L8 – Working with Text Data

- Text as a third kind of feature rather than:
  - Continuous features that describe a quantity
  - Categorical features that are items from a list
- Text data is usually represented as strings, made up of characters – clearly very different from the numeric features
- Many applications:
  - Classifying an email message as spam or a legitimate email
  - In customer service, we often want to find out if a message is a complaint or an inquiry

Types of Data Represented as Strings

- Four different kinds of string data:
  - Categorical data
  - Free strings that can be semantically mapped to categories
  - Structured string data
    - Manually entered values do not correspond to fixed categories
    - But still have some underlying structure, like addresses, names of places or people, dates, telephone numbers, or other identifiers
  - Text data (e.g., tweets, chat logs, hotel reviews & Wikipedia etc.)
    - Freeform text data that consists of phrases or sentences
    - For simplicity's sake, let’s assume all are in one language: English
    - In the content of text analysis, the dataset is often called the corpus
    - Each data point represented as a single text, is called a document

Example Application: Sentiment Analysis of Movie Reviews

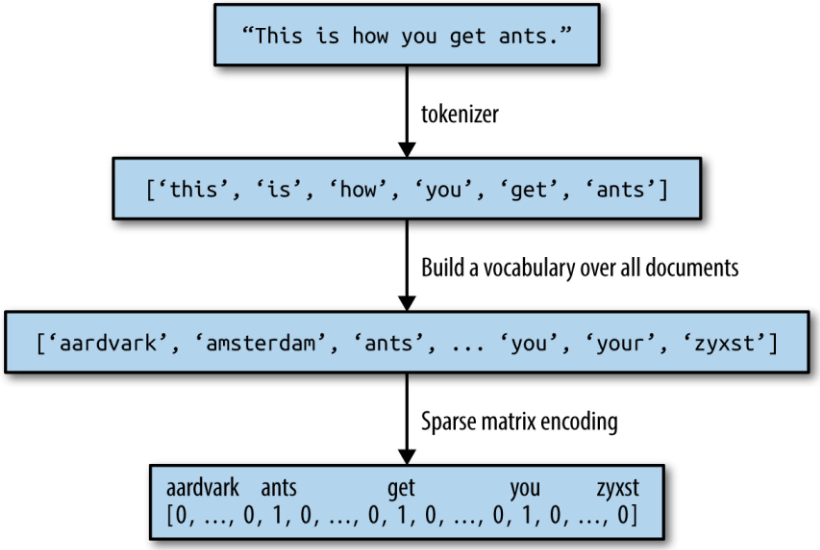
```
!wget -nc http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz -P data
!tar xzf data/aclImdb_v1.tar.gz --skip-old-files -C data

from sklearn.datasets import load_files
import numpy as np
# load_files returns a bunch, containing training texts and training labels
reviews_train = load_files("data/aclImdb/train/")
index = np.where(reviews_train.target!=2)[0]
text_train = [reviews_train.data[i] for i in index]
y_train = [reviews_train.target[i] for i in index]
# to remove the HTML line breaks <br />
text_train = [doc.replace(b"<br />", b" ") for doc in text_train]
print("type of text_train: {}".format(type(text_train)))
print("length of text_train: {}".format(len(text_train)))        print("text_train[6]:{}\n".format(text_train[6]))
np.unique(y_train)

print("Samples per class (training): {}".format(np.bincount(y_train)))
```

```
# load the test dataset in the same manner
reviews_test = load_files("data/aclImdb/test/")
text_test, y_test = reviews_test.data, reviews_test.target
print("Number of documents in test data: {}".format(len(text_test)))
print("Samples per class (test): {}".format(np.bincount(y_test)))
# to remove the HTML line breaks <br />
text_test = [doc.replace(b"<br />", b" ") for doc in text_test]
```

- The task we want to solve is as follows:
  - Given a review, we want to assign the label “positive” or “negative” based on the text content of the review
  - This is a standard binary classification problem
  - Difficulty: the text data is not in a format that a machine learning model can handle.
  - Solution: we need to convert the string representation of the text into a numeric representation that we can apply machine learning algorithms to.

