Chapter 12

Automatic analysis of text

In earlier chapters, you learned about both supervised and unsupervised machine learning as well about dealing with texts. This chapter brings together these elements and discusses how to combine them to automatically analyze large corpora of texts. We begin with a very simple top-down approach in Section 12.1, in which we count occurrences of words from an *a priori* defined list of words. In Section 12.2, we still use pre-defined categories that we want to code, but let the machine "learn" the rules of the coding itself. Finally, in Section 12.3, we employ a bottom-up approach in which we do not use any *a priori* defined lists or coding schemes, but inductively extract topics from our data.

12.1. Dictionary approaches to text analysis

A straightforward way to automatically analyze text is to compile a list of terms you are interested in and simply count how often they occur in each document. For example, if you are interested to find out in how far mentions of political parties in news articles change over the years, you only need to compile a list of all party names and write a small script to count them.

Historically, this is how sentiment analysis was done. Example 12.1 shows how to

do a simple sentiment analysis based on a list of positive and negative words. The logic is straightforward: you count how often each positive word occurs in a text, you do the same for the negative words, and then determine which occur more often.

The example now shows two very differnet approaches. We probably could find a NLTK function or sth to make the python-example more R-like ... but on the other hand, it seems (but maybe that's me (Damian)) more flexible and extensible to build this up as a for loop, especially if you eventually want to support multi-word expressions or regex or so. TO BE DECIDED...

Python Code

```python
import requests
from glob import glob
import csv
import os
positive = requests.get('http://cssbook.net/d/positive.txt').
        text.split('\n')
negative = requests.get('http://cssbook.net/d/negative.txt').
        text.split('\n')

# unpack the dataset from https://ai.stanford.edu/~amaas/data
        /sentiment/aclImdb_v1.tar.gz and store the folder '
        aclImdb' in the same folder as this script
reviews = []
for file in glob(os.path.join('aclImdb','train','unsup','*.
        txt')):
    with open(file) as f:
        reviews.append(f.read())

sent = []
# we only take the first 100 reviews to speed things up
for review in reviews[:100]:
    words = review.split()
    number_of_pos_words = sum([sum([sentword==word for word
            in words]) for sentword in positive])
    number_of_neg_words = sum([sum([sentword==word for word
            in words]) for sentword in negative])
    if number_of_pos_words > number_of_neg_words:
        sent.append(1)
    elif number_of_pos_words < number_of_neg_words:
        sent.append(-1)
    else:
        sent.append(0)

print(f"Average sentiment: {sum(sent)/len(sent)}.")
print('Writing reviews and sentiment score to a csv file...')
with open('sentiment.csv',mode='w') as fo:
    writer = csv.writer(fo)
    writer.writerow(['review', 'sentiment'])
    writer.writerows(zip(reviews, sent))
```

R Code

```r
library(readtext)
library(tidyverse)
library(quanteda)

positive = scan('http://cssbook.net/d/positive.txt', what='
        list')
negative = scan('http://cssbook.net/d/negative.txt', what='
        list')

mydict = dictionary(list(pos=positive, neg=negative))


# unpack the dataset from https://ai.stanford.edu/~amaas/data
        /sentiment/aclImdb_v1.tar.gz and store the folder '
        aclImdb' in the same folder as this script
reviews = readtext(file.path('aclImdb','train','unsup','*.txt
        '))

# let's select only the first 100 reviews for now to speed
        things up
reviews = head(reviews, 100)


d = dfm(reviews$text)
reviews = convert(d %>% dfm_lookup(mydict), to="data.frame")

reviews = reviews %>%
  mutate(sent = case_when(
    pos > neg ~ "1",
    pos == neg ~ "0",
    pos < neg ~ "-1",
    ))

write.csv(reviews,'sentiment.csv')
```

Example 12.1: Different approaches to a simple dictionary-based sentiment analysis: counting and summing all words using a for-loop over all reviews (Python) versus constructing a term-document matrix and looking up the words in there (R). Note that both approaches would be possible in either language.

