ГЛАВА 7. РАБОТА С ТЕКСТОВЫМИ ДАННЫМИ

В главе 4 мы говорили о двух типах признаков, которые могут представлять свойства данных: непрерывных признаках, описывающих количество, и категориальных признаках, которые являются элементами фиксированного списка. Существует еще и третий тип признаков, который можно встретить в различных областях – текст. Например, при классификации сообщений электронной почты на спам и действительно нужные письма, сам по себе текст письма, безусловно, будет содержать важную информацию для данной классификационной задачи. Или нам нужно узнать мнение какого-то политика об иммиграции. В данном случае тексты его выступлений или твиты могут дать полезную информацию. При обслуживании клиентов нам часто требуется выяснить, является ли сообщение жалобой или запросом. Проанализировав тему и содержание сообщения, мы можем автоматически определить намерения клиента, а это в свою очередь позволит нам направить сообщение в соответствующий отдел или даже отправить клиенту автоматический ответ.

Текстовые данные обычно представлены в виде строк, состоящих из символов. Во всех приведенных примерах длина текстовых данных будет разной. Текстовая информация очень отличается от ранее рассмотренных нами непрерывных признаков, и нам сначала предстоит обработать данные, прежде чем мы сможем применить к ним алгоритмы машинного обучения.

Строковые типы данных

Прежде чем углубиться в вопросы предварительной обработки текстовых данных, предшествующие применению машинного обучения, мы хотим кратко рассмотреть различные типы текстовых данных, с которыми можно столкнуться на практике. Текст, как правило, представлен в наборе данных в виде обычной строки, однако далеко не все строковые признаки обрабатываются как текст. Строковый признак иногда может представлять собой категориальные переменные, о чем мы уже говорили в главе 5. Обработка строковых признаков невозможна без предварительного анализа данных.

На практике можно встретить четыре типа строковых данных:

- Категориальные данные
- Неструктурированные строки, которые по смыслу можно сгруппировать в категории
- Структурированные строки
- Текстовые данные

Категориальные данные (*categorical data*) представляют собой данные, которые берутся из фиксированного списка. Например, вы собираете данные с помощью онлайн-опроса, в ходе которого просите людей назвать их любимый цвет и для регистрации ответов используете выпадающий список из 8 значений («красный», «зеленый», «синий», «желтый», «черный», «белый», «фиолетовый» и «розовый»), позволяющий респондентам выбрать нужный вариант. В итоге мы получим восемь различных возможных значений, которые явно можно закодировать в виде категориальной переменной. Чтобы удостовериться в том, что строковые данные можно закодировать в виде категориальной переменной, достаточно взглянуть на них (если вы увидите очень много различных строковых значений, то скорее всего эти значения не являются категориальной переменной) и подтвердить свои предположения, вычислив количество уникальных значений по всему набору данных и, возможно, построив гистограмму частот встречаемости каждого значения. Кроме того, вам, возможно, потребуется проверить, соответствуют ли полученные категории интересущим вас категориям. Возможно в ходе опроса кто-то обнаружит, что слово «черный» написано с ошибкой как «ченый» и внесет соответствующую правку. В результате ваш набор данных будет содержать как значения «черный», так и значения «ченый», которые имеют одно и то же смысловое значение и должны быть объединены в одну категорию.

Теперь представьте себе, что вместо выпадающего меню вы позволите пользователям самим заполнять текстовое поле, отвечая на вопрос о любимом цвете. Многие, возможно, напишут ответы типа «черный» или «синий». Другие могут допустить орфографические ошибки, использовать различные варианты написания, например, «черный» или «чёрный», либо использовать более запоминающиеся и специфические названия типа «полуночно-синий». Кроме того, будут и вовсе странные названия. Несколько хороших примеров можно привести из XKCD Color Survey, где люди должны были назвать цвета и придумывали названия типа «VelociRaptor cloaka» и «оранжевый как кабинет моего стоматолога. Я до сих пор вспоминаю его перхоть, медленно залетающую в мой широко раскрытый рот», которые довольно трудно автоматически (или вообще) сопоставить определенным цветам. Ответы, записанные в текстовом поле, относятся ко второй категории списка, *неструктурированным строкам, которые по смыслу можно сгруппировать в категории* (*free strings that can be semantically mapped to categories*). Вероятно, лучшее всего закодировать их в виде категориальной переменной. Вы можете выбрать категории, выявив часто встречающиеся записи или задав категории, которые включат в себя содержательные

ответы. Возможно, вам потребуется задать несколько категорий и для вполне стандартных цветов, например, категорию «несколько цветов» для людей, которые дали ответы типа «зеленые и красные полоски», а также категорию «другое» для ответов, которые не могут быть закодированы иным способом. Предварительная обработка строковых данных может быть очень трудоемкой и ее сложно автоматизировать. Если у вас есть возможность повлиять на процесс сбора данных и требуется проанализировать понятия, которые лучше всего описываются с помощью категориальных переменных, настоятельно рекомендуем вам отказаться от ручной фиксации данных.

Как правило, строковые значения, введенные вручную, не соответствуют фиксированным категориям, но при этом все же имеют некоторую базовую *структуру* (*structure*), например, адреса, названия мест, имена и фамилии людей, даты, номера телефонов и другие идентификаторы. Этот тип строк очень трудно спарсить и их обработка сильно зависит от контекста и предметной области. Обработка подобных данных выходит за рамки этой книги.

Последняя категория строковых данных – это *текстовые данные* (*text data*), которые состоят из фраз или предложений. Примерами таких данных могут быть твиты, логи чата или отзывы о гостинице, а также собрание сочинений Шекспира, содержание Википедии или проекта «Гутенберг», включающего 50000 электронных книг. Все эти коллекции содержат информацию, представленную преимущественно в виде предложений, составленных из слов. [39] Для простоты давайте предположим, что все наши документы написаны на одном, английском языке.[40] С точки зрения анализа текста набор данных часто называют *корпусом* (*corpus*) и каждая точка данных, представленная в виде отдельного текста, называется *документом* (*document*). Эти термины берут свое начало из *информационного поиска* (*information retrieval, IR*) и *обработки естественного языка* (*natural language processing, NLP*), которые главным образом применяются при анализе текстовых данных.

Пример применения: анализ тональности киноотзывов

В качестве примера мы воспользуемся набором данных, который содержит киноотзывы, оставленные на сайте IMDb (Internet Movie

[39] Возможно, что контент веб-сайтов, упомянутых в твитах, содержит гораздо больше информации, чем текст самих твитов.

[40] Бо́льшая часть материала, которая будет изложена нами в оставшейся части этой главы, применима и к другим языкам, использующим латиницу, а также частично к языкам, в которых можно определить границы слов. В китайском языке границы слова на письме не обозначаются и поэтому возникают проблемы, которые затрудняют применение методов, изложенных в этой главе.

Database) и собранные исследователем Стэнфордского университета Эндрю Маасом.[41] Этот набор данных содержит тексты отзывов, а также метки, которые указывают тональность отзыва («положительный» или «отрицательный»). Сайт IMDb имеет собственную систему оценки фильмов от 1 до 10. Чтобы упростить процесс моделирования, эта система оценки будет сведена к двум классам, отзывы с оценкой 6 или выше помечаются как положительные, а остальные отзывы помечаются как отрицательные. Вопрос, касающийся качества подготовки исходных данных, мы оставляем открытым и просто используем данные в том виде, в каком их нам предоставил Эндрю Маас.

После распаковки набор данных представляет собой две отдельные папки с текстовыми файлами, одна папка – для обучения, а вторая – для тестирования. Каждая папка в свою очередь содержит две подпапки, одна называется *pos*, а другая – *neg*:

```
In[2]:
!tree -L 2 C:/Data/aclImdb

Out[2]:
C:/Data/aclImdb
├── test
│   ├── neg
│   └── pos
└── train
    ├── neg
    └── pos
6 directories, 0 files
```

Папка *pos* содержит все положительные отзывы, каждый отзыв записан в виде отдельного текстового файла, папка *neg* содержит все отрицательные отзывы и так же каждый отзыв представлен в виде отдельного текстового файла. В библиотеке `scikit-learn` есть вспомогательная функция `load_files`. Она позволяет загрузить файлы, для хранения которых используется такая структура папок, где каждая вложенная папка соответствует определенной метке. Сначала мы применим функцию `load_files` к обучающим данным:

```
In[3]:
from sklearn.datasets import load_files
reviews_train = load_files("C:/Data/aclImdb/train/")
# load_files возвращает коллекцию, содержащую обучающие тексты и обучающие метки
text_train, y_train = reviews_train.data, reviews_train.target
print("тип text_train: {}".format(type(text_train)))
print("длина text_train: {}".format(len(text_train)))
print("text_train[1]:\n{}".format(text_train[1]))

Out[3]:
тип text_train: <class 'list'>
длина text_train: 25000
text_train[1]:
b'Words can\'t describe how bad this movie is. I can\'t explain it by writing
 only. You have too see it for yourself to get at grip of how horrible a movie
 really can be. Not that I recommend you to do that. There are so many
 clich\xc3\xa9s, mistakes (and all other negative things you can imagine) here
```

[41] Набор данных доступен по ссылке http://ai.stanford.edu/~amaas/data/sentiment/.

```
that will just make you cry. To start with the technical first, there are a
LOT of mistakes regarding the airplane. I won\'t list them here, but just
mention the coloring of the plane. They didn\'t even manage to show an
airliner in the colors of a fictional airline, but instead used a 747
painted in the original Boeing livery. Very bad. The plot is stupid and has
been done many times before, only much, much better. There are so many
ridiculous moments here that i lost count of it really early. Also, I was on
the bad guys\' side all the time in the movie, because the good guys were so
stupid. "Executive Decision" should without a doubt be you\'re choice over
this one, even the "Turbulence"-movies are better. In fact, every other
movie in the world is better than this one.'
```

Видно, что `text_train` представляет собой список длиной 25000, в котором каждый элемент представляет собой строку, содержащую отзыв. Мы напечатали отзыв с индексом 1. Кроме того, можно увидеть, что отзыв содержит разрывы строк HTML (
). Хотя эти разрывы вряд ли сильно повлияют на модель машинного обучения, лучше выполнить очистку данных и удалить символы форматирования перед тем, как начать работу.

```
In[4]:
text_train = [doc.replace(b"<br />", b" ") for doc in text_train]
```

Тип элементов `text_train` будет зависеть от вашей версии Python. В Python 3 они будут иметь тип `bytes`, который представляет собой двоичное кодирование строковых данных. В Python 2 `text_train` состоит из строк. В данном случае мы не будем вдаваться в подробности различных строковых типов данных, имеющихся в Python, но мы рекомендуем вам прочитать разделы документации по Python 2 и/или Python 3, касающиеся строковых данных и Unicode.