Representing Text Data as a Bag of Words

- One of the most simple but effective & commonly used way
  - Discard most of the structure of the input text
  - Only count how often each word appears in each text
- Three steps for computing the bag-of-words representation:
  1. **Tokenization:** Split each document into the words that appear in it (called tokens);
  2. **Vocabulary building:** Collect a vocabulary of all words that appear in any of the documents and sort them in alphabetical;
  3. **Encoding:** For each document, count how often each of the words in the vocabulary appear in this document.
- Output is one vector of word counts for each document



Steps for Building the bag-of-words representation

- Applying Bag-of-Words to a Toy Dataset
  - The bag-of-words representation is implemented in CountVectorizer, which is a transformer
  - Let's apply it to a toy dataset, consisting of two samples

```
bards_words =["The fool doth think he is wise,",
    "but the wise man knows himself to be a fool"]

from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
vect.fit(bards_words)
```

- Fitting the CountVectorizer consists of the tokenization of the training data and building of the vocabulary
- We can access the vocabulary by the vocabulary_ attribute

```
print("Vocabulary size: {}".format(len(vect.vocabulary_)))
print("Vocabulary content:\n {}".format(vect.vocabulary_))
```

- To create the bag-of-words representation, we call the transform

```
bag_of_words = vect.transform(bards_words)
print("bag_of_words: {}".format(repr(bag_of_words)))
```

- The bag-of-words representation is stored in a SciPy sparse matrix that only stores the entries that are nonzero
- To print it to check, we convert it to a "dense" NumPy array, where the number indicates the word counts for each word

```
print("Dense representation of bag_of_words:\n{}".format(bag_of_words.toarray()))
```

Bag-of-Words for Movie Reviews

- Now we apply the method to the movie reviews
  - Construct the bag-of-words vector

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print("X_train:\n{}".format(repr(X_train)))
```

  - Let's look at the vocabulary in a bit more detail

```
feature_names = vect.get_feature_names()
print("Number of features: {}".format(len(feature_names)))
print("First 20 features:\n{}".format(feature_names[:20]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 2000th feature:\n{}".format(feature_names[::2000]))
```

  - Surprisely, the first 10 entries in the vocabulary are all numbers
  - Weeding out the meaningful from the nonmeaningful "words" is sometimes tricky

- Before we try to improve our feature extraction, let's obtain a quantitative measure of performance by actually building a classifier
  - For high-dimensional & sparse data like this, linear models like LogisticRegression often work best

```
!pip install mglearn
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
import numpy as np
scores = cross_val_score(LogisticRegression(solver='sag'), X_train, y_train, cv=5)
print("Mean cross-validation accuracy: {:.2f}".format(np.mean(scores)))
```

  - We obtain a mean cross-validation score, which indicates reasonable performance for a balanced binary classification task
  - Then turn the regularization parameter C by GridSearchCV

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(LogisticRegression(solver='sag'), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Best parameters: ", grid.best_params_)
```

- We then assess the generalization performance of this parameter setting on the test set

```
X_test = vect.transform(text_test)
print("Test score: {:.2f}".format(grid.score(X_test, y_test)))
```

- There are many words shown in very low count in the dataset, which are uninformative
- To remove uninformative features (like numbers, typos), we remove the tokens that appear in less than *k* documents
- The value of *k* can be set by the min_df parameter

```
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train with min_df: {}".format(repr(X_train)))
```

- We then check the first 50 and every 700 tokens as below

```
feature_names = vect.get_feature_names()
print("First 50 features:\n{}".format(feature_names[:50]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 700th feature:\n{}".format(feature_names[::700]))
```

- It’s found that the uninformative words are removed
- Let’s try to check the best validation accuracy by the grid search

```
grid = GridSearchCV(LogisticRegression(solver='sag'), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

- Stopwords: Another way to get rid of uninformative words
  - Using a language specific list of stopwords
  - Discarding words that appears too frequently
  - scikitlearn has a built-in list of English stopwords in the feature_extraction.text module

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print("Number of stop words: {}".format(len(ENGLISH_STOP_WORDS)))
print("First 10th stopword:\n{}".format(list(ENGLISH_STOP_WORDS)[:10]))
print("Every 10th stopword:\n{}".format(list(ENGLISH_STOP_WORDS)[::10]))
```

  - As a limited number, removing them from the document does only minor change but might lead to an improvement in performance

```
# Specifying stop_words="english" uses the built-in list.
# We could also augment it and pass our own.
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("X_train with stop words:\n{}".format(repr(X_train)))
```