As you may already realize, there are a lot of downsides to this approach. Most notably, our bag-of-words approach does not allow us to allow for negation: "not

good" will be counted as positive. Relatedly, we cannot handle modifiers such as "very good". Also, all words are either positive or negative, while "great" should be more positive than "good". More advanced dictionary-based sentiment analysis packages like Vader [Hutto and Gilbert, 2014], or SentiStrength [Thelwall et al., 2012] include such functionality. Yet, as we will discuss in Section 12.2, also these off-the-shelf packages perform very poorly in many sentiment analysis tasks, especially outside of the domains they were developed for. Dictionary-based sentiment analysis has been shown to be problematic when analyzing news content [e.g. Gonzalez-Bailon and Paltoglou, 2015, Boukes et al., 2019]. They are problematic when accuracy on the sentence level is imporant, but may be satisfactory with longer texts for comparatively easy tasks such as movie review classification [**?**], where there is clear ground truth data and the genre convention implies that the whole text is evaluative and evaluates one object (the film).

Still, there are many use cases where dictionary approaches work very well. Because your list of words can contain anything, not just positive or negative words. Dictionary approaches have been used, for instance, to measure the use of racist words or swearwords in online fora. Dictionary approaches are simple to understand and straightforward, which can be a good argument for using them when it is important that the method is no black-box but fully transparent even for a layperson. Especially when the dictionary is already existing or easy to create, it is also a very cheap method. This goes at the expense, though, of their limitation to measuring easy to operationalize concepts. To put it bluntly: it's great for measuring the visibility of parties or organizations in the news, but it's not good for measuring frames.

What gave dictionary approaches a bit of a bad name is that many researchers applied them without validating them. This is especially problematic when a dictionary is applied in a slightly different domain than for which it was originally made.

If you want to use a dictionary-based approach, we advise the following procedure:

1. Construct a dictionary based on theoretical considerations and by closely reading

a sample of example texts.

2. Code some articles manually and compare with the automated coding.
3. Improve your dictionary and check again.
4. Manually code a validation dataset of sufficient size. The required size depends a bit on how balanced your data is – if one code occurs very infrequently, you will need more data.
5. Calculate the agreement. You could use standard intercoder reliability measures used in manual content analysis, but we would also advise you to calculate precision and recall (see Secion 9.4).

Very extensive dictionaries will have a high recall (it becomes increasingly unlikely that you "miss" a relevant document), but often suffer from low precision (more documents will contain one of the words even though they are irrelevant). Vice versa, a very short dictionary will often be very precise, but miss a lot of documents. It depends on your research question where the right balance lies, but to substantially interpret your results, you need to be able to quantify the performance of your dictionary-based approach.

12.2. Supervised text analysis: automatic classification and sentiment analysis

For many applications, there are good reasons to use the dictionary approach presented in the previous section. First, it is intuitively understandable also for lay people, and results can – in principle – even be varified by hand, which can be an advantage when transparency or communicatibility is of high importance. Second, it is very easy to use.

Dictionary approaches excel under three conditions: First, the variable we want to code is *manifest and concrete* rather than *latent and abstract*: names of actors, specific physical objects, specific phrases, etc., rather than feelings, frames, or topics. Second, all synonyms to be included must be known beforehand. And third, the dictionary entries must not have multiple meanings.

For instance, coding how often gun control is mentioned in political speeches fits these criteria. There are only so many ways to talk about it, and it is rather unlikely that speeches about other topics contain a phrase like "gun control". Similarily, if we want to find references to Angela Merkel, Donald Trump, or any other well-known politician, we can just directly search for their names – even though problems arise when people have very common surnames and are referred to by their surnames only.

Sadly, most interesting concepts are more complex to code. Take a seemingly straightforward problem: distinguishing whether a news article is about the economy or not. This is really easy to do for humans: there may be some edge cases, but in general, people rarely need longer than a second to grasp whether an article is about the economy rather than about sports, culture, etc. Yet, many of these article won't directy state that they are about the economy by explicitly using the word "economy".

We may think of extending our dictionary not only with `economi.*` (a regular expression that includes economists, economic, and so on), but also come up with other words like "stock exchange", "market", "company" – but wait. Here, we run into a problem that we also faced when we dicussed the precision-recall tradeoff in Section 9.4: The more terms we add to our dictionary, the more false positives we will get: articles about geographical space called "market", about some celebrity being seen in "company" of someone else, and so on.

From this example, we can conclude that often (1) it is easy for humans to decide to which class a text belongs, but (2) it is very hard for humans to come op with a list of words (or rules) on which their judgement is based.

