Statistical Natural Language Processing for Sentiment Analysis

10

10.1 Introduction

In this chapter, we will perform sentiment analysis from text data. The term sentiment analysis (or opinion mining) refers to the analysis from data of the attitude of the subject with respect to a particular topic. This attitude can be a judgment (appraisal theory), an affective state, or the intended emotional communication.

Generally, sentiment analysis is performed based on the processing of natural language, the analysis of text and computational linguistics. Although data can come from different data sources, in this chapter we will analyze sentiment in text data, using two particular text data examples: one from film critics, where the text is highly structured and maintains text semantics; and another example coming from social networks (tweets in this case), where the text can show a lack of structure and users may use (and abuse!) text abbreviations.

In the following sections, we will review some basic mechanisms required to perform sentiment analysis. In particular, we will analyze the steps required for data cleaning (that is, removing irrelevant text items not associated with sentiment information), producing a general representation of the text, and performing some statistical inference on the text represented to determine positive and negative sentiments.

Although the scope of sentiment analysis may introduce many aspects to be analyzed, in this chapter and for simplicity, we will analyze binary sentiment analysis categorization problems. We will thus basically learn to classify positive against negative opinions from text data. The scope of sentiment analysis is broader, and it includes many aspects that make analysis of sentiments a challenging task. Some interesting open issues in this topic are as follows:

- Identification of sarcasm: sometimes without knowing the personality of the person, you do not know whether "bad" means bad or good.

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_10

- Lack of text structure: in the case of Twitter, for example, it may contain abbreviations, and there may be a lack of capitals, poor spelling, poor punctuation, and poor grammar, all of which make it difficult to analyze the text.
- Many possible sentiment categories and degrees: positive and negative is a simple analysis, one would like to identify the amount of hate there is inside the opinion, how much happiness, how much sadness, etc.
- Identification of the object of analysis: many concepts can appear in text, and how to detect the object that the opinion is positive for and the object that the opinion is negative for is an open issue. For example, if you say "She won him!", this means a positive sentiment for her and a negative sentiment for him, at the same time.
- Subjective text: another open challenge is how to analyze very subjective sentences or paragraphs. Sometimes, even for humans it is very hard to agree on the sentiment of these highly subjective texts.

10.2 Data Cleaning

In order to perform sentiment analysis, first we need to deal with some processing steps on the data. Next, we will apply the different steps on simple "toy" sentences to understand better each one. Later, we will perform the whole process on larger datasets.

Given the input text data in cell `[1]`, the main task of data cleaning is to remove those characters considered as noise in the data mining process. For instance, comma or colon characters. Of course, in each particular data mining problem different characters can be considered as noise, depending on the final objective of the analysis. In our case, we are going to consider that all punctuation characters should be removed, including other non-conventional symbols. In order to perform the data cleaning process and posterior text representation and analysis we will use the *Natural Language Toolkit* (NLTK) library for the examples in this chapter.

In [1]:
```
raw_docs = ["Here are some very simple basic
    sentences.",
"They won't be very interesting, I'm afraid.",
"The point of these examples is to _learn  how
    basic text \
cleaning works_ on *very simple* data."]
```

The first step consists of defining a list with all word-vectors in the text. NLTK makes it easy to convert documents-as-strings into word-vectors, a process called tokenizing. See the example below.

In [2]:
```
from nltk.tokenize import word_tokenize
tokenized_docs = [word_tokenize(doc) for doc in
    raw_docs]
print tokenized_docs
```

Out[2]:
```
[['Here', 'are', 'some', 'very', 'simple', 'basic',
'sentences', '.'], ['They', 'wo', "n't", 'be', 'very',
'interesting', ',', 'I', "'m", 'afraid', %'.'], ['The',
'point', 'of', 'these', 'examples', 'is', 'to', '_learn',
'how', %'basic', 'text', 'cleaning', 'works_', 'on', '*very',
'simple*', 'data', '.']]
```

Thus, for each line of text in `raw_docs`, `word_tokenize` function will set the list of word-vectors. Now we can search the list for punctuation symbols, for instance, and remove them. There are many ways to perform this step. Let us see one possible solution using the `String` library.