- There are now 305 (27,271-26,966) fewer features in the dataset, which means that most but not all of the stopwords appeared
- Let's run the GridSearchCV now

```
grid = GridSearchCV(LogisticRegression(solver='sag'), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

Rescaling the Data with tf-idf

- Instead of dropping features that are unimportant, another approach is to rescale features
  - Using the *term frequency-inverse document frequency* (**tf-idf**)
  - The intuition
    - Give high weight to any term that appears often in a particular document but not in many documents in the dataset
    - If shown the above characteristic, it is likely to be very descriptive
  - scikit-learn implements the tf-idf method in two classes:
    - TfidfTransformer, which takes in the sparse matrix output produced by CountVectorizer and transforms it;
    - TfidfVectorizer, which takes in the text data and does both the bag-of-words feature extraction and the tf-idf transformation.

- The tf-idf score for word w in document d is given by:

$$\text{tfidf}(w, d) = \text{tf} * \log\left(\frac{N+1}{N_w+1}\right) + 1$$

  - N is the number of documents in the training set
  - N_w is the number of documents in the training set containing w
  - tf (the term frequency) is the number of times that the word w appears in the query document d
  - L2 normalization is applied after computing the tf-idf rep.
  - i.e., we rescale the representation of each document to have Euclidean length 1

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

- Although the result of regression is not improved too much, we can also inspect tf-idf to find which words are most important

```
vectorizer = grid.best_estimator_.named_steps["tfidfvectorizer"]
# transform the training dataset
X_train = vectorizer.transform(text_train)
# find maximum value for each of the features over the dataset
max_value = X_train.max(axis=0).toarray().ravel()
sorted_by_tfidf = max_value.argsort()
# get feature names
feature_names = np.array(vectorizer.get_feature_names())
print("Features with lowest tfidf:\n{}".format(feature_names[sorted_by_tfidf[:20]]))
print("Features with highest tfidf: \n{}".format(feature_names[sorted_by_tfidf[-20:]]))
```

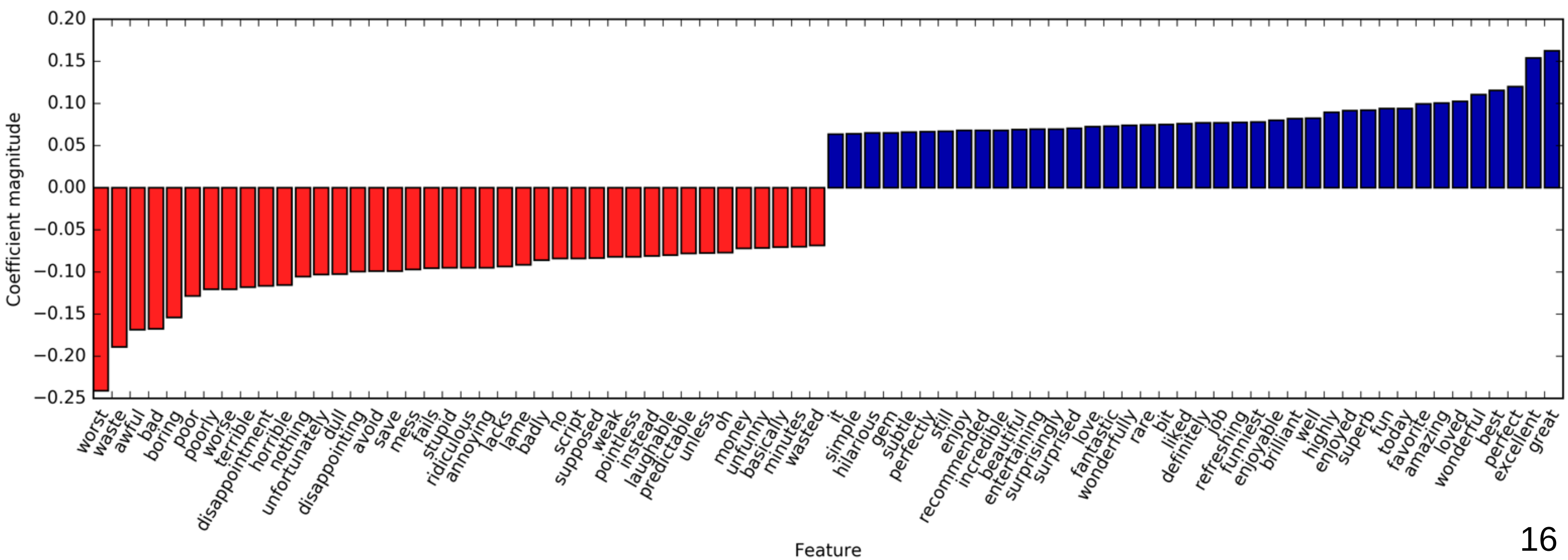
  - Features with low tf-idf are those that either are very commonly used across documents or are only used sparingly
  - Features with high tf-idf actually identify certain shows or movies

- Find those with low idf (i.e., appear frequently but less important)