Набор данных был собран таким образом, чтобы положительный и отрицательный классы были сбалансированы, поэтому количество строк с положительными отзывами и количество строк с отрицательными отзывами одинаковое:

```
In[5]:
print("Количество примеров на класс (обучение): {}".format(np.bincount(y_train)))

Out[5]:
Количество примеров на класс (обучение): [12500 12500]
```

Аналогичным образом загружаем тестовые данные:

```
In[6]:
reviews_test = load_files("C:/Data/aclImdb/test/")
text_test, y_test = reviews_test.data, reviews_test.target
print("Количество документов в текстовых данных: {}".format(len(text_test)))
print("Количество примеров на класс (тест): {}".format(np.bincount(y_test)))
text_test = [doc.replace(b"<br />", b" ") for doc in text_test]

Out[6]:
Количество документов в текстовых данных: 25000
Количество примеров на класс (тест): [12500 12500]
```

Задача, которую мы хотим решить, можно сформулировать следующим образом: каждому отзыву нам нужно присвоить метку «положительный» или «отрицательный» на основе текста отзыва. Это стандартная задача бинарной классификации. Однако текстовые данные представлены в формате, который модель машинного обучения не умеет обрабатывать. Нам нужно преобразовать строковое представление текста в числовое представление, к которому можно применить алгоритмы машинного обучения.

Представление текстовых данных в виде «мешка слов»

Один из самых простых, но эффективных и широко используемых способов подготовки текста для машинного обучения – это представление текстовой информации в виде «мешка слов» (*bag-of-words*). Используя это представление, мы удаляем структуру исходного текста, например, главы, параграфы, предложения, форматирование, и лишь подсчитываем частоту встречаемости каждого слова в каждом документе корпуса. Удаление структуры и подсчет частоты каждого слова позволяет получить образное представление текста в виде «мешка слов». Получение представления «мешок слов» включает следующие три этапов:

1. *Токенизация* (*tokenization*). Разбиваем каждый документ на слова, которые встречаются в нем (*токены*), например, с помощью пробелов и знаков пунктуации.
2. *Построение словаря* (*vocabulary building*). Собираем словарь всех слов, которые появляются в любом из документов, и пронумеровываем их (например, в алфавитном порядке).
3. *Создание разреженной матрицы* (*sparse matrix encoding*). Для каждого документа подсчитываем, как часто каждое из слов, занесенное в словарь, встречается в документе.

Этапы 1 и 2 имеют некоторые нюансы, которые мы обсудим подробнее чуть ниже. Сейчас давайте посмотрим, как мы можем применить обработку данных «мешок слов», используя `scikit-learn`. Рис. 7.1 иллюстрирует процесс на примере строки `"This is how you get ants"`. В итоге каждый документ можно представить в виде вектора частот слов. Для каждого слова, записанного в словаре, мы подсчитываем частоту его встречаемости в каждом документе. Это означает, что в нашем числовом представлении каждый признак соответствуют каждому уникальному слову набора данных. Обратите внимание, порядок слов в исходной строке абсолютно не имеет никакого значения для представления признаков «мешок слов».

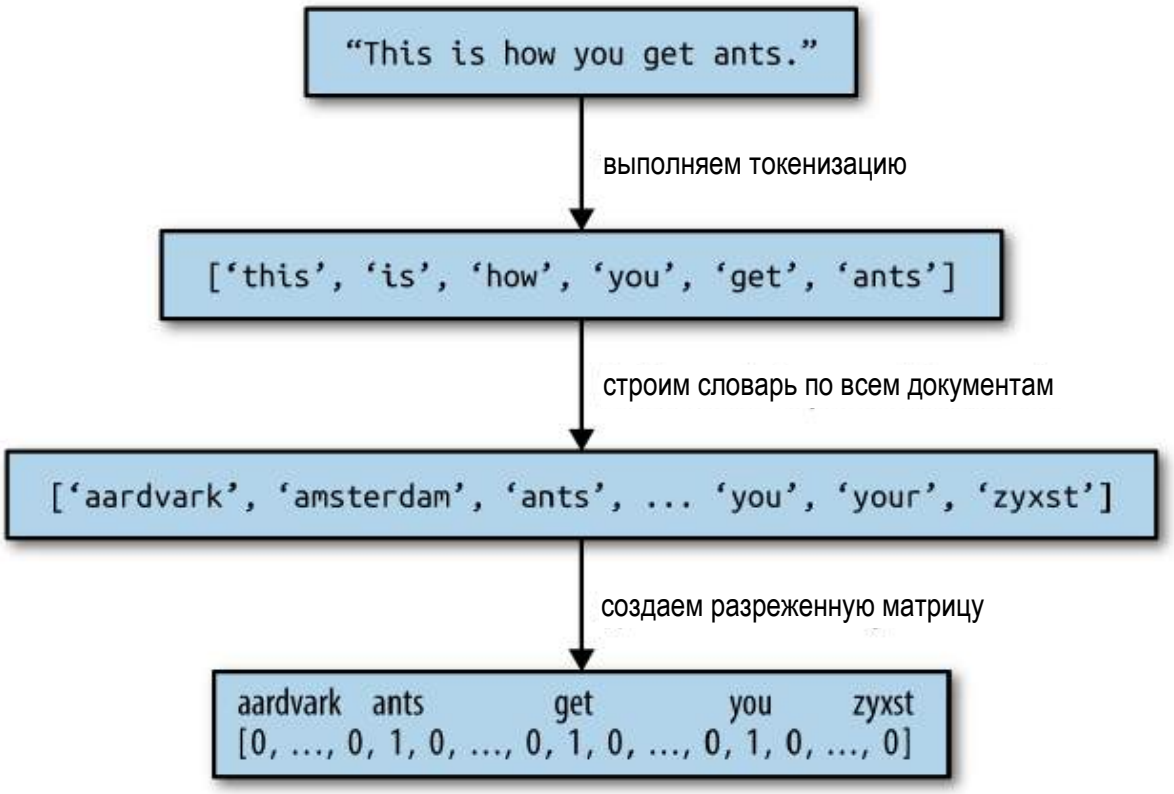

Рис. 7.1 Обработка «мешок слов»

Применение модели «мешка слов» к синтетическому набору данных

Модель «мешка слов» реализована в классе `CountVectorizer`, который выполняет соответствующее преобразование. Давайте сначала применим `CountVectorizer` к синтетическому набору данных, состоящему из двух примеров, чтобы проиллюстрировать его работу:

```
In[7]:
bards_words =["The fool doth think he is wise,",
              "but the wise man knows himself to be a fool"]
```

Мы импортируем класс `CountVectorizer`, создам экземпляр класса и подгоняем модель к нашим синтетическим данным следующим образом:

```
In[8]:
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
vect.fit(bards_words)
```

Процесс подгонки `CountVectorizer` включает в себя токенизацию обучающих данных и построение словаря, к которому мы можем получить доступ с помощью атрибута `vocabulary_`:

```
In[9]:
print("Размер словаря: {}".format(len(vect.vocabulary_)))
print("Содержимое словаря:\n {}".format(vect.vocabulary_))

Out[9]:
Размер словаря: 13
Содержимое словаря:
 {'the': 9, 'himself': 5, 'wise': 12, 'he': 4, 'doth': 2, 'to': 11, 'knows': 7,
  'man': 8, 'fool': 3, 'is': 6, 'be': 0, 'think': 10, 'but': 1}
```

Словарь состоит из 13 слов, начинается со слова «be» и заканчивается словом «wise».

Чтобы получить представление «мешок слов» для обучающих данных, мы вызываем метод `transform`:

```
In[10]:
bag_of_words = vect.transform(bards_words)
print("bag_of_words: {}".format(repr(bag_of_words)))

Out[10]:
bag_of_words: <2x13 sparse matrix of type '<class 'numpy.int64'>'
    with 16 stored elements in Compressed Sparse Row format>
```

Представление «мешок слов» записывается в разреженной матрице SciPy, которая хранит только ненулевые элементы (см. главу 1). Матрица имеет форму 2 x 13, в ней каждая строка соответствует точке данных, а каждый столбец (признак) соответствуют слову, записанному в словаре. Использование разреженной матрицы обусловлено тем, что документы, как правило, содержат лишь небольшое количество слов, записываемое в словарь, таким образом, большая часть элементов массива будет равна 0. Подумайте о том, сколько различных слов может встретиться в киноотзыве, учитывая словарный запас английского языка. Хранение всех этих нулей – ненужные затраты памяти. Чтобы взглянуть на фактическое содержимое разреженной матрицы, мы можем преобразовать ее в «плотный» массив NumPy (который помимо ненулевых элементов также хранит все нулевые элементы) с помощью метода `toarray`:[42]

```
In[11]:
print("Плотное представление bag_of_words:\n{}".format(
    bag_of_words.toarray()))

Out[11]:
Плотное представление bag_of_words:
[[0 0 1 1 1 0 1 0 0 1 1 0 1]
 [1 1 0 1 0 1 0 1 1 1 0 1 1]]
```

Видно, что частоты слов равны либо 0, либо 1. Ни одна из двух строк массива `bards_words` не содержит слов, которые встречались бы дважды. Давайте разберемся, как нужно работать с этими векторами признаков.

[42] Это возможно благодаря тому, что мы используем небольшой синтетический набор данных, который содержит лишь 13 слов. Если бы мы взяли реальный набор данных, то получили бы исключение `MemoryError`.

Первая строка (`"The fool doth think he is wise"`) соответствует первому ряду элементов, и слово `"be"`, записанное в словаре первым, встречается в ней ноль раз. Второе слово `"but"` тоже встречается в этой строке ноль раз. Третье слово `"doth"` встречается один раз. Взглянув на оба ряда, мы можем увидеть, что четвертое слово `"fool"`, десятое слово `"the"` и тринадцатое слово `"wise"` встречаются в обеих строках.