In [3]:
```
import string
string.punctuation
```

Out[3]:
```
'!"#\$\%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

See that `string.punctuation` contains a set of common punctuation symbols. This list can be modified according to the symbols you want to remove. Let us see with the next example using the *Regular Expressions* (RE) package how punctuation symbols can be removed. Note that many other possibilities to remove symbols exist, such as directly implementing a loop comparing position by position.

In the input cell `[6]`, and without going into the details of RE, `re.compile` contains a list of "expressions", the symbols contained in `string.punctuation`.

Then, for each item in `tokenized_docs` that matches an expression/symbol contained in `regex`, the part of the item corresponding to the punctuation will be substituted by `u"` (where `u` refers to unicode encoding). If the item after substitution corresponds to `u"`, it will be not included in the final list. If the new item is different from `u"`, it means that the item contained text other than punctuation, and thus it is included in the new list without punctuation `tokenized_docs_no_punctuation`. The results of applying this script are shown in the output cell `[7]`.

In [4]:
```
import re
import string
regex = re.compile('[%s]' % re.escape(string.
    punctuation))
tokenized_docs_no_punctuation = []
for review in tokenized_docs:
    new_review = []
    for token in review:
        new_token = regex.sub(u'', token)
        if not new_token == u'':
            new_review.append(new_token)
    tokenized_docs_no_punctuation.append(new_review
        )
print tokenized_docs_no_punctuation
```

Out[4]:
```
[['Here', 'are', 'some', 'very', 'simple', 'basic',
'sentences'],
['They', 'wo', u'nt', 'be', 'very', 'interesting', 'I', u'm',
'afraid'],
['The', 'point', 'of', 'these', 'examples', 'is', 'to',
u'learn', 'how', 'basic', 'text', 'cleaning', u'works', 'on',
u'very', u'simple', 'data']]
```

One can see that punctuation symbols are removed, and those words containing a punctuation symbol are kept and marked with an initial `u`. If the reader wants more details, we recommend to read information about the RE package[1] for treating expressions.

Another important step in many data mining systems for text analysis consists of stemming and lemmatizing. Morphology is the notion that words have a root form. If you want to get to the basic term meaning of the word, you can try applying a stemmer or lemmatizer. This step is useful to reduce the dictionary size and the posterior high-dimensional and sparse feature spaces. NLTK provides different ways of performing this procedure. In the case of running the `porter.stem(word)` approach, the output is shown next.

In [5]:
```
from nltk.stem.porter import PorterStemmer
from nltk.stem.snowball import SnowballStemmer
from nltk.stem.wordnet import WordNetLemmatizer
porter = PorterStemmer()
#snowball = SnowballStemmer('english')
#wordnet = WordNetLemmatizer()