```
sorted_by_idf = np.argsort(vectorizer.idf_)
print("Features with lowest idf:\n{}".format(feature_names[sorted_by_idf[:100]]))
```

- Let’s look in a bit more detail into coefficients of logistic regression
  - Look at the largest coefficients and see which words these correspond to
  - Both the negative and the positive coefficients are considered

```
!pip install mglearn


import mglearn
mglearn.tools.visualize_coefficients( grid.best_estimator_.named_steps["logisticregression"].coef_,
        feature_names, n_top_features=40)
```



Bag-of-Words with More than One Word (n-Grams)

- One of the main disadvantage of using a bag-of-words representation is that word order is completely discarded
  - Improved by not only considering the counts of single tokens but also the counts of pairs (*bigrams*) or triplets of tokens (*trigrams*)
  - By changing the ngram_range parameter of CountVectorizer or TfidfVectorizer
  - The ngram_range parameter is a tuple, consisting of the minimum and the maximum lengths

```
print("bards_words:\n{}".format(bards_words))

cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

- To look only at bigrams by setting ngram_range to (2,2)

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

- Using longer sequences of tokens usually results in many more features, and in more specific features

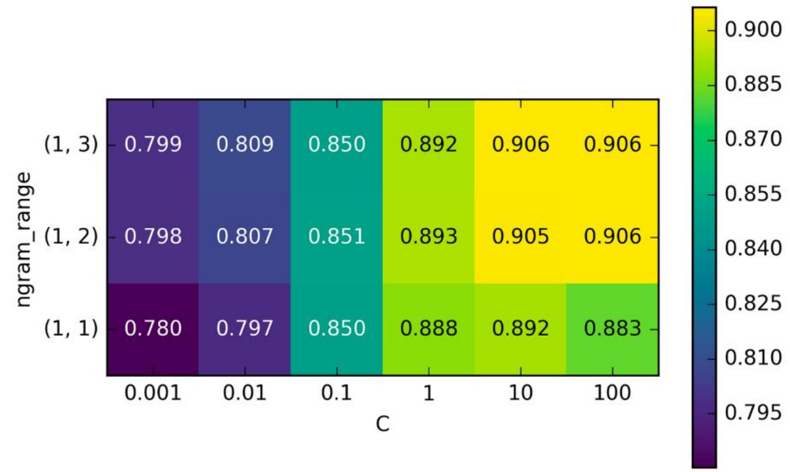
```
print("Transformed data (dense):\n{}".format(cv.transform(bards_words).toarray()))

cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

- For most applications, single words often capture a lot of meaning
  - Adding bigrams helps in most cases
  - Adding more n-grams might lead to overfitting
- Let's try out the TfidfVectorizer on the IMDb movie review data and find the best setting of n-gram range using a grid search

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
# running the grid search takes a long time because of the
# relatively large grid and the inclusion of trigrams
param_grid = {"logisticregression__C": [0.001, 0.01, 0.1, 1, 10, 100], "tfidfvectorizer__ngram_range": [(1, 1),
(1, 2), (1, 3)]}
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Best parameters:\n{}".format(grid.best_params_))
```