Such a situation is the perfect use case for supervised machine learing: After all, it won't take us much time to annotate, say, 1000 articles based on whether they are

about the economy or not (probably this takes less time than thoroughly finetuning a list of words to in- or exclude); and the difficult part, deciding on the exact rules underlying the decision to classify an article as economic is done by the computer in seconds.

12.2.1. Putting together a workflow

With the knowledge we gained in previous chapters, it is not difficult to set up a supervised machine learning classifier to automatically determine, for instance, the topic of a news article.

Let us recap the building blocks that we need. In Chapter 9, you learned how to use different classifiers, how to evaluate them, and how to choose the best settings. However, in these examples, we used numerical data as features; now, we have text. In Chapter [INSERT REF TO CHAPTER 10/PROCESSING TEXT], you learned how to turn text into numerical features. And that's all we need to get started!

Typical examples for supervised machine learning in the analysis of communication include the classification of topics [e.g., Scharkow, 2011], frames [e.g., Burscher et al., 2014], user characteristics such as gender or ideology , or sentiment.

references

Let us consider the case of sentiment analysis more in detail. Classical sentiment analysis is done with a dictionary approach: you take a list of positive words, a list of negative words, and count what occurs more. Addtionally, one may attach a weight to each word, such that "perfect" gets a higher weight than "good", for instance. An obvious drawbacks include that these pure bag-of-words approaches cannot cope with negation ("not good") and intensifiers "very good"), which is why extensions have been developed that take these (and other features, such as punctuation) into accout [Thelwall et al., 2012, Hutto and Gilbert, 2014, De Smedt et al., 2012].

But while available off-the-shelf packages that implement these extended dictionary-based methods are very easy to use (in fact, they spit out a sentiment score with one single line of code), it is questionable how well they work in practice. After all, "sen-

timent" is not exactly a clear, manifest concept for which we can enumerate a list of words. It has been shown that results obtained with multiple of these packages correlate very poorly with each other and with human annotations .

add reference paper wouter

Consequently, it has been suggested that it is better to use supervised machine learning to automatically code the sentiment of texts [Gonzalez-Bailon and Paltoglou, 2015, Vermeer et al., 2019]. In particular, one may need to annotate a custom, own dataset: training a classifier on, for instance, movie reviews and then using it to predict sentiment in political texts violates the assumption that training set, test set, and the unlabeled data that are to be classified are (at least in principle and approximately) drawn from the same population.

To illustrate the workflow, we will use the ACL IMDB dataset, a large dataset that consists out of a training dataset of 25,000 movie reviews (out of which 12,500 positives ones and 12,500 negative ones) and an equally sized test dataset [Maas et al., 2011]. It can be downloaded at https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz

These data do not come in one file, but rather in a set of textfiles that are sorted in different folders named after the dataset they belong to (`test` or `train`) and their label (`pos` and `neg`). This means that we cannot simply use a pre-defined function to read them, but we need to think of a way of reading the content into a datastructure that we can use. One way of doing so is shown in Example 12.2.

Sparse versus dense matrices and why it matters a lot for choosing between R and Python for machine learning. In a document-term matrix, you would typically find a lot of zeros: most words do *not* appear in any given document. For instance, the reviews in the IMDB dataset contain more than 100,000 unique words. Hence, the matrix has more than 100,000 columns. Yet, most reviews only consist of a couple of hundreds of

Python Code

```python
import os
from glob import glob

# unpack the dataset from https://ai.stanford.edu/~amaas/data
        /sentiment/aclImdb_v1.tar.gz and store the folder '
        aclImdb' in the same folder as this script

def read_data(dataset):
    texts = []
    labels = []
    for label in ['pos', 'neg']:
        for file in glob(os.path.join('aclImdb',dataset,label
                ,'*.txt')):
            with open(file) as f:
                texts.append(f.read())
                labels.append(label)
    return texts, labels

X_train_fulltext, y_train = read_data('train')
X_test_fulltext, y_test= read_data('test')
```