#each of the following commands perform stemming on
    word

porter.stem(word)
#snowball.stem(word)
#wordnet.lemmatize(word)
```

Out[5]:
```
[['Here', 'are', 'some', 'very', 'simple', 'basic',
'sentences'], ['They', 'wo', u'nt', 'be', 'very',
'interesting', 'I', u'm', 'afraid'], ['The', 'point', 'of',
'these', 'examples', 'is', 'to', u'learn', 'how', 'basic',
'text', 'cleaning', u'works', 'on', u'very', u'simple',
'data']]
[['Here', 'are', 'some', 'veri', 'simpl', 'basic', 'sentenc'],
['They', 'wo', u'nt', 'be', 'veri', 'interest', 'I', u'm',
'afraid'], ['The', 'point','of', 'these', 'exampl', 'is',
'to', u'learn', 'how', 'basic', 'text', 'clean', u'work', 'on',
u'veri',u'simpl', 'data']]
```

[1] https://docs.python.org/2/library/re.html.

This kind of approaches are very useful in order to reduce the exponential number of combinations of words with the same meaning and match similar texts. Words such as “interest” and “interesting” will be converted into the same word “interest” making the comparison of texts easier, as we will see later.

Another very useful data cleaning procedure consists of removing HTML entities and tags. Those may contain words and other symbols that were not removed by applying the previous procedures, but that do not provide useful meaning for text analysis and will introduce noise in our posterior text representation procedure. There are many possibilities for removing these tags. Here we show another example using the same NLTK package.

In [6]:
```
import nltk
test_string ="<p>While many of the stories tugged
    at the heartstrings, I never felt manipulated by
     the authors. (Note: Part of the reason why I
    don't like the 'Chicken Soup for the Soul'
    series is that I feel that the authors are just
    dying to make the reader clutch for the box of
    tissues.)</a>"
print 'Original text:'
print test_string
print 'Cleaned text:'
nltk.clean_html(test_string.decode())
```

Out[6]:
```
Original text:
<p>While many of the stories tugged at the heartstrings, I
never felt manipulated by the authors. (Note: Part of the
reason why I don't like the "Chicken Soup for the Soul" series
is that I feel that the authors are just dying to make the
reader clutch for the box of tissues.)</a>

Cleaned text:
u"While many of the stories tugged at the heartstrings, I never
felt manipulated by the authors. (Note: Part of the reason why
I don't like the "Chicken Soup for the Soul" series is that I
feel that the authors are just dying to make the reader clutch
for the box of tissues.)"
```

You can see that tags such as “<p>” and “” have been removed. The reader is referred to the RE package documentation to learn more about how to use it for data cleaning and HTLM parsing to remove tags.

10.3 Text Representation

In the previous section we have analyzed different techniques for data cleaning, stemming, and lemmatizing, and filtering the text to remove other unnecessary tags for posterior text analysis. In order to analyze sentiment from text, the next step consists of having a representation of the text that has been cleaned. Although different rep-

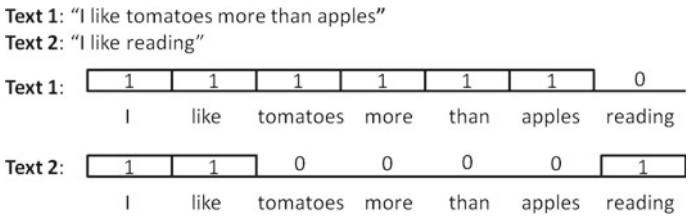

Fig. 10.1 Example of BoW representation for two texts

resentations of text exist, the most common ones are variants of *Bag of Words* (BoW) models [1]. The basic idea is to think about word frequencies. If we can define a dictionary of possible different words, the number of different existing words will define the length of a feature space to represent each text. See the toy example in Fig. 10.1. Two different texts represent all the available texts we have in this case. The total number of different words in this dictionary is seven, which will represent the length of the feature vector. Then we can represent each of the two available texts in the form of this feature vector by indicating the number of word frequencies, as shown in the bottom of the figure. The last two rows will represent the feature vector codifying each text in our dictionary.

Next, we will see a particular case of bag of words, the *Vector Space Model* of text: TF–IDF (term frequency–inverse distance frequency). First, we need to count the terms per document, which is the term frequency vector. See a code example below.

In [7]:
```
mydoclist = ['Mireia loves me more than Hector
    loves me',
     'Sergio likes me more than Mireia loves me',
     'He likes basketball more than football']
from collections import Counter
for doc in mydoclist:
     tf = Counter()
     for word in doc.split():
          tf[word] += 1
     print tf.items()
```

Out[7]:
```
[('me', 2), ('Mireia', 1), ('loves', 2), ('Hector', 1),
('than', 1), ('more', 1)] [('me', 2), ('Mireia', 1), ('likes',
1), ('loves', 1), ('Sergio', 1), ('than', 1), ('more', 1)]
[('basketball', 1), ('football', 1), ('likes', 1), ('He', 1),
('than', 1), ('more', 1)]
```

Here, we have introduced the Python object called a `Counter`. Counters are only in Python 2.7 and higher. They are useful because they allow you to perform this exact kind of function: counting in a loop. A Counter is a dictionary subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts.

Elements are counted from an iterable or initialized from another mapping (or Counter).

In [8]:
```
c = Counter()            # a new, empty counter
c = Counter('gallahad') # a new counter from an
    iterable
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a KeyError.