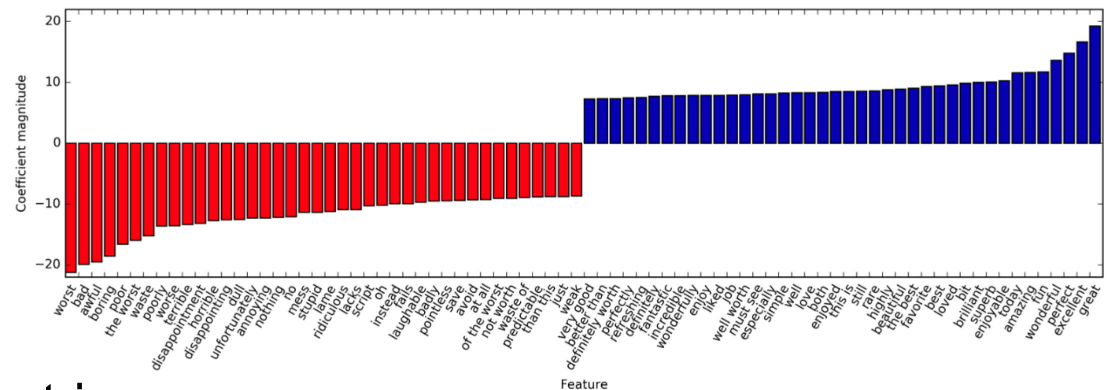
- Can visualize the cross-validation accuracy as a heat map

```
# extract scores from grid_search
scores = grid.cv_results_['mean_test_score'].reshape(-1, 3).T
# visualize heat map
heatmap = mglearn.tools.heatmap(scores, xlabel="C",
    ylabel="ngram_range", cmap="viridis", fmt="%.3f",
    xticklabels=param_grid['logisticregression__C'],
    yticklabels=param_grid['tfidfvectorizer__ngram_range'])
plt.colorbar(heatmap)
```



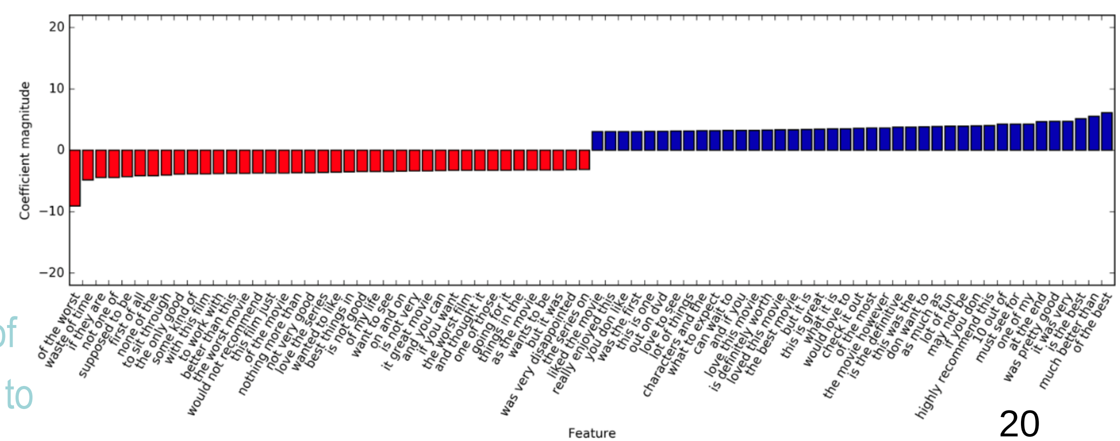
- Then, we can also visualize the important coefficient for the best model (including unigrams, bigrams, and trigrams)

# extract feature names and coefficients
vect = grid.best_estimator_.named_steps['tfidfvectorizer']
feature_names = np.array(vect.get_feature_names())
coef = grid.best_estimator_.named_steps['logisticregression'].coef_
mglearn.tools.visualize_coefficients(
    coef, feature_names,
    n_top_features=40)



– Next, we visualize only trigrams

# find 3-gram features
mask = np.array([len(feature.split(" ")) for feature in feature_names]) == 3
# visualize only 3-gram features
mglearn.tools.visualize_coefficients(
    coef.ravel()[mask],
    feature_names[mask],
    n_top_features=40)
# Many useful information but the impact of
# these features is quite limited compared to
# the importance of the unigram features



Topic Modeling and Document Clustering

- One particular technique that is often applied to text data
  - Describing the task of assigning each document to one or multiple topics, usually without supervision
  - For topic modeling, one decomposition method called *Latent Dirichlet Allocation* (often *LDA* for short) is often used
  - It is often good to remove very common words as they might otherwise dominate the analysis
  - We will limit the bag-of-word model to the 10,000 words after removing the top 15 percent

```
vect = CountVectorizer(max_features=10000, max_df=.15)
X = vect.fit_transform(text_train)
print("Shape of X: {}".format(X.shape))
```