R Code

```r
library(readtext)

# unpack the dataset from https://ai.stanford.edu/~amaas/data
        /sentiment/aclImdb_v1.tar.gz and store the folder '
        aclImdb' in the same folder as this script

train_pos = readtext(file.path('aclImdb','train','pos','*.txt
        '))
train_pos$label = 'pos'
train_neg = readtext(file.path('aclImdb','train','neg','*.txt
        '))
train_neg$label = 'neg'

X_train_fulltext = c(train_neg$text, train_pos$text)
y_train = c(train_neg$label, train_pos$label)

test_pos = readtext(file.path('aclImdb','test','pos','*.txt')
        )
test_pos$label = 'pos'

test_neg = readtext(file.path('aclImdb','test','neg','*.txt')
        )
test_neg$label = 'neg'

X_test_fulltext = c(test_neg$text, test_pos$text)
y_test = c(test_neg$label, test_pos$label)

# Cleaning up a bit...
rm(test_pos, test_neg, train_pos, train_neg)
```

Example 12.2: Reading the ACL IMDB dataset.

words. As a consequence, more than 99% of the cells in the table contain a zero. In a sparse matrix, we do not store all these zeros, but only store the values for cells that actually contain a value. This drastically reduces the memory needed. But even if you have a huge amount of memory, this does not solve the issue: In R, the number of cells in a matrix is limited to 2,147,483,647. It is therefore impossible to store a matrix with 100,000 features and 25,000 documents as a dense matrix. Unfortunately, many models that you can run via *caret* in R will convert your sparse document-term matrix to a dense matrix, and hence are effectively only usable for very small datasets. An alternative is using the *quanteda* package, which does use sparse matrices throughout. However, at the time of writing this book, quanteda only provides a very limited number of models. As all of these problems do not arise in *scikit-learn*, you may want to consider using

Python for many text classification tasks.

Let us now train our first classifier. We choose a Naïve Bayes classifier with a simple count vectorizer (Example 12.3). In the Python example, pay attention to the fitting of the vectorizer: we fit on the training data *and* transorm the training data with it, but we only transform the test data *without re-fitting the vectorizer*. Fitting, here, includes the decision which words to include (by definition, words that are not present in the training data are not included; but we could also choose additional constraints, such as exluding very rare or very common words), but also assigning an (internally used) identifier (variable name) to each word. If we would fit the classifier again, these would not be compatible any more. In R, the same is achieved in a slightly different way: Two term-document matrices are created independently, before they are matched in such a way that only the features that are presen t in the training matrix are retained in the test matrix.

Note: A word that is not present in the training data, but is present in the test data, is thus ignored. If you want to use the information such out-of-vocabulary words can entail (e.g., they may be synonyms), you need to use a word embedding approach

We do not necessarily expect this first model to be the best classifier we could come up with, but it provides us with a reasonable baseline. In fact, even without any further adjustments, it works reasonably well: precision is higher for positive reviews and recall is higher for negative reviews (classifying a positive review as negative happens twice as much as the reverse), but none of the values is concerningly low.

12.2.2. Finding the best classifier

https://github.com/quanteda/quanteda.textmodels seems to be under heavy development right now. In the CRAN version of quanteda, there is only a NB classifier present. Let's wait for a moment to check which textmodels will be available via quanteda and then incorporate them here.

Python Code

```python
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import confusion_matrix,
        classification_report

vectorizer = CountVectorizer(stop_words='english')
X_train = vectorizer.fit_transform(X_train_fulltext)
X_test = vectorizer.transform(X_test_fulltext)

nb = MultinomialNB()
nb.fit(X_train, y_train)

y_pred = nb.predict(X_test)

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

R Code

```r
library(quanteda)
library(glue)

X_train = dfm(X_train_fulltext)
# for the test data, we need to make sure that we use only
        features present in the train data
X_test = dfm_match(dfm(X_test_fulltext), features = featnames
        (X_train))

myclassifier = textmodel_nb(X_train, y_train)
y_pred = predict(myclassifier, newdata = X_test)

tab_class <- table(y_test, y_pred)
glue("Performance for predicting positive reviews:")
confusionMatrix(tab_class, mode = "everything", positive = '
        pos')
glue("Performance for predicting negative reviews:")
confusionMatrix(tab_class, mode = "everything", positive = '
        neg')
```

Output