In [9]:
```
c = Counter(['eggs', 'ham'])
c['bacon']
```

Out[9]: 0

Let us call this a first stab at representing documents quantitatively, just by their word counts (also thinking that we may have previously filtered and cleaned the text using previous approaches). Here we show an example for computing the feature vector based on word frequencies.

In [10]:
```
def build_lexicon(corpus):
# define a set with all possible words included in
    all the sentences or "corpus"
    lexicon = set()
    for doc in corpus:
        lexicon.update([word for word in doc.split
            ()])
    return lexicon
def tf(term, document):
    return freq(term, document)
def freq(term, document):
    return document.split().count(term)
vocabulary = build_lexicon(mydoclist)
doc_term_matrix = []
print 'Our vocabulary vector is [' +
      ', '.join(list(vocabulary)) + ']'
for doc in mydoclist:
    print 'The doc is "' + doc + '"'
    tf_vector = [tf(word, doc) for word in
        vocabulary]
    tf_vector_string = ', '.join(format(freq, 'd')
                                 for freq
                                 in tf_vector)
    print 'The tf vector for Document %d is [%s]'
          % ((mydoclist.index(doc)+1),
              tf_vector_string)
    doc_term_matrix.append(tf_vector)
print 'All combined, here is our master document
      term matrix: '
print doc_term_matrix
```

Out[10]:
```
Our vocabulary vector is [me, basketball, Julie, baseball,
likes, loves, Jane, Linda, He, than, more]
The doc is "Julie loves me more than Linda loves me"
The tf vector for Document 1 is [2, 0, 1, 0, 0, 2, 0, 1, 0, 1,
1]
The doc is "Jane likes me more than Julie loves me"
The tf vector for Document 2 is [2, 0, 1, 0, 1, 1, 1, 0, 0, 1,
1]
The doc is "He likes basketball more than baseball"
The tf vector for Document 3 is [0, 1, 0, 1, 1, 0, 0, 0, 1, 1,
1]
All combined, here is our master document term matrix:
[[2, 0, 1, 0, 0, 2, 0, 1, 0, 1, 1], [2, 0, 1, 0, 1, 1, 1, 0, 0,
1, 1], [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1]]
```

Now, every document is in the same feature space, meaning that we can represent the entire corpus in the same dimensional space. Once we have the data in the same feature space, we can start applying some machine learning methods: learning, classifying, clustering, and so on. But actually, we have a few problems. Words are not all equally informative. If words appear too frequently in a single document, they are going to muck up our analysis. We want to perform some weighting of these term frequency vectors into something a bit more representative. That is, we need to do some vector normalizing. One possibility is to ensure that the L2 norm of each vector is equal to 1.

In [11]:
```
import math

def l2_normalizer(vec):
    denom = np.sum([el**2 for el in vec])
    return [(el / math.sqrt(denom)) for el in vec]
doc_term_matrix_l2 = []
for vec in doc_term_matrix:
    doc_term_matrix_l2.append(l2_normalizer(vec))
print 'A regular old document term matrix: '
print np.matrix(doc_term_matrix)
print '\nA document term matrix with row-wise L2
   norm:'
print np.matrix(doc_term_matrix_l2)
```

Out[11]:
```
A regular old document term matrix:
[[2 0 1 0 0 2 0 1 0 1 1]
[2 0 1 0 1 1 1 0 0 1 1]
[0 1 0 1 1 0 0 0 1 1 1]]
A document term matrix with row-wise L2 norm:
[[ 0.57735027 0. 0.28867513 0. 0. 0.57735027
0. 0.28867513 0. 0.28867513 0.28867513]
[ 0.63245553 0. 0.31622777 0. 0.31622777 0.31622777
0.31622777 0. 0. 0.31622777 0.31622777]
[ 0. 0.40824829 0. 0.40824829 0.40824829 0. 0.
0. 0.40824829 0.40824829 0.40824829]]
```

You can see that we have scaled down the vectors so that each element is between [0, 1]. This will avoid getting a diminishing return on the informative value of a word massively used in a particular document. For that, we need to scale down words that appear too frequently in a document.