- We then learn a model with 10 topics (setting “max_iter”)

```
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_components=10,
            learning_method="batch", max_iter=5, random_state=0)
# We build the model and transform the data in one step
# Computing transform takes some time,
# and we can save time by doing both at once
document_topics = lda.fit_transform(X)
```

- The size of components_ is (n_topics, n_words)

```
print("lda.components_.shape: {}".format(lda.components_.shape))
```

- The print_topics function provides a nice format for features

```
# For each topic (a row in the components_), sort the features (ascending)
# Invert rows with [:, ::-1] to make sorting descending
sorting = np.argsort(lda.components_, axis=1)[:, ::-1]
# Get the feature names from the vectorizer
feature_names = np.array(vect.get_feature_names())
# Print out the 10 topics:
mglearn.tools.print_topics(topics=range(10), feature_names=feature_names, sorting=sorting,
topics_per_chunk=5, n_words=10)
```

- Next, we will learn another model with 100 topics

```
lda100 = LatentDirichletAllocation(n_components=100,
            learning_method="batch", max_iter=5, random_state=0)
document_topics100 = lda100.fit_transform(X)
```

- Let’s select some interesting and representative topics to check

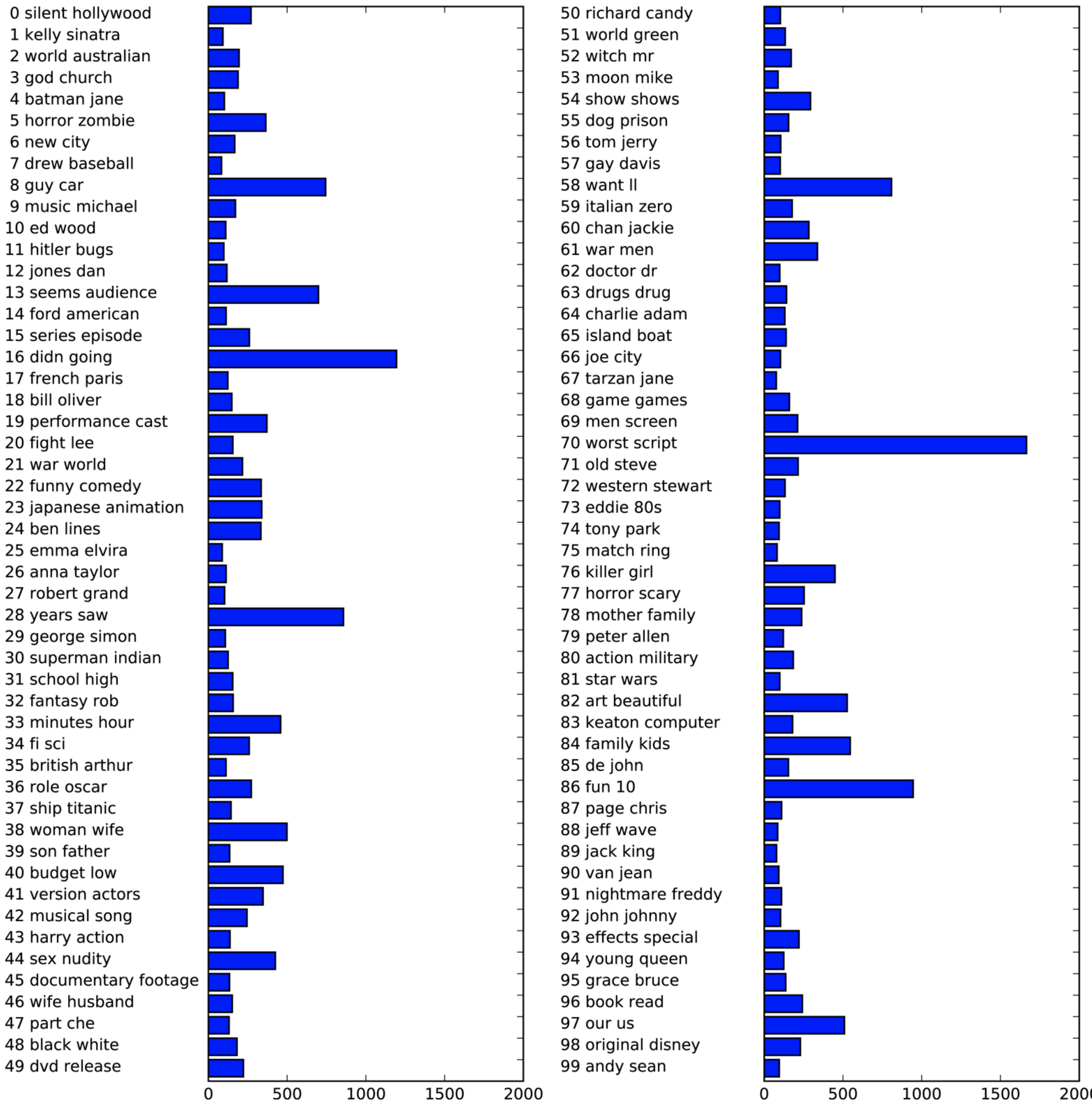
```
topics = np.array([7, 16, 24, 25, 28, 36, 37, 45, 51, 53, 54, 63, 89, 97])
sorting = np.argsort(lda100.components_, axis=1)[:, ::-1]
feature_names = np.array(vect.get_feature_names())
mglearn.tools.print_topics(topics=topics, feature_names=feature_names,
sorting=sorting, topics_per_chunk=5, n_words=20)
```