Модель «мешка слов» для киноотзывов

Теперь, когда мы детально разобрали процесс построения модели «мешка слов», давайте применим ее для анализа тональности киноотзывов. Ранее мы загрузили наши обучающие и тестовые данные, сформированные на основе отзывов IMDb, в виде списков строк (`text_train` и `text_test`) и сейчас обработаем их:

```
In[12]:
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print("X_train:\n{}".format(repr(X_train)))

Out[12]:
X_train:
<25000x74849 sparse matrix of type '<class 'numpy.int64'>'
    with 3431196 stored elements in Compressed Sparse Row format>
```

Матрица `X_train` соответствует обучающим данным, представленным в виде «мешка слов». Она имеет форму 25000 x 74849, указывая на то, что словарь включает 74849 элементов. Как видим, данные снова записаны в виде разреженной матрицы SciPy. Давайте более детально исследуем словарь. Еще один способ получить доступ к словарю – это использование метода `get_feature_name`. Он возвращает удобный список, в котором каждый элемент соответствует одному признаку:

```
In[13]:
feature_names = vect.get_feature_names()
print("Количество признаков: {}".format(len(feature_names)))
print("Первые 20 признаков:\n{}".format(feature_names[:20]))
print("Признаки с 20010 по 20030:\n{}".format(feature_names[20010:20030]))
print("Каждый 2000-й признак:\n{}".format(feature_names[::2000]))

Out[13]:
Количество признаков: 74849
Первые 20 признаков:
['00', '000', '0000000000001', '00001', '00015', '000s', '001', '003830',
 '006', '007', '0079', '0080', '0083', '0093638', '00am', '00pm', '00s',
 '01', '01pm', '02']
Признаки с 20010 до 20030:
['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'drawback',
 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings', 'drawl',
 'drawled', 'drawling', 'drawn', 'draws', 'draza', 'dre', 'drea']
Каждый 2000-й признак:
['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bete', 'chicanery',
 'condensing', 'cunning', 'detox', 'draper', 'enshrined', 'favorit', 'freezer',
 'goldman', 'hasan', 'huitieme', 'intelligible', 'kantrowitz', 'lawful',
 'maars', 'megalunged', 'mostey', 'norrland', 'padilla', 'pincher',
 'promisingly', 'receptionist', 'rivals', 'schnaas', 'shunning', 'sparse',
```

```
 'subset', 'temptations', 'treatises', 'unproven', 'walkman', 'xylophonist']
```

Возможно, факт того, что первые 10 элементов словаря являются числами, немного удивителен. Все эти числа встречаются в отзывах и поэтому рассматриваются как слова. Большинство из этих чисел не имеют никакого смысла, за исключением числа `"007"`, которое скорее всего связано с фильмами о Джеймсе Бонде.[43] Выделение значимой информации из незначимых «слов» является иногда довольно сложной задачей. Далее мы находим в словаре ряд английских слов, начинающихся с `"dra"`. Можно заметить, что единственное число слов `"draught"`, `"drawback"` и `"drawer"` обрабатывается отдельно от их множественного числа. Данные термины очень тесно связаны между собой по смыслу и обработка этих слов как разных, соответствующих различным признакам, не может быть оптимальным решением.

Перед тем как мы пытаемся улучшить выделение признаков, давайте измерим качество модели, построив классификатор. У нас есть обучающие метки, хранящиеся в `y_train` и обучающие данные, представленные в виде «мешка слов» `X_train`, таким образом, мы можем обучить классификатор по этим данным. Как правило, для подобных высокоразмерных разреженных данных лучше всего работают линейные модели типа `LogisticRegression`.

Давайте сначала применим `LogisticRegression` с использованием перекрестной проверки:[44]

```
In[14]:
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
scores = cross_val_score(LogisticRegression(), X_train, y_train, cv=5)
print("Средняя правильность перекр проверки: {:.2f}".format(np.mean(scores)))

Out[14]:
Средняя правильность перекр проверки: 0.88
```

Мы получаем среднее значение правильности перекрестной проверки, равное 88%, что указывает на приемлемое качество модели для задачи сбалансированной бинарной классификации. Известно, что логистическая регрессия имеет параметр регуляризации `C`, который мы можем настроить с помощью перекрестной проверки:

```
In[15]:
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
```

[43] Беглый анализ данных подтверждает, что это действительно так. Попробуйте самостоятельно убедиться в этом!

[44] Внимательный читатель может заметить, что в данном случае мы нарушаем принципы перекрестной проверки, изложенные в главе 6. На самом деле используя настройки, установленные для `CountVectorizer` по умолчанию, мы не собираем какие-либо статистики, поэтому наши результаты достоверны. Использование конвейера с самого начала было бы наилучшим выбором, но мы отложим его ради простоты.

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Наилучшее значение перекрестной проверки: {:.2f}".format(grid.best_score_))
print("Наилучшие параметры: ", grid.best_params_)
```

```
Out[15]:
Наилучшее значение перекрестной проверки: 0.89
Наилучшие параметры: {'C': 0.1}
```

Используя `C=0.1`, мы получаем значение перекрестной проверки 89%. Теперь мы можем оценить на тестовом наборе обобщающую способность при использовании данной настройки параметра:

```
In[16]:
X_test = vect.transform(text_test)
print("Правильность на тестовом наборе: {:.2f}".format(grid.score(X_test, y_test)))
```

```
Out[16]:
Правильность на тестовом наборе: 0.88
```

Теперь, давайте посмотрим, можно ли улучшить процесс извлечения слов. `CountVectorizer` извлекает токены с помощью регулярных выражений. По умолчанию используется регулярное выражение `"\b\w\w+\b"`. Для тех, кто не знаком с регулярными выражениями, поясним: это выражение позволяет найти все последовательности символов, которые состоят как минимум из двух букв или цифр (`\w`) и отделены друг от друга границами слов (`\b`). Его не интересуют слова, состоящие из одного символа, сокращения типа «doesn't» или «bit.ly» оно разбивает на два слово, однако последовательность символов «h8ter» будет обработана как отдельное слово. Затем `CountVectorizer` преобразует все слова в строчные, поэтому слова «soon», «Soon» и «sOon» соответствуют одному и тому же токену (и, следовательно, одному и тому же признаку). Этот простой принцип достаточно хорошо работает на практике, однако, как мы уже видели ранее, можно получить массу неинформативных признаков (например, числа). Один из способов решить эту проблему – использовать только те токены, которые встречаются по крайней мере в двух документах (или по крайней мере в пяти документах и т.д.). Токен, который встретился только в одном документе, вряд ли встретится в тестовом наборе и поэтому бесполезен. С помощью параметра `min_df` мы можем задать минимальное количество документов, в котором должен появиться токен.

```
In[17]:
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train с min_df: {}".format(repr(X_train)))
```

```
Out[17]:
X_train с min_df: <25000x27271 sparse matrix of type '<class 'numpy.int64'>'
    with 3354014 stored elements in Compressed Sparse Row format>
```

Задав `min_df=5`, мы уменьшаем количество признаков до 27271, и если сравнить этот результат с предыдущим выводом, теперь мы используем лишь треть исходных признаков. Давайте снова взглянем на токены:

```
In[18]:
feature_names = vect.get_feature_names()

print("Первые 50 признаков:\n{}".format(feature_names[:50]))
print("Признаки с 20010 по 20030:\n{}".format(feature_names[20010:20030]))
print("Каждый 700-й признак:\n{}".format(feature_names[::700]))

Out[18]:
Первые 50 признаков:
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08',
 '09', '10', '100', '1000', '100th', '101', '102', '103', '104', '105', '107',
 '108', '10s', '10th', '11', '110', '112', '116', '117', '11th', '12', '120',
 '12th', '13', '135', '13th', '14', '140', '14th', '15', '150', '15th', '16',
 '160', '1600', '16mm', '16s', '16th']
Признаки с 20010 по 20030:
['repentance', 'repercussions', 'repertoire', 'repetition', 'repetitions',
 'repetitious', 'repetitive', 'rephrase', 'replace', 'replaced', 'replacement',
 'replaces', 'replacing', 'replay', 'replayable', 'replayed', 'replaying',
 'replays', 'replete', 'replica']
Каждый 700-й признак:
['00', 'affections', 'appropriately', 'barbra', 'blurbs', 'butchered',
 'cheese', 'commitment', 'courts', 'deconstructed', 'disgraceful', 'dvds',
 'eschews', 'fell', 'freezer', 'goriest', 'hauser', 'hungary', 'insinuate',
 'juggle', 'leering', 'maelstrom', 'messiah', 'music', 'occasional', 'parking',
 'pleasantville', 'pronunciation', 'recipient', 'reviews', 'sas', 'shea',
 'sneers', 'steiger', 'swastika', 'thrusting', 'tvs', 'vampyre', 'westerns']
```

Четко видно, что намного реже стали встречаться числа и, похоже, что исчезли некоторые странные или неправильно написанные слова. Давайте оценим качество нашей модели, вновь выполнив решатчатый поиск:

```
In[19]:
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Наилучшее значение перекр проверки: {:.2f}".format(grid.best_score_))

Out[19]:
Наилучшее значение перекр проверки: 0.89
```

Наилучшее значение правильности, полученное в ходе перекрестной проверки, по-прежнему равно 89%. Мы не смогли улучшить качество нашей модели, однако сокращение количества признаков ускорит предварительную обработку, а исключение бесполезных признаков, возможно, улучшит интерпретабельность модели.

Если метод `transform` класса `CountVectorizer` применяется для документа, который содержит слова, отсутствующие в обучающем наборе, эти слова будут проигнорированы, поскольку не являются частью словаря. Это не относится к проблеме классификации, так как невозможно узнать какую-либо информацию о словах,

отсутствующих в обучающих данных. Однако в некоторых прикладных задачах, например, для обнаружении спама, возможно, было бы полезным добавить признак, который бы фиксировал, сколько слов, отсутствующих в словаре, встретилось в каждом документе. Для этого вам необходимо задать `min_df`, в противном случае этот признак не будет использоваться в ходе обучения.

Стоп-слова

Еще один способ, с помощью которого мы можем избавиться от неинформативных слов – исключение слов, которые встречаются слишком часто, чтобы быть информативными. Существуют два основных подхода: использование списка стоп-слов (на основе соответствующего языка), или удаление слов, которые встречаются слишком часто. Библиотека `scikit-learn` предлагает встроенный список английских стоп-слов, реализованный в модуле `feature_extraction.text`:

```
In[20]:
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print("Количество стоп-слов: {}".format(len(ENGLISH_STOP_WORDS)))
print("Каждое 10-е стоп-слово:\n{}".format(list(ENGLISH_STOP_WORDS)[::10]))

Out[20]:
Количество стоп-слов: 318
Каждое 10-е стоп-слово:
['above', 'elsewhere', 'into', 'well', 'rather', 'fifteen', 'had', 'enough',
 'herein', 'should', 'third', 'although', 'more', 'this', 'none', 'seemed',
 'nobody', 'seems', 'he', 'also', 'fill', 'anyone', 'anything', 'me', 'the',
 'yet', 'go', 'seeming', 'front', 'beforehand', 'forty', 'i']
```

Очевидно, что удаление стоп-слов с помощью списка может уменьшить количество признаков лишь ровно на то количество, которое есть в списке (318), но, возможно, это позволит улучшить качество модели. Давайте попробуем:

```
In[21]:
# настройка stop_words="english" задает встроенный список стоп-слов.
# мы можем также расширить его и передать свой собственный.
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("X_train с использованием стоп-слов:\n{}".format(repr(X_train)))