Finally, we have a final task to perform. Just as not all words are equally valuable within a document, not all words are valuable across all documents. We can try reweighting every word by its inverse document frequency.

In [12]:
```
def numDocsContaining(word, doclist):
    doccount = 0
    for doc in doclist:
        if freq(word, doc) > 0:
            doccount += 1
    return doccount
def idf(word, doclist):
    n_samples = len(doclist)
    df = numDocsContaining(word, doclist)
    return np.log(n_samples / (float(df)) )
my_idf_vector = [idf(word, mydoclist) for word in
    vocabulary]
print 'Our vocabulary vector is [' + ', '.join(list
    (vocabulary)) + ']'
print 'The inverse document frequency vector is
    [' + ', '.join(format(freq, 'f') for freq in
    my_idf_vector) + ']'
```

Out[12]:
```
Our vocabulary vector is [me, basketball, Mireia, football,
likes, loves, Sergio, Hector, He, than, more]
The inverse document frequency vector is [0.405465, 1.098612,
0.405465, 1.098612, 0.405465, 0.405465, 1.098612, 1.098612,
1.098612, 0.000000, 0.000000]
```

Now we have a general sense of information values per term in our vocabulary, accounting for their relative frequency across the entire corpus. Note that this is an inverse. To get TF–IDF weighted word-vectors, we have to perform the simple calculation of the term frequencies multiplied by the inverse frequency values.

In the next example we convert our IDF vector into a matrix where the diagonal is the IDF vector.

In [13]:
```
def build_idf_matrix(idf_vector):
    idf_mat = np.zeros((len(idf_vector), len(
        idf_vector)))
    np.fill_diagonal(idf_mat, idf_vector)
    return idf_mat
my_idf_matrix = build_idf_matrix(my_idf_vector)
print my_idf_matrix
```

Out[13]:
```
[[ 0.40546511 0. 0. 0. 0. 0. 0. 0. 0. 0. ]
[ 0. 1.09861229 0. 0. 0. 0. 0. 0. 0. 0. ]
[ 0. 0. 0.40546511 0. 0. 0. 0. 0. 0. 0. ]
[ 0. 0. 0. 1.09861229 0. 0. 0. 0. 0. 0. ]
[ 0. 0. 0. 0. 0.40546511 0. 0. 0. 0. 0. ]
[ 0. 0. 0. 0. 0. 0.40546511 0. 0. 0. 0. ]
[ 0. 0. 0. 0. 0. 0. 1.09861229 0. 0. 0. ]
[ 0. 0. 0. 0. 0. 0. 0. 1.09861229 0. 0. ]
[ 0. 0. 0. 0. 0. 0. 0. 0. 1.09861229 0. ]
[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ]
[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ]]
```

That means we can now multiply every term frequency vector by the inverse document frequency matrix. Then, to make sure we are also accounting for words that appear too frequently within documents, we will normalize each document using the L2 norm.

In [14]:
```
doc_term_matrix_tfidf = []
#performing tf-idf matrix multiplication
for tf_vector in doc_term_matrix:
    doc_term_matrix_tfidf.append(np.dot(tf_vector,
        my_idf_matrix))
#normalizing
doc_term_matrix_tfidf_l2 = []
for tf_vector in doc_term_matrix_tfidf:
    doc_term_matrix_tfidf_l2.
    append(l2_normalizer(tf_vector))
print vocabulary
# np.matrix() just to make it easier to look at
print np.matrix(doc_term_matrix_tfidf_l2)
```