- Topic 45 seems about music, let’s check the review content

```
# sort by weight of "music" topic 45
music = np.argsort(document_topics100[:, 45])[::-1]
# print the five documents where the topic is most important
for i in music[:10]:
    # show first two sentences
    print(b".".join(text_train[i].split(b".")[:2]) + b".\n")
```

– Another interesting way to inspect the topics is to see how much weight each topic gets overall, by summing the document_topics over all reviews
– We name each topic by the two most common words

```
fig, ax = plt.subplots(1, 2, figsize=(10, 10))
topic_names = ["{:>2} ".format(i) + " ".join(words)
                    for i, words in enumerate(feature_names[sorting[:, :2]])]
# two column bar chart:
for col in [0, 1]:
        start = col * 50
        end = (col + 1) * 50
        ax[col].barh(np.arange(50), np.sum(document_topics100, axis=0)[start:end])
        ax[col].set_yticks(np.arange(50))
        ax[col].set_yticklabels(topic_names[start:end], ha="left", va="top")
        ax[col].invert_yaxis()
        ax[col].set_xlim(0, 2000)
        yax = ax[col].get_yaxis()
        yax.set_tick_params(pad=130)
plt.tight_layout()
```


0 silent hollywood
1 kelly sinatra
2 world australian
3 god church
4 batman jane
5 horror zombie
6 new city
7 drew baseball
8 guy car
9 music michael
10 ed wood
11 hitler bugs
12 jones dan
13 seems audience
14 ford american
15 series episode
16 didn going
17 french paris
18 bill oliver
19 performance cast
20 fight lee
21 war world
22 funny comedy
23 japanese animation
24 ben lines
25 emma elvira
26 anna taylor
27 robert grand
28 years saw
29 george simon
30 superman indian
31 school high
32 fantasy rob
33 minutes hour
34 fi sci
35 british arthur
36 role oscar
37 ship titanic
38 woman wife
39 son father
40 budget low
41 version actors
42 musical song
43 harry action
44 sex nudity
45 documentary footage
46 wife husband
47 part che
48 black white
49 dvd release
0
500
1000
1500
2000
50 richard candy
51 world green
52 witch mr
53 moon mike
54 show shows
55 dog prison
56 tom jerry
57 gay davis
58 want ll
59 italian zero
60 chan jackie
61 war men
62 doctor dr
63 drugs drug
64 charlie adam
65 island boat
66 joe city
67 tarzan jane
68 game games
69 men screen
70 worst script
71 old steve
72 western stewart
73 eddie 80s
74 tony park
75 match ring
76 killer girl
77 horror scary
78 mother family
79 peter allen
80 action military
81 star wars
82 art beautiful
83 keaton computer
84 family kids
85 de john
86 fun 10
87 page chris
88 jeff wave
89 jack king
90 van jean
91 nightmare freddy
92 john johnny
93 effects special
94 young queen
95 grace bruce
96 book read
97 our us
98 original disney
99 andy sean
0
500
1000
1500
2000

- Summary
  - Natural language and text processing is a large research field
  - For more advanced text-processing methods, try
    - the Python packages spacy (a relatively new but very efficient and well designed package),
    - nltk (a very well-established and complete but somewhat dated library),
    - and gensim (an NLP package with an emphasis on topic modeling)
  - There have been several very existing new developments
    - As implementation in word2vec library