```
[[10995 1505]
 [ 3003 9497]]
              precision   recall f1-score  support

         neg       0.79     0.88     0.83    12500
         pos       0.86     0.76     0.81    12500

   micro avg       0.82     0.82     0.82    25000
   macro avg       0.82     0.82     0.82    25000
weighted avg       0.82     0.82     0.82    25000
```

Example 12.3: Training a Naïve Bayes classifier with simple word counts as features

Let us start by comparing two simple classifiers we know (Naïve Bayes and Logistic Regression, see Section 9.3 and two vectorizers that transform our texts into two numerical representations that we know: word counts and $tf \cdot idf$ scores [ADD REF TO CHAPTER 11].

We can also tune some things in the vectorizer, such as filtering out stopwords, or specifying a minimum number (or proportion) of documents in which a word needs to occur in order to be included, or the maximum number (or proportion) of documents in which it is allowed to occur. For instance, it could make sense to say that a word that occurs in less than $n = 5$ documents is probably a spelling mistake or so unusual that it just unnecessarily bloats our feature matrix; and on the other hand, a word that is so common that it occurs in moret than 50% of all documents is so common that it does not help us to distinguish between different classes.

We can try all of these things out by hand by just re-running the code from

Example 12.3 and only changing the line in which the vectorizer is specified and the line in which the classifier is specified. There's nothing wrong with that, but it's a bit cumbersome and not too elegant if we want to do this for more than two or three sections. Especially in the Python world, copy-pasting essentially the same code is considered bad style, as it makes your code unnecessary long and increases the likelihood of errors creeping in when you, for instance, need to apply the same changes to multiple copies of the code. A more elegant approach is outlined in Example 12.4: We define a function that gives us a short summary of only the output we are interested in, and then use a for-loop to iterate over all configurations we want to evaluate, fits them and calls the function we defined before. In fact, with 23 lines of code, we manage to compare four different models, while we already needed 15 lines (in Example **??**) to evaluate only one model.

The output of this little example gives us already quite a bit of insight of how to tackle our specific classification tasks: First, we see that a $tf \cdot idf$ classifier seems to be slightly but consistently superior to a count classifier (this is often, but not always the case). Second, we see that the logistic regression performs better than the Naïve Bayes classifier (also this is often, but not always, the case). In particular, in our case, the logistic regression improved on the exessive misclassifcation of positive reviews as negative, and achieves a very balanced performance.

There may be instances where one nevertheless may want to use a Count Vectorizer with a Naïve Bayes classifier instead (especially if it is too computationally expensive to estimate the other model), but for now, we may settle on the best performing combination, logistic regression with a $tf \cdot idf$ vectorizer. You could also try fitting a Support Vector Machine instead, but we have little reason to believe that our data isn't linearly separable, which means that there is little reason to believe that the SVM will perform better. Given the good performance we already achieved, we decide to stick to the logistic regression for now.

We can now go as far as we like, include more models, use crossvalidation and gridsearch (see Section 9.4.3), etc. However, our workflow now consists of *two* steps:

Python Code

```
from sklearn import metrics
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression

def short_classification_report (y_test, y_pred):
    print("              \t precision \t recall")
    print(f"positive reviews:\t {metrics.precision_score(y_test,y_pred, pos_label='pos'):0.2f} \t\t {metrics.recall_score(y_test,
            y_pred, pos_label='pos'):0.2f}")
    print(f"negative reviews:\t {metrics.precision_score(y_test,y_pred, pos_label='neg'):0.2f} \t\t {metrics.recall_score(y_test,
            y_pred, pos_label='neg'):0.2f}")

configurations = [('NB with Count', CountVectorizer(min_df=5, max_df=.5), MultinomialNB()),
                ('NB with TfIdf', TfidfVectorizer(min_df=5, max_df=.5), MultinomialNB()),
                ('LogReg with Count', CountVectorizer(min_df=5, max_df=.5), LogisticRegression(solver='liblinear')),
                ('LogReg with TfIdf', TfidfVectorizer(min_df=5, max_df=.5), LogisticRegression(solver='liblinear'))]

for description, vectorizer, classifier in configurations:
    print(description)
    X_train = vectorizer.fit_transform(X_train_fulltext)
    X_test = vectorizer.transform(X_test_fulltext)
    classifier.fit(X_train, y_train)
    y_pred = classifier.predict(X_test)
    short_classification_report(y_test, y_pred)
    print('\n')
```

Output

```
NB with Count
                    precision     recall
positive reviews:     0.87          0.77
negative reviews:     0.79          0.88


NB with TfIdf
                    precision     recall
positive reviews:     0.87          0.78
negative reviews:     0.80          0.88


LogReg with Count
                    precision     recall
positive reviews:     0.87          0.85
negative reviews:     0.85          0.87