Out[14]:
```
set(['me', 'basketball', 'Mireia', 'football', 'likes',
'loves', 'Sergio', 'Linda', 'He', 'than', 'more'])
[[ 0.49474872 0. 0.24737436 0. 0. 0.49474872 0. 0.67026363 0.
0. 0. ]
[ 0.52812101 0. 0.2640605 0. 0.2640605 0.2640605 0.71547492 0.
0. 0. 0. ]
[ 0. 0.56467328 0. 0.56467328 0.20840411 0. 0. 0. 0.56467328 0.
0. ]]
```

10.3.1 Bi-Grams and n-Grams

It is sometimes useful to take significant bi-grams into the model based on the BoW. Note that this example can be extended to n-grams. In the fields of computational linguistics and probability, an n-gram is a contiguous sequence of n items from a given sequence of text or speech. The items can be phonemes, syllables, letters, words, or base pairs according to the application. The n-grams are typically collected from a text or speech corpus.

A n-gram of size 1 is referred to as a "uni-gram"; size 2 is a "bi-gram" (or, less commonly, a "digram"); size 3 is a "tri-gram". Larger sizes are sometimes referred to by the value of n, e.g., "four-gram", "five-gram", and so on. These n-grams can be introduced within the BoW model just by considering each different n-gram as a new position within the feature vector representation.

10.4 Practical Cases

Python packages provide useful tools for analyzing text. The reader is referred to the NLTK and *Textblob* package[2] documentation for further details. Here, we will perform all the previously presented procedures for data cleaning, stemming, and representation and introduce some binary learning schemes to learn the text representations in the feature space. The binary learning schemes will receive examples for training positive and negative sentiment texts and we will apply them later to unseen examples from a test set.

We will apply the whole sentiment analysis process in two examples. The first corresponds to the Large Movie reviews dataset [2]. This is one of the largest public available data sets for sentiment analysis, which includes more than 50,000 texts from movie reviews including the groundtruth annotation related to positive and negative movie reviews. As a proof on concept, for this example we use a subset of the dataset consisting of about 30% of the data.

The code reuses part of the previous examples for data cleaning, reads training and testing data from the folders as provided by the authors of the dataset. Then, TF–IDF is computed, which performs all steps mentioned previously for computing feature space, normalization, and feature weights. Note that at the end of the script we perform training and testing based on two different state-of-the-art machine learning approaches: *Naive Bayes* and *Support Vector Machines*. It is beyond the scope of this chapter to give details of the methods and parameters. The important point here is that the documents are represented in feature spaces that can be used by different data mining tools.

[2] https://textblob.readthedocs.io/en/dev/.

In [15]:

```
from nltk.tokenize import word_tokenize
from nltk.stem.porter import PorterStemmer
from sklearn.feature_extraction.text import
    TfidfVectorizer
from nltk.classify import NaiveBayesClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn import svm
from unidecode import unidecode

def BoW(text):
    # Tokenizing text
    text_tokenized = [word_tokenize(doc) for doc in
        text]
    # Removing punctuation
    regex = re.compile('[%s]' % re.escape(string.
        punctuation))
    tokenized_docs_no_punctuation = []
    for review in text_tokenized:
        new_review = []
        for token in review:
            new_token = regex.sub(u'', token)
            if not new_token == u'':
                new_review.append(new_token)
        tokenized_docs_no_punctuation.append(
            new_review)
    # Stemming and Lemmatizing
    porter = PorterStemmer()
    preprocessed_docs = []
    for doc in tokenized_docs_no_punctuation:
        final_doc = ''
        for word in doc:
            final_doc = final_doc + ' ' + porter.
                stem(word)
        preprocessed_docs.append(final_doc)
    return preprocessed_docs

#read your train text data here
textTrain=ReadTrainDataText()
preprocessed_docs=BoW(textTrain) # for train data
# Computing TIDF word space
tfidf_vectorizer = TfidfVectorizer(min_df = 1)
trainData = tfidf_vectorizer.fit_transform(
    preprocessed_docs)

textTest=ReadTestDataText() #read your test text
    data here
prepro_docs_test=BoW(textTest) # for test data
testData = tfidf_vectorizer.transform(
    prepro_docs_test)
```