Out[21]:
X_train с использованием стоп-слов:
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'
    with 2149958 stored elements in Compressed Sparse Row format>
```

Теперь у нас стало на 305 признаков меньше (количество признаков уменьшилось с 27271 до 26966). Это означает, что большинство стоп-слов (но не все) встретились в корпусе документов. Давайте снова запустим решетчатый поиск:

```
In[22]:
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Наилучшее значение перекр проверки: {:.2f}".format(grid.best_score_))

Out[22]:
Наилучшее значение перекр проверки: 0.88
```

При использовании стоп-слов качество модели чуть снизилось. Это не повод для беспокойства, однако учитывая тот факт, что исключение 305 признаков из более чем 27000 вряд ли сильно изменит качество или интерпретабельность модели, признаем, использование списка стоп-слов в данном случае бесполезно. Как правило, фиксированные списки могут быть полезны при работе с небольшими наборами данных. Небольшие наборы данных не имеют достаточного объема информации, позволяющего модели самостоятельно определить, какие слова являются стоп-словами. В качестве примера вы можете попробовать другой подход, исключив часто встречающиеся слова, задав опцию `max_df` для `CountVectorizer` и посмотреть, как это повлияет на количество признаков и качество модели.

Масштабирование данных с помощью tf-idf

Следующий подход вместо исключения несущественных признаков пытается масштабировать признаки в зависимости от степени их информативности. Одним из наиболее распространенных способов такого масштабирования является метод *частота термина-обратная частота документа* (*term frequency-inverse document frequency, tf-idf*). Идея этого метода заключается в том, чтобы присвоить большой вес термину, который часто встречается в конкретном документе, но при этом редко встречается в остальных документах корпуса. Если слово часто появляется в конкретном документе, но при этом редко встречается в остальных документах, оно, вероятно, будет описывать содержимое этого документа лучше. В библиотеке `scikit-learn` метод `tf-idf` реализован в двух классах: `TfidfTransformer`, который принимает на вход разреженную матрицу, полученную с помощью `CountVectorizer`, и преобразует ее, и `TfidfVectorizer`, который принимает на вход текстовые данные и выполняет как выделение признаков «мешок слов», так и преобразование tf-idf. Для преобразования tf-idf существует несколько вариантов взвешивания частот, о которых вы можете прочитать в Википедии. Значение tf-idf для слова w в документе d вычисляется с помощью классов `TfidfTransformer` и `TfidfVectorizer` по формуле:[45]

[45] Мы привели здесь эту формулу для большей ясности. Чтобы выполнить tf-idf преобразование, вам не обязательно помнить ее.

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

где N – это количество документов в обучающем наборе, N_w – это количество документов обучающего набора, в которых встретилось слово w, и tf (частота термина) – это частота встречаемости термина в запрашиваемом документе d (документе, который вы хотите преобразовать). Кроме того, оба класса применяют L2 нормализацию после того, как вычисляют представление tf-idf. Другими словами, они масштабируют векторизированное представление каждого документа к единичной евклидовой норме (длине). Подобное масштабирование означает, что длина документа (количество слов) не меняет его векторизованное представление.

Поскольку tf-idf фактически использует статистические свойства обучающих данных, мы воспользуемся конвейером (о котором рассказывалось в главе 6), чтобы убедиться в достоверности результатов нашего решетчатого поиска. Для этого пишем следующий программный код:

```
In[23]:
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5, norm=None),
                     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("Наилучшее значение перекр проверки: {:.2f}".format(grid.best_score_))

Out[23]:
Наилучшее значение перекр проверки: 0.89
```

Видно, что применение преобразования tf-idf вместо обычных частот слов дало определенное улучшение. Кроме того, мы можем выяснить, какие слова в результате преобразования tf-idf стали наиболее важными. Имейте в виду, что масштабирование tf-idf призвано найти слова, которые лучше всего дискриминируют документы, но при этом оно является методом неконтролируемого обучения. Таким образом, «важное» не обязательно должно быть связано с интересующими метками «положительный отзыв» и «отрицательный отзыв». Сначала мы извлекаем из конвейера наилучшую модель `TfidfVectorizer`, найденную с помощью решетчатого поиска:

```
In[24]:
vectorizer = grid.best_estimator_.named_steps["tfidfvectorizer"]
# преобразуем обучающий набор данных
X_train = vectorizer.transform(text_train)
# находим максимальное значение каждого признака по набору данных
```

```
max_value = X_train.max(axis=0).toarray().ravel()
sorted_by_tfidf = max_value.argsort()
# получаем имена признаков
feature_names = np.array(vectorizer.get_feature_names())

print("Признаки с наименьшими значениями tfidf:\n{}".format(
    feature_names[sorted_by_tfidf[:20]]))

print("Признаки с наибольшими значениями tfidf: \n{}".format(
    feature_names[sorted_by_tfidf[-20:]]))
```

```
Out[24]:
Признаки с наименьшими значениями tfidf:
['poignant' 'disagree' 'instantly' 'importantly' 'lacked' 'occurred'
 'currently' 'altogether' 'nearby' 'undoubtedly' 'directs' 'fond' 'stinker'
 'avoided' 'emphasis' 'commented' 'disappoint' 'realizing' 'downhill'
 'inane']
Признаки с наибольшими значениями tfidf:
['coop' 'homer' 'dillinger' 'hackenstein' 'gadget' 'taker' 'macarthur'
 'vargas' 'jesse' 'basket' 'dominick' 'the' 'victor' 'bridget' 'victoria'
 'khouri' 'zizek' 'rob' 'timon' 'titanic']
```

Признаки с низкими значениями tf-idf – это признаки, которые либо встречаются во многих документах, либо используются редко и только в очень длинных документах. Интересно отметить, что многие признаки с высокими значениями tf-idf на самом деле соответствуют названиям некоторых шоу или фильмов. Эти термины встречаются лишь в отзывах, посвященным конкретному шоу или франшизе, но при этом они встречаются в данных отзывах очень часто. Это очевидно для таких терминов как "`smallville`" и "`doodlebops`", но в нашем случае и вполне нейтральное слово "`scanners`" тоже относится к названию фильма. Эти слова вряд ли помогут нам классифицировать тональность отзывов (если только некоторые франшизы не оцениваются всеми зрителями положительно или отрицательно), однако они, разумеется, содержат много конкретной информации об отзывах.

Кроме того, мы можем найти слова, которые имеют низкое значение обратной частоты документа, то есть слова, которые встречаются часто и поэтому считаются менее важными. Значения обратной частоты документа, найденные для обучающего набора, хранятся в атрибуте `idf_`:

```
In[25]:
sorted_by_idf = np.argsort(vectorizer.idf_)
print("Признаки с наименьшими значениями idf:\n{}".format(
    feature_names[sorted_by_idf[:100]]))
```

```
Out[25]:
Признаки с наименьшими признаками idf:
['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with'
 'was' 'as' 'on' 'movie' 'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all'
 'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they' 'there' 'if' 'his' 'out'
 'just' 'about' 'he' 'or' 'has' 'what' 'some' 'good' 'can' 'more' 'when'
 'time' 'up' 'very' 'even' 'only' 'no' 'would' 'my' 'see' 'really' 'story'
 'which' 'well' 'had' 'me' 'than' 'much' 'their' 'get' 'were' 'other'
 'been' 'do' 'most' 'don' 'her' 'also' 'into' 'first' 'made' 'how' 'great'
 'because' 'will' 'people' 'make' 'way' 'could' 'we' 'bad' 'after' 'any'
 'too' 'then' 'them' 'she' 'watch' 'think' 'acting' 'movies' 'seen' 'its'
 'him']
```

Как и следовало ожидать, словами с низкими значениями idf стали английские стоп-слова типа "the" и "no". Но некоторые из них характерны для киноотзывов. Это слова типа "movie", "film", "time", "story" и так далее. Интересно, что в соответствии с метрикой tf-idf слова "good", "great" и "bad" также были отнесены к самым часто встречающимся и потому «самым нерелевантным» словам, хотя можно было бы ожидать, что они будут иметь очень важное значение для анализа тональности.

Исследование коэффициентов модели

И, наконец, давайте посмотрим чуть более детально на информацию, полученную с помощью модели логистической регрессии. Поскольку у нас имеется большое количество признаков (27271 после удаления малоинформативных слов), мы не можем посмотреть все коэффициенты сразу. Однако мы можем посмотреть на коэффициенты, получившие максимальные значения, а также сопоставить их словам. Мы воспользуемся последней построенной моделью на основе признаков tf-idf.

Следующая гистограмма (рис. 7.2) показывает 25 наибольших и 25 наименьших коэффициентов модели логистической регрессии, каждый столбик соответствует величине коэффициента:

```
In[26]:
mglearn.tools.visualize_coefficients(
    grid.best_estimator_.named_steps["logisticregression"].coef_,
    feature_names, n_top_features=40)
```

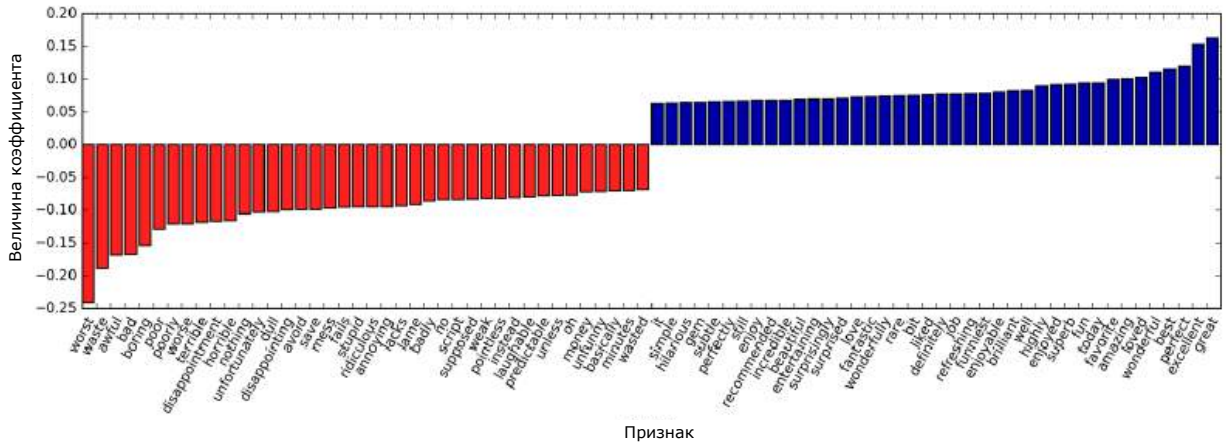

Рис. 7.2 Наибольшие и наименьшие значения коэффициентов логистической регрессии, построенной на основе признаков tf-idf

Отрицательные коэффициенты, расположенные в левой части гистограммы, относятся к словам, которые в соответствии с моделью указывают на негативные отзывы, а положительные коэффициенты,

расположенные в правой части гистограммы, принадлежат словам, которые означают положительные отзывы. Большая часть терминов интуитивно понятна, например, слова "worst", "waste", "disappointment" и "laughable" указывают на плохие киноотзывы, в то время как слова "excellent", "wonderful", "enjoyable" и "refreshing" свидетельствуют о положительных киноотзывах. Что касается слов типа "bit", "job" и "today", их связь с тональностью киноотзыва менее ясна, но они могут быть частью фразы, например, «good job» или «best today».

Модель «мешка слов» для последовательностей из нескольких слов (n-грамм)

Один из главных недостатков представления «мешок слов» заключается в полном игнорировании порядка слов. Таким образом, две строки «it's bad, not good at all» и «it's good, not bad at all» будут иметь одинаковое представление, хотя противоположны по смыслу. Употребление частицы «not» перед словом – это лишь один из примеров того, какое важное значение имеет контекст. К счастью, существует способ, позволяющий учитывать контекст при использовании представления «мешок слов», фиксируя не только частоты одиночных токенов, но и пары, тройки токенов, которые появляются рядом друг с другом. Пары токенов называют *биграммами* (*bigrams*), тройки токенов известны как *триграммы* (*trigrams*), а в более широком смысле последовательности токенов известны как *n-граммы* (*n-grams*). Мы можем изменить диапазон токенов, которые рассматриваются в качестве признаков, изменив параметр `ngram_range` для `CountVectorizer` или `TfidfVectorizer`. Параметр `ngram_range` задает нижнюю и верхнюю границы диапазона n-значений для различных извлекаемых n-грамм. Таким образом, будут использованы все значения n, которые удовлетворяют условию min_n <= n <= max_n. Ниже приводится пример на основе синтетических данных, использованных нами ранее:

```
In[27]:
print("bards_words:\n{}".format(bards_words))
```

```
Out[27]:
bards_words:
['The fool doth think he is wise,',
 'but the wise man knows himself to be a fool']
```

По умолчанию для каждой последовательности токенов с `min_n=1` и `max_n=1` (одиночные токены еще называются *юниграммами* или *unigramms*) `CountVectorizer` или `TfidfVectorizer` создает один признак:

```
In[28]:
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
```

```
print("Размер словаря: {}".format(len(cv.vocabulary_)))
print("Словарь:\n{}".format(cv.get_feature_names()))
```

```
Out[28]:
Размер словаря: 13
Словарь:
['be', 'but', 'doth', 'fool', 'he', 'himself', 'is', 'knows', 'man', 'the',
 'think', 'to', 'wise']
```

Чтобы посмотреть только биграммы, то есть последовательности из двух токенов, следующих друг за другом, мы можем задать `ngram_range` равным `(2, 2)`:

```
In[29]:
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print("Размер словаря: {}".format(len(cv.vocabulary_)))
print("Словарь:\n{}".format(cv.get_feature_names()))
```

```
Out[29]:
Размер словаря: 14
Словарь:
['be fool', 'but the', 'doth think', 'fool doth', 'he is', 'himself to',
 'is wise', 'knows himself', 'man knows', 'the fool', 'the wise',
 'think he', 'to be', 'wise man']
```

Использование более длинных последовательностей токенов, как правило, приводит к гораздо большему числу признаков и большей детализации признаков. Нет ни одной биграммы, которая встретилась бы в обоих строках массива `bard_words`:

```
In[30]:
print("Преобразованные данные (плотн):\n{}".format(cv.transform(bards_words).toarray()))
```

```
Out[30]:
Преобразованные данные (плотн):
[[0 0 1 1 1 0 1 0 0 1 0 1 0 0]
 [1 1 0 0 0 1 0 1 1 0 1 0 1 1]]
```

В большинстве прикладных задач минимальное количество токенов в последовательности должно быть равно единице, поскольку одиночные слова позволяют зафиксировать множество смысловых значений. Добавление биграмм помогает в большинстве случаев. Включение в анализ более длинных последовательностей, вплоть до 5-грамм, тоже, вероятно, поможет, но это вызовет взрывной рост количества признаков и может привести к переобучению, поскольку появится большое количество детализированных признаков. В принципе, количество биграмм может быть равно количеству юниграмм, возведенному в квадрат, а количество триграмм может быть равно количеству юниграмм в кубе, что приведет к очень большому пространству признаков. На практике количество сгенерированных n-грамм большей длины хоть и будет внушительным, но получится значительно меньше вышеназванных расчетных значений из-за структуры (английского) языка.

Теперь попробуем использовать для `bards_words` юниграммы, биграммы и триграммы:

```
In[31]:
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print("Размер словаря: {}".format(len(cv.vocabulary_)))
print("Словарь:\n{}".format(cv.get_feature_names()))

Out[31]:
Размер словаря: 39
Словарь:
['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think',
 'doth think he', 'fool', 'fool doth', 'fool doth think', 'he', 'he is',
 'he is wise', 'himself', 'himself to', 'himself to be', 'is', 'is wise',
 'knows', 'knows himself', 'knows himself to', 'man', 'man knows',
 'man knows himself', 'the', 'the fool', 'the fool doth', 'the wise',
 'the wise man', 'think', 'think he', 'think he is', 'to', 'to be',
 'to be fool', 'wise', 'wise man', 'wise man knows']
```

Давайте применим `TfidfVectorizer` к киноотзывам, собранных на сайте IMDb, и найдем оптимальное значение `ngram_range` с помощью решетчатого поиска:

```
In[32]:
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
# выполнение решетчатого поиска займет много времени из-за
# относительно большой сетки параметров и включения триграмм
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("Наилучшее значение перекр проверки: {:.2f}".format(grid.best_score_))
print("Наилучшие параметры:\n{}".format(grid.best_params_))

Out[32]:
Наилучшее значение перекр проверки: 0.91
Наилучшие параметры:
{'tfidfvectorizer__ngram_range': (1, 3), 'logisticregression__C': 100}
```

Из результатов видно, что мы улучшили качество чуть более чем на один процент, добавив биграммы и триграммы. Мы можем представить правильность перекрестной проверки в виде функции параметров `ngram_range` и `C`, использовав теплокарту, как это уже делали в главе 5 (см. рис. 7.3):

```
In[33]:
# извлекаем значения правильности, найденные в ходе решетчатого поиска
scores = grid.cv_results_['mean_test_score'].reshape(-1, 3).T
# визуализируем теплокарту
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)
```

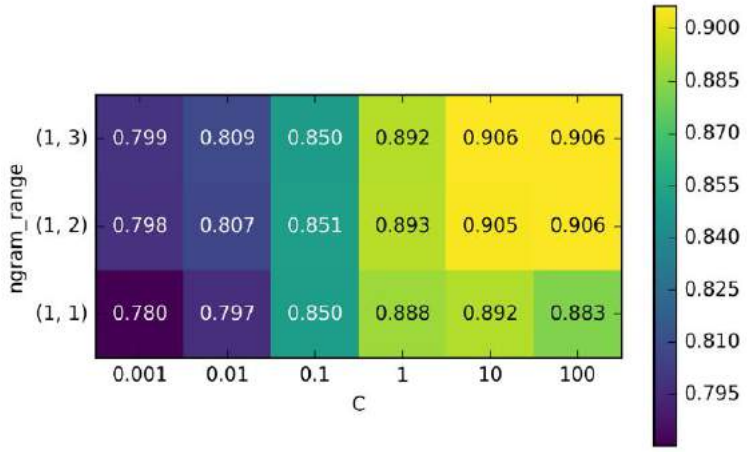

Рис. 7.3 Теплокарта для усредненной правильности перекрестной проверки, выраженной в виде функции параметров ngram_range и C

На теплокарте видно, что использование биграмм довольно значительно увеличивает качество модели, тогда как добавление триграмм дает очень небольшое преимущество с точки зрения правильности. Чтобы лучше понять, как повысилось качество модели, мы можем визуализировать наиболее важные коэффициенты наилучшей модели, которая включает юниграммы, биграммы и триграммы (см. рис. 7.4).

```
In[34]:
# извлекаем названия признаков и коэффициенты
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)
```

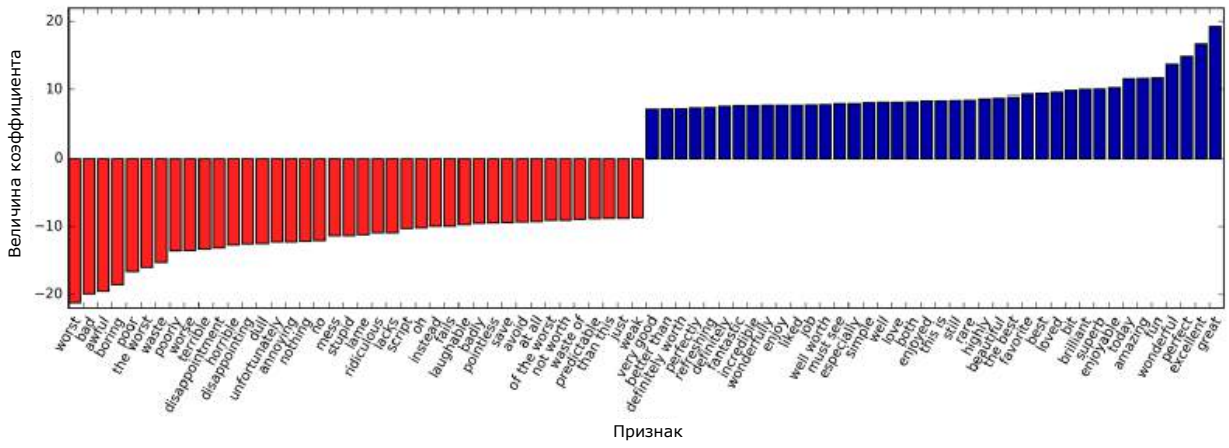

Рис. 7.4 Наиболее важные признаки, использовалось масштабирование tf-idf с включением юниграмм, биграмм и триграмм

Теперь у нас появились весьма интересные признаки со словом «worth», которые отсутствовали в юниграммной модели: `"not worth"`

указывает на отрицательный отзыв, в то время как "definitely worth" и "worth" свидетельствуют о положительном отзыве. Это яркий пример того, как контекст влияет на смысл слова «worth».

Далее мы визуализируем только триграммы, чтобы лучше понять, какие признаки являются полезными. Многие полезные биграммы и триграммы типа "none of the", "the only good", "on and on", "this is one", "of the most" и другие состоят из общеупотребимых слов, которые не были бы информативными сами по себе. Однако, как можно увидеть на рис. 7.5, влияние триграммных признаков по сравнению с важностью юниграммных признаков выражено гораздо слабее:

```
In[35]:
# находим триграммные признаки
mask = np.array([len(feature.split(" ")) for feature in feature_names]) == 3
# визуализируем только 3-граммные признаки
mglearn.tools.visualize_coefficients(coef.ravel()[mask],
                                     feature_names[mask], n_top_features=40)
```

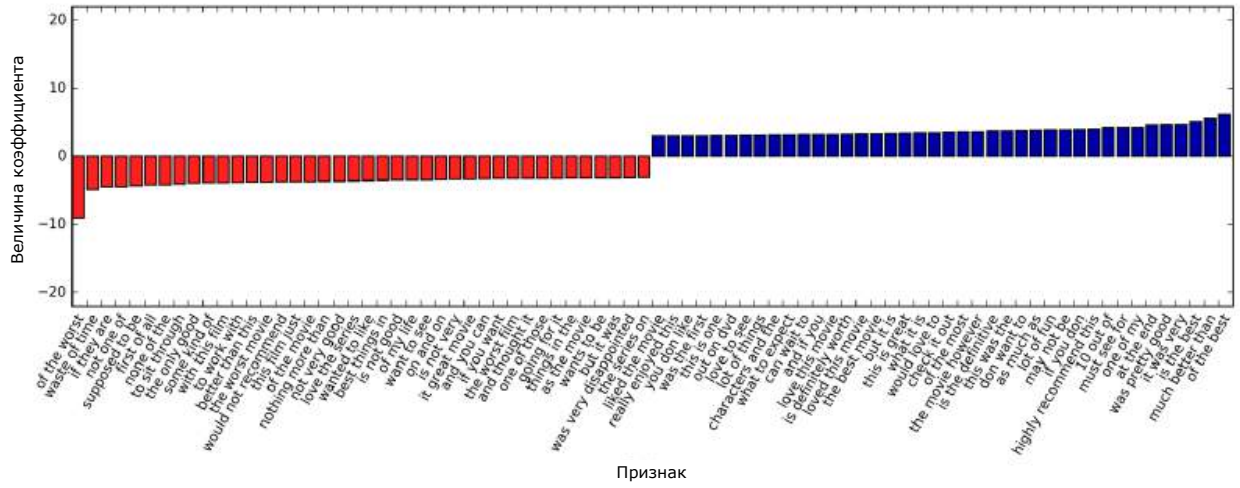

Рис. 7.5 Визуализация наиболее важных триграммных признаков модели модели

Продвинутая токенизация, стемминг и лемматизация

Как упоминалось ранее, выделение признаков в `CountVectorizer` и `TfidfVectorizer` является относительно простым процессом, однако вы можете применить гораздо более сложные методы. В более сложных задачах обработки текста часто возникает необходимость улучшить токенизацию, которая является первым этапом создания модели «мешка слов». Этот этап определяет, что представляет собой слово в плане извлечения признаков.

Ранее мы видели, что словарь часто содержит одновременно единственное и множественное число одинаковых по смыслу слов, например, "drawback" и "drawbacks", "drawer" и "drawers", "drawing" и

"drawings". При построении модели «мешка слов» необходимо учитывать близость слов "drawback" и "drawbacks" по смыслу, присутствие этих слов в виде отдельных признаков лишь увеличит переобучение вместо того, чтобы позволить модели в полной мере использовать обучающие данные. Аналогично, мы обнаружили, что словарь включает в себя такие слова, как "replace", "replaced", "replacement", "replaces" и "replacing", которые представляют собой разные глагольные формы и существительное, связанное с глаголом "replace". Как и в случае с единственным и множественным числом, обработка различных глагольных форм и взаимосвязанных слов как отдельных токенов является препятствием, не позволяющим добиться хорошей обобщающей способности модели.

Эту проблему можно решить, найдя для каждого слова его *основу* (*word stem*). Это подразумевает идентификацию или *объединение* (*conflating*) всех слов с одной и той же основой. Если этот процесс выполняется с помощью эвристик на основе правил (например, удаление общих суффиксов), его обычно называют *стеммингом* (*stemming*). Если вместо этого используется словарь с заранее заданными формами слов (явный процесс, контролируемый человеком) и учитывается роль слова в предложении (то есть принимаем во внимание, к какой части речи относится слово), то этот процесс называется *лемматизацией* (*lemmatization*), а стандартизированная форма слова называется *леммой* (*lemma*). Лемматизация и стемминг являются способами *нормализации* (*normalization*), которые пытаются извлечь определенную нормальную (то есть начальную) форму слова. Еще один интересный случай нормализации – это исправление орфографических ошибок, которое может быть полезно на практике, однако выходит за рамки данной книги.

Чтобы получить более полное представление о нормализации, давайте сравним стемминг (мы воспользуемся стеммингом Портера, широко используемым набором эвристик, в данном случае импортируем его из пакета `nltk`) с лемматизацией, реализованной в пакете `spacy`:[46]

```
In[36]:
import spacy
import nltk

# загружаем модели пакета spacy для английского языка
en_nlp = spacy.load('en')
# создаем экземпляр стеммера Портера из пакета nltk
stemmer = nltk.stem.PorterStemmer()

# задаем функцию, сравнивающую лемматизацию в spacy со стеммингом в nltk
def compare_normalization(doc):
```

[46] Для получения дополнительной информации по интерфейсу обратитесь к документации по nltk и spacy. В данном случае нас интересуют общие принципы работы. Если вы используете Anaconda, установите `spacy` с помощью команды `conda install -c spacy spacy=0.101.0`. Также не забудьте установить языковую модель `spacy` с помощью команды `python -m spacy.en.download`.

```
    # токенизируем документ в spacy
    doc_spacy = en_nlp(doc)
    # печатаем леммы, найденные с помощью spacy
    print("Лемматизация:")
    print([token.lemma_ for token in doc_spacy])
    # печатаем токены, найденные с помощью стеммера Портера
    print("Стемминг:")
    print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])
```

Мы сравним результаты лемматизации и использования стеммера Портера на простом примере:

```
In[37]:
compare_normalization(u"Our meeting today was worse than yesterday, "
                      "I'm scared of meeting the clients tomorrow.")

Out[37]:
Лемматизация:
['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be',
'scared', 'of', 'meet', 'the', 'client', 'tomorrow', '.']

Стемминг:
['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', "'m",
'scare', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
```

Стемминг всегда выполняет обрезку слова до его основы, поэтому `"was"` становится `"wa"`, в то время как лемматизация может извлечь правильную базовую форму глагола `"be"`. Аналогично, лемматизация может нормализовать `"worse"` к форме `"bad"`, тогда как в результате стемминга мы получим `"wors"`. Еще одно важное отличие состоит в том, что стемминг сокращает оба слова `"meeting"` до `"meet"`. При использовании лемматизации первое вхождение слова `"meeting"` будет распознано как существительное и оставлено в неизменном виде, тогда как второе вхождение слова будет распознано как глагол и сокращено до `"meet"`.

Несмотря на то что в библиотеке `scikit-learn` не реализован ни один из способов нормализации, `CountVectorizer` позволяет задать собственный токенизатор, который преобразует каждый документ в список токенов с помощью параметра `tokenizer`. Мы можем использовать лемматизацию из пакета `spacy`, чтобы создать функцию, которая примет в качестве аргумента строку и сгенерует список лемм:

```
In[38]:
# С технической точки зрения мы хотим применить токенизатор на основе
# регулярных выражений (regexp), который используется в CountVectorizer, а
# пакет spacy  использовать лишь для лемматизации. Для этого мы
# заменим en_nlp.tokenizer (токенизатор пакета spacy)
# токенизатором на основе регулярных выражений.
import re
# regexp, используемые в CountVectorizer
regexp = re.compile('(?u)\\b\\w\\w+\\b')

# загружаем языковую модель spacy и сохраняем старый токенизатор
en_nlp = spacy.load('en')
old_tokenizer = en_nlp.tokenizer
# заменяем токенизатор  старым на основе регулярных выражений
en_nlp.tokenizer = lambda string: old_tokenizer.tokens_from_list(
```

```
    regexp.findall(string))

# создаем пользовательский токенизатор с помощью конвейера обработки документов spacy
# (теперь используем наш собственный токенизатор)
def custom_tokenizer(document):
    doc_spacy = en_nlp(document, entity=False, parse=False)
    return [token.lemma_ for token in doc_spacy]

# задаем countvectorizer с пользовательским токенизатором
lemma_vect = CountVectorizer(tokenizer=custom_tokenizer, min_df=5)
```

Давайте преобразуем данные и проверим размер словаря:

```
In[39]:
# преобразуем text_train, используя CountVectorizer с лемматизацией
X_train_lemma = lemma_vect.fit_transform(text_train)
print("форма X_train_lemma: {}".format(X_train_lemma.shape))

# стандартный CountVectorizer для сравнения
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("форма X_train: {}".format(X_train.shape))

Out[39]:
форма X_train_lemma: (25000, 21596)
форма X_train: (25000, 27271)
```

Как видно из вывода, в результате лемматизации количество признаков с 27271 (когда использовалась стандартная обработка `CountVectorizer`) снизилось до 21596. Лемматизацию можно рассматривать как своего рода регуляризацию, так как она объединяет некоторые признаки. Таким образом, мы ожидаем, что лемматизация улучшает качество модели, когда набор данных невелик. Чтобы проиллюстрировать пользу от применения лемматизации, мы применим стратегию перекрестной проверки `StratifiedShuffleSplit`, задав лишь 1% данных для обучения, а остальные данные будем использовать для тестирования:

```
In[40]:
# строим модель решетчатого поиска, используя 1% данных в качестве обучающего набора
from sklearn.model_selection import StratifiedShuffleSplit

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
cv = StratifiedShuffleSplit(n_iter=5, test_size=0.99,
                            train_size=0.01, random_state=0)
grid = GridSearchCV(LogisticRegression(), param_grid, cv=cv)
# выполняем решетчатый поиск, используя данные, к которым был
# применен стандартный CountVectorizer
grid.fit(X_train, y_train)
print("Наилучшее значение перекрестной проверки "
      "(стандартный CountVectorizer): {:.3f}".format(grid.best_score_))
# выполняем решетчатый поиск, используя данные, к которым была
# применена лемматизация
grid.fit(X_train_lemma, y_train)
print("Наилучшее значение перекрестной проверки "
      "(лемматизация): {:.3f}".format(grid.best_score_))