LogReg with TfIdf
                    precision     recall
positive reviews:     0.89          0.88
negative reviews:     0.88          0.89
```

Example 12.4: An example of a custom function to give a brief overview of the performance of four simple vectorizer-classifier combinations.

fitting/transforming our input data using a vectorizer, and fitting a classifier. To make things easier, in scikit-learn, both steps can be combined into a so-called pipe. Example **??** shows how the loop in Example **??** can be re-written using pipes (the result stays the same).

Such a pipeline lends itself very well for performing a gridsearch. Example 12.6 gives you an example. With `LogisticRegression?` and `TfIdfVectorizer?`, we can get a list of all possible hyperparameters that we may want to tune. For instance, these

Python Code

```
from sklearn.pipeline import make_pipeline

for description, vectorizer, classifier in configurations:
    print(description)
    pipe = make_pipeline(vectorizer, classifier)
    pipe.fit(X_train_fulltext, y_train)
    y_pred = pipe.predict(X_test_fulltext)
    short_classification_report(y_test, y_pred)
    print('\n')
```

Example 12.5: Instead of fitting vectorizer and classifier separately, they can be combined in a pipeline.

could be the minimum and maximum frequency for words to be included or whether we want to use only unigrams (single words) or also bigrams (combinations of two words, see section [ADD REFERENCE TO APPROPRIATE SECTIONS]). For the Logistic Regression, it may be the regularization hyperparameter C, which applies a penalty for too complex models. We can simply put all values for these parameters that we want to consider in a dictionary, with as key a string with the step of the pipeline followed by two underscores and the name of the hyperparameter, and a list of all values we want to consider as corresponding value.

The gridsearch procedure will then estimate all combinations of all values, using cross-validation (see Section 9.4). In our example, we have $2 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 24$ different models, and 24 models $\cdot 5$ folds $= 120$ models to estimate. Hence, it can take you some time to run the code.

We see that we could further improve our model to precision and recall values of .90, by excluding extremely infrequent and extremely frequent words), including both unigrams and bigrams (which, we may speculate, helps us accounting for the "not good" vs "not", "good" problem), and changing the default penalty of C=1 to C=100.

Let us, just for the sake of it, compare the performance of our model with an off-the-shelf sentiment analysis package, in this case Vader [Hutto and Gilbert, 2014]. For any text, it will directly estimate sentiment scores (more specifically, a positivity score, a negativity score, a neutrality score, and a compound measure that combines them), without any need to have training data. However, as Example 12.7 shows,

Python Code

```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline

pipe = Pipeline(steps = [('vectorizer', TfidfVectorizer()), ('classifier', LogisticRegression(solver='liblinear'))])
grid = {
    'vectorizer__ngram_range' : [(1,1), (1,2)],
    'vectorizer__max_df': [0.5, 1.0],
    'vectorizer__min_df': [0, 5],
    'classifier__C': [0.01, 1, 100]
}

search = GridSearchCV(estimator=pipe,
                      param_grid=grid,
                      scoring='accuracy', # all classes are balanced, let's just score on accuracy
                      cv=5,
                      n_jobs=-1, # use all cpus
                      verbose=10)
search.fit(X_train_fulltext, y_train)
print(f'Using these hyperparameters {search.best_params_}, we get the best performance:')
print(short_classification_report(y_test, search.predict(X_test_fulltext)))
```

Output

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Using these hyperparameters {'classifier__C': 100, 'vectorizer__max_df': 0.5, 'vectorizer__min_df': 5, 'vectorizer__ngram_range': (1, 2)
       }, we get the best performance:
                    precision     recall
positive reviews:     0.90          0.90
negative reviews:     0.90          0.90
None
```

Example 12.6: A gridsearch to find the best hyperparameters for a pipeline consisting of a vectorizer and a classifier. Note that we can tune any parameter that either the vectorizer or the classifier accepts as an input, not only the four hyperparameters we chose in this example.

such a method is clearly inferior to a supervised machine learning approach. While in almost all cases (except $n = 11$), Vader was able to make a choice (getting scores of 0 is a notorious problem in very short texts), precision and recall are clearly worse than even the simple baseline model we started with, and much worse than those of the final model we finished with. In fact, we miss half (!) of the negative reviews. There are probably very few applications in the analysis of communication in which we would find this acceptable. It is important to highlight that this is not because the off-the-shelf package we chose is a particularly bad one (on the contrary, it is actually comparatively good), but because of the inherent limitations of dictionary-based sentiment analysis.