In [16]:

```
print('Training and testing on training Naive Bayes
    ')
gnb = GaussianNB()
testData.todense()
y_pred = gnb.fit(trainData.todense(), targetTrain)
            .predict(trainData.todense())
print("Number of mislabeled training points out of
    a total %d points : %d"
       % (trainData.shape[0],(targetTrain != y_pred)
           .sum()))

y_pred = gnb.fit(trainData.todense(), targetTrain)
            .predict(testData.todense())
print("Number of mislabeled test points out of a
    total %d points : %d" %
       (testData.shape[0],(targetTest != y_pred).sum
           ()))

print('Training and testing on train with SVM')
clf = svm.SVC()
clf.fit(trainData.todense(), targetTrain)
y_pred = clf.predict(trainData.todense())
print("Number of mislabeled test points out of a
    total %d points : %d" %
       (trainData.shape[0],(targetTrain != y_pred).
           sum()))

print('Testing on test with already trained SVM')
y_pred = clf.predict(testData.todense())
print("Number of mislabeled test points out of a
    total %d points : %d" %
       (testData.shape[0],(targetTest != y_pred).sum
           ()))
```

In addition to the machine learning implementations provided by the Scikit-learn module used in this example, NLTK also provides useful learning tools for text learning, which also includes Naive Bayes classifiers. Another related package with similar functionalities is Textblob. The results of running the script are shown next.

Out[16]:
```
Training and testing on training Naive Bayes
Number of mislabeled training points out of a total 4313 points
: 129
Number of mislabeled test points out of a total 6292 points :
2087
Training and testing on train with SVM
Number of mislabeled test points out of a total 4313 points :
1288
Testing on test with already trained SVM
Number of mislabeled test points out of a total 6292 points :
1680
```

We can see that the training error of Naive Bayes on the selected data is 129/4313 while in testing it is 2087/6292. Interestingly, the training error using SVM is higher (1288/4313), but it provides a better generalization of the test set than Naive Bayes (1680/6292). Thus it seems that Naive Bayes produces more overfitting of the data (selecting particular features for better learning the training data but producing such high modifications of the feature space for testing that cannot be recovered, just reducing the generalization capability of the technique). However, note that this is a simple execution with standard methods on a subset of the dataset provided. More data, as well as many other aspects, will influence the performance. For instance, we could enrich our dictionary by introducing a list of already studied positive and negative words.[3] For further details of the analysis of this dataset, the reader is referred to [2].

Finally, let us see another example of sentiment analysis based on tweets. Although there is some work using more tweet data[4] here we present a reduced set of tweets which are analyzed as in the previous example of movie reviews. The main code remains the same except for the definition of the initial data.

In [17]:
```
textTrain = ['I love this sandwich.', 'This is an
    amazing place!', 'I feel very good about these
    beers.', 'This is my best work.', 'What an
    awesome view', 'I do not like this restaurant',
    'I am tired of this stuff.', 'I can not deal
    with this', 'He is my sworn enemy!', 'My boss is
     horrible.']
targetTrain = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
preprocessed_docs=BoW(textTrain)
tfidf_vectorizer = TfidfVectorizer(min_df = 1)
trainData = tfidf_vectorizer.fit_transform(
    preprocessed_docs)
```

[3] Such as those provided in http://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html.
[4] http://www.sananalytics.com/lab/twitter-sentiment/.