Out[40]:
Наилучшее значение перекрестной проверки (стандартный CountVectorizer): 0.721
Наилучшее значение перекрестной проверки (лемматизация): 0.731
```

В данном случае лемматизация дала незначительное улучшение качества. Как и в случае с другими методами извлечения признаков, результат варьирует в зависимости от набора данных. Лемматизация и стемминг иногда могут помочь построить более качественные (или, по крайней мере, более компактные модели) модели, поэтому, если вы из последнего пытаетесь выжать качество модели, мы предлагаем вам воспользоваться этими методами.

Моделирование тем и кластеризация документов

Еще один метод, который часто применяется к текстовым данным – *моделирование тем* (*topic modeling*). Моделирование тем – это зонтичный термин, описывающий процедуру присвоения каждому документу одной или нескольких тем, которая осуществляется, как правило, без учителя. Хорошим примером моделирования тем является новостные данные, которые можно сгруппировать по таким темам, как «политика», «спорт», «финансы» и так далее. Если каждый документ может иметь только одну тему, то речь идет о задаче кластеризации документов, которая рассматривалась в главе 3. Если каждый документ может иметь несколько тем, эта задача относится к декомпозиционным методам, освещавшимся в главы 3. Каждая полученная компонента соответствует одной теме, а коэффициенты компонент, которые описывают документ, позволяют нам судить о том, насколько тесно данный документ связан с конкретной темой. Часто, когда люди говорят о моделирования тем, они имеют в виду конкретный декомпозиционный метод под названием *латентное размещение Дирихле* (*Latent Dirichlet Allocation*, *LDA*).[47]

Латентное размещение Дирихле

Говоря простым языком, модель LDA пытается найти группы слов (темы или топики), которые часто появляются вместе. LDA также подразумевает, что каждый документ можно интерпретировать как «смесь» из нескольких тем. Важно понимать, что для модели машинного обучения «тема» - это далеко не то же самое, что мы подразумеваем под «темой» в повседневной речи. В данном случае «тема» больше напоминает извлекаемые с помощью PCA или NMF компоненты (о котором мы говорили в главе 3), которые могут иметь или не иметь

[47] Существует еще одна модель машинного обучения, которую тоже часто сокращенно называют LDA: линейный дискриминантный анализ (Linear Discriminant Analysis), являющийся линейной моделью классификации. Это приводит к некоторой путанице. В этой книге под LDA подразумевается латентное размещение Дирихле (Latent Dirichlet Allocation).

смысловое значение. Даже если «тема», полученная с помощью LDA, и имеет смысловое значение, все равно она не тождественна «теме» в ее традиционном понимании. Вернемся к примеру с новостными статями. Представьте, у нас есть набор статей о спорте, политике и финансах, написанных двумя конкретными авторами. В политической статье мы могли бы ожидать появление слов типа «губернатор», «голос», «партия» и т.д., тогда как в спортивном статье мы могли бы встретить слова типа «сборная», «очко» и «сезон». Слова в каждой из этих групп с большой вероятностью встречаются вместе, в то время как вероятность совместного появления слов «сборная» и «губернатор» будет существенно меньше. Однако это не единственные группы слов, которые, по нашему мнению, встречаются вместе. Возможно, что два репортера предпочитают употреблять различные фразы или различные варианты слов. Возможно, один любит использовать слово «разграничивать», а другому нравится слово «поляризовать». Тогда «темами» уже будут «слова, часто используемые репортером A» и «слова, часто используемые репортером B», хотя они не являются темами в обычном смысле этого слова.

Давайте применим LDA к нашим киноотзывам, чтобы посмотреть, как этот метод работает на практике. Для моделей неконтролируемого обучения, применяющихся к текстовым документам, часто бывает полезно удалить наиболее часто употребляемые слова, поскольку в противном случае они в ходе анализа будут выбраны в качестве самых важных. Мы удалим слова, которые появляются по крайней мере в 15% документов, и ограничим модель «мешка слов» до 10000 слов, которые представляют собой наиболее часто встречающиеся слова, оставшиеся после удаления:

```
In[41]:
vect = CountVectorizer(max_features=10000, max_df=.15)
X = vect.fit_transform(text_train)
```

Мы построим модель, выделив 10 тем, что довольно мало для рассмотрения всех возможных вариантов.[48] Аналогично компонентам в NMF темы не имеют какого-то внутреннего порядка и изменение количества извлекаемых тем изменит содержательную суть всех тем. Мы воспользуемся методом обучения `"batch"`, который работает несколько медленнее по сравнению с методом `"online"`, установленным по умолчанию, но дает, как правило, лучшие результаты. Кроме того, мы увеличим значение параметра `max_iter`, что также позволить построить модель лучшего качества:

[48] На самом деле NMF и LDA решают во многом аналогичные задачи и мы могли бы также использовать NMF для извлечения тем.

```
In[42]:
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_topics=10, learning_method="batch",
                                max_iter=25, random_state=0)
# Мы строим модель и преобразуем данные в один этап
# Преобразование займет некоторое время,
# и мы можем сэкономить время, выполнив обе операции сразу
document_topics = lda.fit_transform(X)
```

Как и декомпозиционные методы, рассмотренные нами в главе 3, `LatentDirichletAllocation` имеет атрибут `components_`, который хранит информацию о том, насколько каждое слово важно для каждой выделенной темы. Атрибут `components_` имеет форму `(n_topics, n_words)`:

```
In[43]:
lda.components_.shape

Out[43]:
(10, 10000)
```

Чтобы лучше понять содержательный смысл каждой темы, мы проанализируем наиболее важные слова для каждой из тем. Функция `print_topics` позволяет представить эти признаки в удобном формате:

```
In[44]:
# Для каждой темы (строки в components_) сортируем признаки (по возрастанию)
# Инвертируем строки с помощью [:, ::-1], чтобы получить сортировку по убыванию
sorting = np.argsort(lda.components_, axis=1)[:, ::-1]
# Получаем имена признаков из векторизатора
feature_names = np.array(vect.get_feature_names())

In[45]:
# Выводим 10 тем:
mglearn.tools.print_topics(topics=range(10), feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=5, n_words=10)
```

Out[45]:

topic 0	topic 1	topic 2	topic 3	topic 4
between	war	funny	show	didn
young	world	worst	series	saw
family	us	comedy	episode	am
real	our	thing	tv	thought
performance	american	guy	episodes	years
beautiful	documentary	re	shows	book
work	history	stupid	season	watched
each	new	actually	new	now
both	own	nothing	television	dvd
director	point	want	years	got

topic 5	topic 6	topic 7	topic 8	topic 9
horror	kids	cast	performance	house
action	action	role	role	woman
effects	animation	john	john	gets
budget	game	version	actor	killer
nothing	fun	novel	oscar	girl
original	disney	both	cast	wife
director	children	director	plays	horror
minutes	10	played	jack	young
pretty	kid	performance	joe	goes
doesn	old	mr	performances	around

Если судить по словам, связанных с той или иной темой, похоже, что тема 1 соответствует историческим и военным фильмам, тема 2, возможно, связана с плохими комедиями, а тема 3, вероятно, соответствует ТВ-сериалам. Похоже, что тема 4 вобрала в себя некоторые очень распространенные слова, тогда как тема 6, по всей видимости, связана с детскими фильмами. Тема 8, по-видимому, содержит отзывы, связанные с кинонаградами. При `n_topics=10` темы должны быть очень широкими, чтобы вобрать в себя все многообразие киноотзывов.

Теперь мы построим еще одну модель, на этот раз выделив 100 тем. Увеличение `n_topics` в значительной мере усложняет анализ, но при этом повышает вероятность найти с помощью полученных тем интересные подмножества данных:

In[46]:
```
lda100 = LatentDirichletAllocation(n_topics=100, learning_method="batch",
                                   max_iter=25, random_state=0)
document_topics100 = lda100.fit_transform(X)
```

Вывод всех 100 тем было бы немного громоздким, поэтому мы выбрали лишь некоторые интересные и характерные темы:

In[47]:
```
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=7, n_words=20)
```

Out[47]:

topic 7	topic 16	topic 24	topic 25	topic 28
thriller	worst	german	car	beautiful
suspense	awful	hitler	gets	young
horror	boring	nazi	guy	old
atmosphere	horrible	midnight	around	romantic
mystery	stupid	joe	down	between
house	thing	germany	kill	romance
director	terrible	years	goes	wonderful
quite	script	history	killed	heart
bit	nothing	new	going	feel
de	worse	modesty	house	year
performances	waste	cowboy	away	each
dark	pretty	jewish	head	french
twist	minutes	past	take	sweet
hitchcock	didn	kirk	another	boy
tension	actors	young	getting	loved
interesting	actually	spanish	doesn	girl
mysterious	re	enterprise	now	relationship
murder	supposed	von	night	saw
ending	mean	nazis	right	both
creepy	want	spock	woman	simple

topic 36	topic 37	topic 41	topic 45	topic 51
performance	excellent	war	music	earth
role	highly	american	song	space
actor	amazing	world	songs	planet
cast	wonderful	soldiers	rock	superman
play	truly	military	band	alien
actors	superb	army	soundtrack	world
performances	actors	tarzan	singing	evil
played	brilliant	soldier	voice	humans
supporting	recommend	america	singer	aliens
director	quite	country	sing	human
oscar	performance	americans	musical	creatures
roles	performances	during	roll	miike
actress	perfect	men	fan	monsters
excellent	drama	us	metal	apes
screen	without	government	concert	clark
plays	beautiful	jungle	playing	burton
award	human	vietnam	hear	tim
work	moving	ii	fans	outer
playing	world	political	prince	men
gives	recommended	against	especially	moon

Похоже, что темы, извлеченные на этот раз, более конкретны, хотя многие из них трудно интерпретировать. Тема 7, по-видимому, соответствует фильмам ужасов и триллерам, темы 16 и 54 зафиксировали плохие отзывы, тогда как тема 63 преимущественным образом вобрала в себя положительные отзывы о комедиях. Если мы хотим сделать дополнительные выводы о выделенных темах, мы должны подтвердить свои предположения, выдвинутые нами, исходя из анализа наиболее важных слов по каждой теме. Для этого необходимо взглянуть на документы, которые были отнесены к этим темам. Например, тема 45,

похоже, связана с музыкой. Давайте посмотрим, какие отзывы отнесены к этой теме:

```
In[48]:
# сортируем документы по весу темы 45 "музыка"
music = np.argsort(document_topics100[:, 45])[::-1]
# печатаем пять документов, в которых данная тема является наиболее важной
for i in music[:10]:
    # выводим первые два предложения
    print(b".".join(text_train[i].split(b".")[:2]) + b".\n")
```

