.

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 a LDA is the bag-of-words model we discusse earlier in this chapter. Given a bag-of-words matrix as input, LDA decomposes it into two new matrices:

- A document of 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.

3.2 LDA with scikit-learn

In this subsection, we will use the *LatentDirichAllocation* 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.

First, we are going to load the dataset into a pandas *DataFrame* using the local *movie_data.csv* file of the movie reviews that we have created at the beginning of this chapter:

```
In [31]: import pandas as pd

         df = pd.read_csv('movie_data.csv', encoding='utf-8')
```

Next, we are going to use the already familiar *CountVectorizer* to create the bag-of-words matrix as input to the LDA. For convenience, we will use scikit-learn's built-in English stop word library via *stop_words='english'*:

```
In [32]: from sklearn.feature_extraction.text import CountVectorizer

         count = CountVectorizer(stop_words='english',
                                 max_df=.1,
                                 max_features=5000)
         X = count.fit_transform(df['review'].values)
```

Notice that we set the maximum document frequency of words to be considered to 10 percent (*max_df=.1*) to exclude words that occur too frequently across documents. The rationale behind the removal of frequently occuring words is that these might be common words appearing across all documents and are therefore less likely associated with a specific topic category of a given document. Also, we limited the number of words to be considered to the most frequently occuring 5.000 words (*max_features=5000*), to limit the dimensionality of this dataset so that it improves the inference performed by LDA. However, both *max_df-.1* and *max_features=5000* are hyperparameter values that I chose arbitrarily, and readers are encouraged to tune them while comparing the results.

The following code example demonstrates how to fit a *LatentDirichletAllocation* estimator to the bag-of-words matrix and infer the 10 different topics from the documents (note that the model fitting can take up to five minutes or more on a laptop or standard desktop computer):

```
In [33]: from sklearn.decomposition import LatentDirichletAllocation

         lda = LatentDirichletAllocation(n_topics=10,
                                         random_state=123,
                                         learning_method='batch')
         X_topics = lda.fit_transform(X)
```

```
/usr/lib/python3.6/site-packages/sklearn/decomposition/online_lda.py:294: DeprecationWarning: n_
  DeprecationWarning)
```

By setting *learning_method='batch'*, we let the *lda* estimator do its estimation based on all available training data (the bag-of-words matrix) in one iteration, which is slower than the alternative *'online'* learning method but can lead to more accurate results (setting *learning_method='online'* is analogous to online and mini-batch learning that we discussed previously).

After fitting the LDA, we now have access to the *components_* attribute of the *lda* instance, which stores a matrix containing the word importance (here, 5000) for each of the 10 topics in increasing order:

```
In [34]: lda.components_.shape

Out[34]: (10, 5000)
```

To analyse the results, let's print the five most important words for each of the 10 topics. Note that the word importance are ranked in increasing order. Thus, to print the five top words, we need to sort the *topic* array in reverse order:

```
In [35]: n_top_words = 5
         feature_names = count.get_feature_names()
         for topic_idx, topic in enumerate(lda.components_):
             print('Topic %d:' % (topic_idx + 1))
             print(' '.join([feature_names[i] for i in topic.argsort()[:-n_top_words-1:-1]]))
```

```
Topic 1:
worst minutes script awful stupid
Topic 2:
family mother father children girl
Topic 3:
american war dvd music tv
Topic 4:
human audience cinema art sense
Topic 5:
police dead guy car murder
Topic 6:
horror house sex girl woman
Topic 7:
role performance comedy actor performances
Topic 8:
series episode war episodes season
Topic 9:
book version original effects read
Topic 10:
action fight guy guys cool
```

Based on reading the five most important words for each topic, we may guess that the LDA identified the following topics:

1. Generally bad movies (not really a topic category)
2. Movies about families
3. War movies
4. Art movies
5. Crime movies
6. Horror movies
7. Comedy movies
8. Movies somehow related to TV shows
9. Movies based on books
10. Action movies

To confirm that the categories make sense based on the reviews, let's plot three movies from the horror movie category (horror movies belong to category 6 at index position 5):

```
In [37]: horror = X_topics[:, 5].argsort()[::-1]
         for iter_idx, movie_idx in enumerate(horror[:3]):
             print('\nHorror movie #%d:' % (iter_idx + 1))
             print(df['review'][movie_idx][:300], '...')
```

```
Horror movie #1:
Once upon a time in a castle... Two little girls are playing in the garden's castle. They are si

Horror movie #2:
House of Dracula works from the same basic premise as House of Frankenstein from the year before

Horror movie #3:
<br /><br />Horror movie time, Japanese style. Uzumaki/Spiral was a total freakfest from start t
```

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 do not 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*).

4 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 algorith 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.

5 Serializing fitted scikit-learn estimators

```
In [40]: import pickle
         import os

         dest = os.path.join('movieclassifier', 'pkl_objects')
         if not os.path.exists(dest):
             os.makedirs(dest)

         pickle.dump(stop, open(os.path.join(dest, 'stopwords.pkl'), 'wb'), protocol=4)
         pickle.dump(clf, open(os.path.join(dest, 'classifier.pkl'), 'wb'), protocol=4)
```