In [18]:
```
textTest = ['The beer was good.', 'I do not enjoy
    my job', 'I aint feeling dandy today', 'I feel
    amazing!', 'Gary is a friend of mine.', 'I can
    not believe I am doing this.']
targetTest = [0, 1, 1, 0, 0, 1]
preprocessed_docs=BoW(textTest)
testData = tfidf_vectorizer.transform(
    preprocessed_docs)

print('Training and testing on test Naive Bayes')
gnb = GaussianNB()
testData.todense()
y_pred = gnb.fit(trainData.todense(), targetTrain)
           .predict(trainData.todense())
print("Number of mislabeled training points out of
    a total %d points : %d" % (trainData.shape[0],(
    targetTrain != y_pred).sum()))

y_pred = gnb.fit(trainData.todense(), targetTrain)
           .predict(testData.todense())
print("Number of mislabeled test points out of a
    total %d points : %d" % (testData.shape[0],(
    targetTest != y_pred).sum()))

print('Training and testing on train with SVM')
clf = svm.SVC()
clf.fit(trainData.todense(), targetTrain)
y_pred = clf.predict(trainData.todense())
print("Number of mislabeled test points out of a
    total
        %d points : %d"
        % (trainData.shape[0],(targetTrain != y_pred
            ).sum()))

print('Testing on test with already trained SVM')
y_pred = clf.predict(testData.todense())
print("Number of mislabeled test points out of a
    total
       %d points : %d"
       % (testData.shape[0],(targetTest != y_pred).
           sum()))
```

Out[17]:
```
Training and testing on test Naive Bayes
Number of mislabeled training points out of a total 10 points : 0
Number of mislabeled test points out of a total 6 points : 2
Training and testing on train with SVM
Number of mislabeled test points out of a total 10 points : 0
Testing on test with already trained SVM
Number of mislabeled test points out of a total 6 points : 2
```

In this scenario both learning strategies achieve the same recognition rates in both training and test sets. Note that similar words are shared between tweets. In practice,

with real examples, tweets will include unstructured sentences and abbreviations, making recognition harder.

10.5 Conclusions

In this chapter, we have analyzed the problem of binary sentiment analysis of text data: data cleaning to remove irrelevant symbols, punctuation and tags; stemming in order to define the same root for different works with the same meaning in terms of sentiment; defining a dictionary of words (including n-grams); and representing text in terms of a feature space with the length of the dictionary. We have also seen codification in this feature space, based on normalized and weighted term frequencies. We have defined feature vectors that can be used by any machine learning technique in order to perform sentiment analysis (binary classification in the examples shown), and reviewed some useful Python packages, such as NLTK and Textblob, for sentiment analysis.

As discussed in the introduction of this chapter, we have only reviewed the sentiment analysis problem and described common procedures for performing the analysis resulting from a binary classification problem. Several open issues can be addressed in further research, such as the identification of sarcasm, a lack of text structure (as in tweets), many possible sentiment categories and degrees (not only binary but also multiclass, regression, and multilabel problems, among others), identification of the object of analysis, or subjective text, to name a few.

The tools described in this chapter can define a basis for dealing with those more challenging problems. One recent example of current state-of-the-art research is the work of [3], where deep learning architectures are used for sentiment analysis. Deep learning strategies are currently a powerful tool in the fields of pattern recognition, machine learning, and computer vision, among others; the main deep learning strategies are based on neural network architectures. In the work of [3], a deep learning model builds up a representation of whole sentences based on the sentence structure, and it computes the sentiment based on how words form the meaning of longer phrases. In the methods explained in this chapter, n-grams are the only features that capture those semantics. For further discussion in this field, the reader is referred to [4,5].

Acknowledgements This chapter was co-written by Sergio Escalera and Santi Seguí.

References

1. Z. Ren, J. Yuan, J. Meng, Z. Zhang, IEEE Transactions on Multimedia **15**(5), 1110 (2013)

2. A.L. Maas, R.E. Daly, P.T. Pham, D. Huang, A.Y. Ng, C. Potts, in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies* (Association for Computational Linguistics, Portland, Oregon, USA, 2011), pp. 142–150. URL http://www.aclweb.org/anthology/P11-1015
3. R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng, C. Potts, Conference on Empirical Methods in Natural Language Processing (2013)
4. E. Cambria, B. Schuller, Y. Xia, C. Havasi, IEEE Intelligent Systems **28**(2), 15 (2013)
5. B. Pang, L. Lee, Found. Trends Inf. Retr. **2**(1–2), 1 (2008)