```
Out[48]:
b'I love this movie and never get tired of watching. The music in it is great.\n'
b"I enjoyed Still Crazy more than any film I have seen in years. A successful
  band from the 70's decide to give it another try.\n"
b'Hollywood Hotel was the last movie musical that Busby Berkeley directed for
  Warner Bros. His directing style had changed or evolved to the point that
  this film does not contain his signature overhead shots or huge production
  numbers with thousands of extras.\n'
b"What happens to washed up rock-n-roll stars in the late 1990's?
  They launch a comeback / reunion tour. At least, that's what the members of
  Strange Fruit, a (fictional) 70's stadium rock group do.\n"
b'As a big-time Prince fan of the last three to four years, I really can\'t
  believe I\'ve only just got round to watching "Purple Rain". The brand new
  2-disc anniversary Special Edition led me to buy it.\n'
b"This film is worth seeing alone for Jared Harris' outstanding portrayal
  of John Lennon. It doesn't matter that Harris doesn't exactly resemble
  Lennon; his mannerisms, expressions, posture, accent and attitude are
  pure Lennon.\n"
b"The funky, yet strictly second-tier British glam-rock band Strange Fruit
  breaks up at the end of the wild'n'wacky excess-ridden 70's. The individual
  band members go their separate ways and uncomfortably settle into lackluster
  middle age in the dull and uneventful 90's: morose keyboardist Stephen Rea
  winds up penniless and down on his luck, vain, neurotic, pretentious lead
  singer Bill Nighy tries (and fails) to pursue a floundering solo career,
  paranoid drummer Timothy Spall resides in obscurity on a remote farm so he
  can avoid paying a hefty back taxes debt, and surly bass player Jimmy Nail
  installs roofs for a living.\n"
b"I just finished reading a book on Anita Loos' work and the photo in TCM
  Magazine of MacDonald in her angel costume looked great (impressive wings),
  so I thought I'd watch this movie. I'd never heard of the film before, so I
  had no preconceived notions about it whatsoever.\n"
b'I love this movie!!! Purple Rain came out the year I was born and it has had
  my heart since I can remember. Prince is so tight in this movie.\n'
b"This movie is sort of a Carrie meets Heavy Metal. It's about a highschool
  guy who gets picked on alot and he totally gets revenge with the help of a
  Heavy Metal ghost.\n"
```

Как видно из вывода, данная тема охватывает широкий спектр музыкальных отзывов, посвященных мюзиклам, биографическим фильмам и трудно определимым жанрам, как в последнем отзыве. Еще один интересный способ исследовать темы – посмотреть, какой вес получает каждая тема в целом, просуммировав `document_topics` по всем отзывам. Каждой теме мы дадим названия, используя два самых часто встречаемых слова. Рис. 7.6 показывает вычисленные веса тем:

```
In[49]:
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]])]
# две столбиковые диаграммы:
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()
```

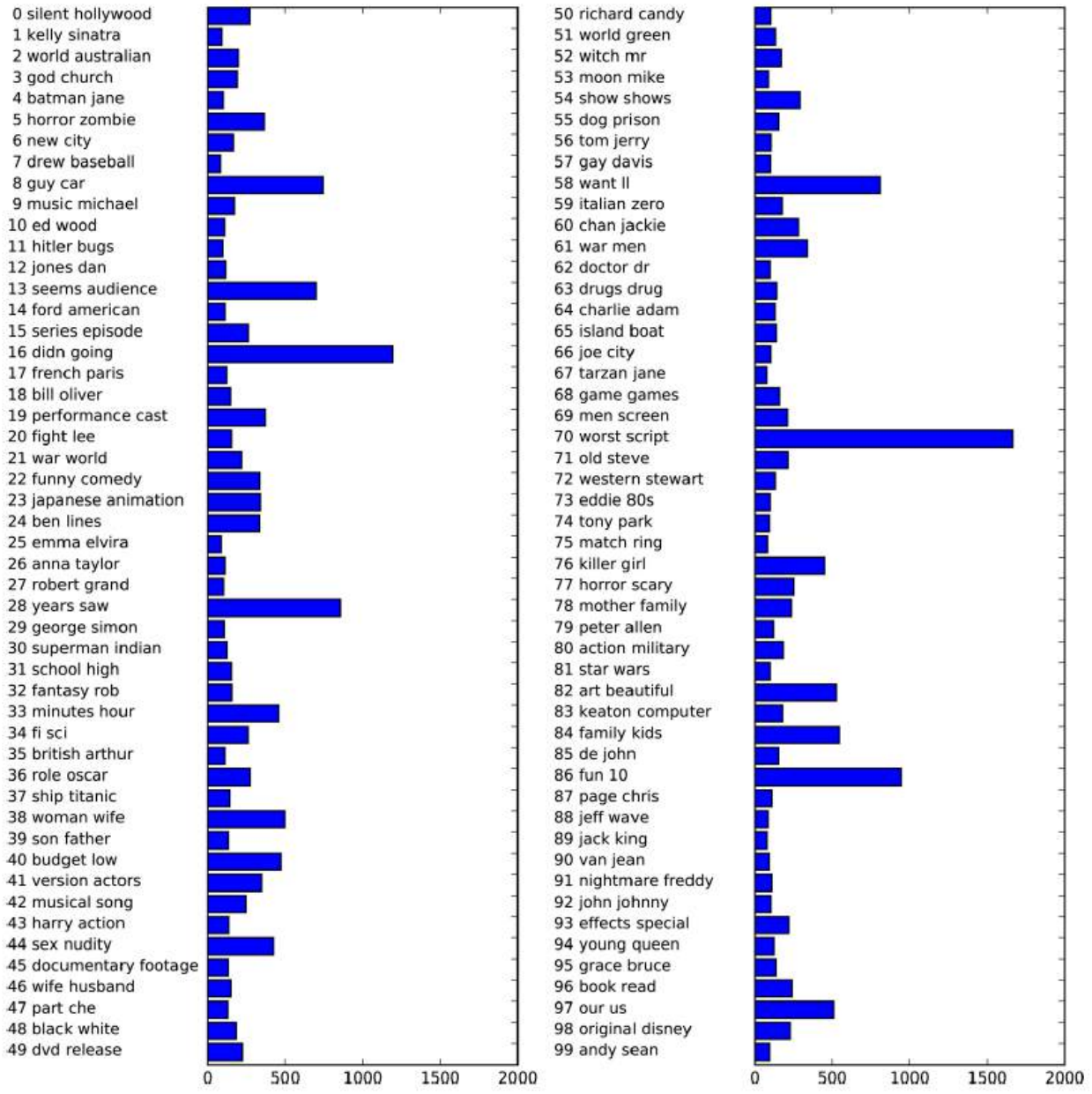

Рис. 7.6 Веса топиков, полученные с помощью LDA

Наиболее важными темами являются темы 70 и 16, которые, по-видимому, содержат отрицательные отзывы.

Похоже, что LDA в основном выделил два типа тем: темы, описывающие жанровые особенности фильмов и темы, обосновывающие ту или иную рейтинговую оценку. Кроме того, несколько тем не удалось отнести к конкретному типу. Это интересное открытие, так как большинство отзывов составлены из нескольких комментариев,

связанных с жанром описываемого фильма, и нескольких комментариев, в которых автор обосновывает или подчеркивает свою оценку.

Модели, получаемые с помощью LDA, представляют собой интересные методы, позволяющие интерпретировать огромные корпусы текстов, когда метки классов отсутствуют или когда они имеются, как в данном случае. Однако алгоритм LDA является рандомизированным и разные значения параметра `random_state` могут привести к совершенно различным результаты. Несмотря на то что выделение тем может быть полезным, любые выводы, которые можно сделать, исходя из результатов модели неконтролируемого обучения, нужно принимать с определенной долей сомнения и мы рекомендуем проверять выводы, анализируя документы, присвоенные определенной теме. Кроме того, темы, полученные с помощью метода `LDA.transform`, можно иногда использовать в качестве входного признака для машинного обучения с учителем.

Выводы и перспективы

В этой главе мы рассмотрели основы обработки текста, которая еще известна как *обработка естественного языка* (*natural language processing*, *NLP*), использовав в качестве примера классификацию киноотзывов. Рассмотренные здесь инструменты должны стать идеальной отправной точкой для обработки текстовых данных. В частности, при решении задач классификации текстов типа обнаружения спама и мошенничества или анализа настроений представление «мешок слов» является простым и эффективным методом. Как это часто бывает в машинном обучении, представление данных играет ключевую роль в прикладных задачах NLP, а исследование извлекаемых токенов и *n*-грамм позволяет глубже понять суть процесса моделирования. При решении прикладных задач обработки текста (как контролируемых, так и неконтролирумых) часто существует возможность заглянуть внутрь модели, как мы уже видели в этой главе. Вы должны в полной мере воспользоваться этой возможностью при использовании методов NLP на практике.

Естественный язык и обработка текста – это крупная научная область, и обсуждение деталей передовых методов выходит за рамки данной книги. Если вы хотите получить больше информации, мы рекомендуем книгу издательства O'Reilly *Natural Language Processing with Python*, написанную Стивеном Бердом, Эваном Кляйном и Эдвардом Лопером, в которой дан обзор NLP, а также рассказывается о питоновском пакете `nltk` для NLP. Еще одна интересная и более концептуальная книга – это стандартное справочное издание Introduction to Information Retrieval, написанная Кристофером Меннингом, Прабхакаром Рагхаваном и

Генрихом Шютце и посвященная основным алгоритмам информационного поиска, NLP и машинного обучения. Обе книги имеют онлайн-версии, которые можно получить бесплатно. Как мы уже говорили ранее, классы `CountVectorizer` и `TfidfVectorizer` позволяют реализовать только относительно простые методы обработки текста. Чтобы воспользоваться более продвинутыми методами обработки текста, мы рекомендуем питоновские пакеты `spacy` (относительно новый, но очень эффективный и хорошо разработанный пакет), `nltk` (очень хорошо отлаженная библиотека, хотя и несколько устаревшая) и `gensim` (пакет, предназначенный для NLP, с упором на моделирование тем).

В последние годы в области обработки текста появилось несколько очень интересных направлений, которые выходят за рамки этой книги и относятся к нейронным сетям. Первое направление – это использование непрерывных векторных представлений (также известных как векторы слов или распределенные представления слов), которые реализованы в библиотеке `word2vec`. Оригинальная статья *Distributed Representations of Words and Phrases and Their Compositionality*, написанная Томасом Миколовым с соавторами, является прекрасным введением в тему. Пакеты `spacy` и `gensim` обеспечивают функциональные возможности для реализации методов, рассмотренных в данной статье и ее продолжениях.

Еще одно направление в NLP, которое стало популярным в последние годы – это применение *рекуррентных нейронных сетей* (*recurrent neural networks*, *RNN*) для обработки текста. Рекуррентные нейронные сети – это особенно мощный вид нейронной сети, который в качестве вывода может снова генерировать текст в отличие от моделей классификации, которые могут лишь назначать метки классов. Способность генерировать текст в качестве вывода позволяет с успехом использовать рекуррентные нейронные сети для автоматического перевода и реферирования. Познакомиться с этой темой можно в исключительно технической статье Sequence to Sequence Learning with Neural Networks Ильи Сускевера, Ориоля Виньялса и Куока Ле. Более практичное пособие по использованию фреймворка `tensorflow` можно найти на веб-сайте TensorFlow.