We need to keep in mind, though, that with this dataset, we chose one of the easiest sentiment analysis tasks: a set of long, rather formal texts (compared to informal short

Python Code

```
from nltk.sentiment import vader

analyzer = vader.SentimentIntensityAnalyzer()
y_vader = []
for review in X_test_fulltext:
    sentiment = analyzer.polarity_scores(review)
    if sentiment['compound']>0:
        y_vader.append('pos')
    elif sentiment['compound']<0:
        y_vader.append('neg')
    else:
        y_vader.append('dont know')
print(confusion_matrix(y_test, y_vader))
print(classification_report(y_test, y_vader))
```

Output

```
[[   0    0    0]
 [   6 6696 5798]
 [   5 1751 10744]]
              precision   recall f1-score  support

   dont know      0.00     0.00     0.00        0
         neg      0.79     0.54     0.64    12500
         pos      0.65     0.86     0.74    12500

   micro avg      0.70     0.70     0.70    25000
   macro avg      0.48     0.47     0.46    25000
weighted avg      0.72     0.70     0.69    25000
```

Example 12.7: For the sake of comparison, we calculate how an off-the-shelf sentiment analysis package would have performed in this task

social media messages), that evaluate exactly one entity (one film), and that are not ambigous at all. Many applications that communications scientists are interested in are much less straight-forward. Therefore, how tempting it may be to use an off-the-shelf package, doing so requires a thorough test based on at least some human-annotated data.

12.2.3. Using the model

So far, we have focused on training and evalualting models, almost forgetting why we were doing this in the first place: to use them to predict the label for new data that we did not annotate.

Of course, we could always re-train the model when we need to use it – but that has two downsides: first, as you may have seen, it may actually take considerable time

to train it, and second, you need to have the training data available, which may be a problem both in terms of storage space and of copyright and/or privacy if you want to share your classifier with others.

Therefore, it makes sense to save both our classifier and our vectorizer to a file, so that we can reload them later (Example 12.8). Keep in mind that you need not to re-use *both* – after all, the feature matrix (e.g., which features (=words) are included, and in which column which feature is) will be different when fitting a new vectorizer. Therefore, as you see, you do not do any fitting any more, and only use the `.transform()` method of the (already fitted) vectorizer and the `.predict()` method of the (already fitted) classifier.

Python Code

```python
import pickle
from sklearn.externals import joblib

# let's save the vectorizer and classifier from the first example
pickle.dump(vectorizer,open("myvectorizer.pkl",mode="wb"))
joblib.dump(nb, "myclassifier.pkl")

#Then, later on, instead of fitting a new vectorizer, you can simply load the old one and use it:

listwithnewdata = ['This is a great movie', 'I hated this one.', 'What an awful fail']

myvectorizer = pickle.load(open("myvectorizer.pkl",mode="rb"))
new_features = vectorizer.transform(listwithnewdata)

myclassifier = joblib.load("myclassifier.pkl")
predictions = myclassifier.predict(new_features)

for review, label in zip(listwithnewdata, predictions):
    print(f"'{review}' probably belongs to class '{label}'.")
```

R Code

```r
print('TO BE DONE')
```

Output

```
'This is a great movie' probably belongs to class 'pos'.
'I hated this one.' probably belongs to class 'neg'.
'What an awful fail' probably belongs to class 'neg'.
```

Example 12.8: Saving and loading a vectorizer and a classifier

Another thing that we might want to do is to get a better idea of the features that the model uses to arrive at its prediction; in our example, what actually characterizes the best and the worst reviews. Example 12.9 shows how this can be done in one line of code using eli5 – a model that aims to "*e*xplain [the model] *l*ike *I*'m *5* years old".

Here, we re-use the `pipe` we constructed earlier to provide both the vectorizer and the classifier to the eli5 – if we would have only provided the classifier, than the feature names would have been internal identifiers (which are meaningless to us) rather than human-readable words.

We can also use eli5 to explain how the classifier arrived at a prediction for a specific document, by using different shades of green and red to explain how much different features contributed to the classification, and in which direction (Example 12.10).

Include an example of RNN to text classification

12.3. Unsupervised text analysis: topic modeling [?]