Обработка текста на естественном языке

Наумов Д.А., доц. каф. КТ

Экспертные системы и искусственный интеллект, 2019

Содержание лекции

1. Строковые данные
2. Обработка естественного языка
3. Мешок слов
4. Построение модели
5. Стоп-слова
6. Масштабирование данных при помощи tf-idf
7. n-граммы

Введение

Типы признаков, которые могут представлять свойства данных:

- непрерывные признаки, описывающие количество;
- категориальные признаки, которые являются элементами фиксированного списка;
- текст.

Пример:

- классифицировать сообщения электронной почты на спам и действительно нужные письма;
- узнать мнение какого-то политика об иммиграции из текстов его выступлений или твитов;
- выяснить, является ли сообщение жалобой или запросом, проанализировав тему и содержание сообщения.

Введение

Строковые данные:

- 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.

Неструктурированне строки

ответы, записанные в текстовом поле, которые по смыслу можно сгруппировать в категории.

Введение

Структурированные строки

строковые значения, введенные вручную, не соответствующие фиксированным категориям, но при этом все же имеющие некоторую базовую структуру, например, адреса, названия мест, имена и фамилии людей, даты, номера телефонов и другие идентификаторы.

Текстовые данные

оторые состоят из фраз или предложений.

Пример:

- твиты, логи чата;
- отзывы о гостинице;
- собрание сочинений Шекспира;
- содержание Википедии.

NLP

Обработка естественного языка (Natural Language Processing, NLP) - важная частью современных систем.

Цель NLP

разработка алгоритмов, которые позволяли бы компьютерам распознавать свободный текст и понимать живую речь.

Области применения:

- поисковые системы;
- речевые интерфейсы;
- процессоры документов.

Применения подходов на основе машинного обучения:

- сбор огромных массивов текста (для понимания контекста);
- обучение алгоритма для выполнения различных задач (категоризация текста, сентимент-анализ, тематическое моделирование).

Пример: анализ тональности киноотзывов

Воспользуемся набором данных, который содержит киноотзывы, оставленные на сайте IMDB.

- собранны исследователем Стэнфордского университета Эндрю Маасом;
- набор данных доступен по ссылке http://ai.stanford.edu/ amaas/data/sentiment/;
- набор содержит тексты отзывов, а также метки, которые указывают тональность отзыва («положительный» или «отрицательный»).

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

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

Пример: анализ тональности киноотзывов

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. 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
```

Пример: анализ тональности киноотзывов

- text_train - список длиной 25000, в котором каждый элемент представляет собой строку, содержащую отзыв.
- отзыв содержит разрывы строк HTML (br). 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]
```

Пример: анализ тональности киноотзывов

Набор данных был собран таким образом, чтобы положительный и отрицательный классы были сбалансированы.

```
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]
```

Пример: анализ тональности киноотзывов

Постановка задачи

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.
- необходимо преобразовать строковое представление текста в числовое представление, к которому можно применить алгоритмы машинного обучения.

С точки зрения анализа текста набор данных часто называют **корпусом** (corpus) и каждая точка данных, представленная в виде отдельного текста, называется **документом** (document).

Мешок слов, bag-ofwords

представление, в котором удалена структура исходного текста (главы, параграфы, предложения, форматирование) и подсчитана частота встречаемости каждого слова в каждом документе корпуса.

Три этапа получения мешка слов:

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.

Обработка «мешок слов»

Пример: «мешок слов» для синтетического набора данных

Модель «мешка слов» реализована в классе CountVectorizer, который выполняет соответствующее преобразование.

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

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 «мешок слов», 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>
```

Пример: «мешок слов» для синтетического набора данных

Представление «мешок слов» записывается в разреженной матрице SciPy, которая хранит только ненулевые элементы. Чтобы взглянуть на фактическое содержимое разреженной матрицы, мы можем преобразовать ее в «плотный» массив NumPy:

```
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]]
```

Пример: «мешок слов» для киноотзывов

Обработаем обучающие и тестовые данные, сформированные на основе отзывов IMDb, в виде списков строк (text_train и text_test):

```
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 в разреженной матрицы SciPy.

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',
```

Проблемы: числа, выделение значимой информации, обработка форм слов.

Как правило, для подобных высокоразмерных разреженных данных лучше всего работают линейные модели типа LogisticRegression. Измерим качество модели, построив классификатор:

```
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
```

Среднее значение правильности перекрестной проверки - 88%, что указывает на приемлемое качество модели для задачи сбалансированной бинарной классификации.

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]}

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

Out[15]:

```
Наилучшее значение перекрестной проверки: 0.89
Наилучшие параметры: {'C': 0.1}
```

Будем использовать только те токены, которые встречаются по крайней мере в двух документах (или по крайней мере в пяти документах и т.д.), задавания параметр min_df:

```
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:

```
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. 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.

Стоп-слова

Мы можем избавиться от неинформативных слов:

- использование списка стоп-слов (на основе соответствующего языка);
- 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']
```

Стоп-слова

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:

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 стопслов (но не все) встретились в корпусе документов.

Стоп-слова

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
```

- качество модели чуть снизилось;
- исключили 305 признаков из более чем 27000;
- использование списка стоп-слов в данном случае бесполезно;
- фиксированные списки могут быть полезны при работе с небольшими наборами данных;
- можно попробовать другой подход, задав опцию max_df для CountVectorizer.

Масштабирование данных при помощи tf-idf

Частота термина-обратная частота документа (term frequency-inverse document frequency, tf-idf)

идея заключается в том, чтобы присвоить большой вес термину, который часто встречается в конкретном документе, но при этом редко встречается в остальных документах корпуса.

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

- N – количество документов в обучающем наборе,
- Nw – количество документов обучающего набора, в которых встретилось слово w,
- tf (частота термина) – это частота встречаемости термина в запрашиваемом документе d (документе, который вы хотите преобразовать).

Масштабирование данных при помощи tf-idf

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;
- TfidfVectorizer, который принимает на вход текстовые данные и выполняет как выделение признаков «мешок слов», так и преобразование tf-idf.

Воспользуемся конвейером, чтобы убедиться в достоверности результатов нашего решетчатого поиска:

```
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
```

Масштабирование данных при помощи tf-idf

Мы можем выяснить, какие слова в результате преобразования tf-idf стали наиболее важными.

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:
```

Масштабирование данных при помощи tf-idf

- признаки с низкими значениями tf-idf – это признаки, которые либо встречаются во многих документах, либо используются редко и только в очень длинных документах.
- 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 онивстречаются в данных отзывах очень часто ("smallville"и "doodlebops").

Мы можем найти слова, которые имеют низкое значение обратной частоты документа (атрибут 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]:

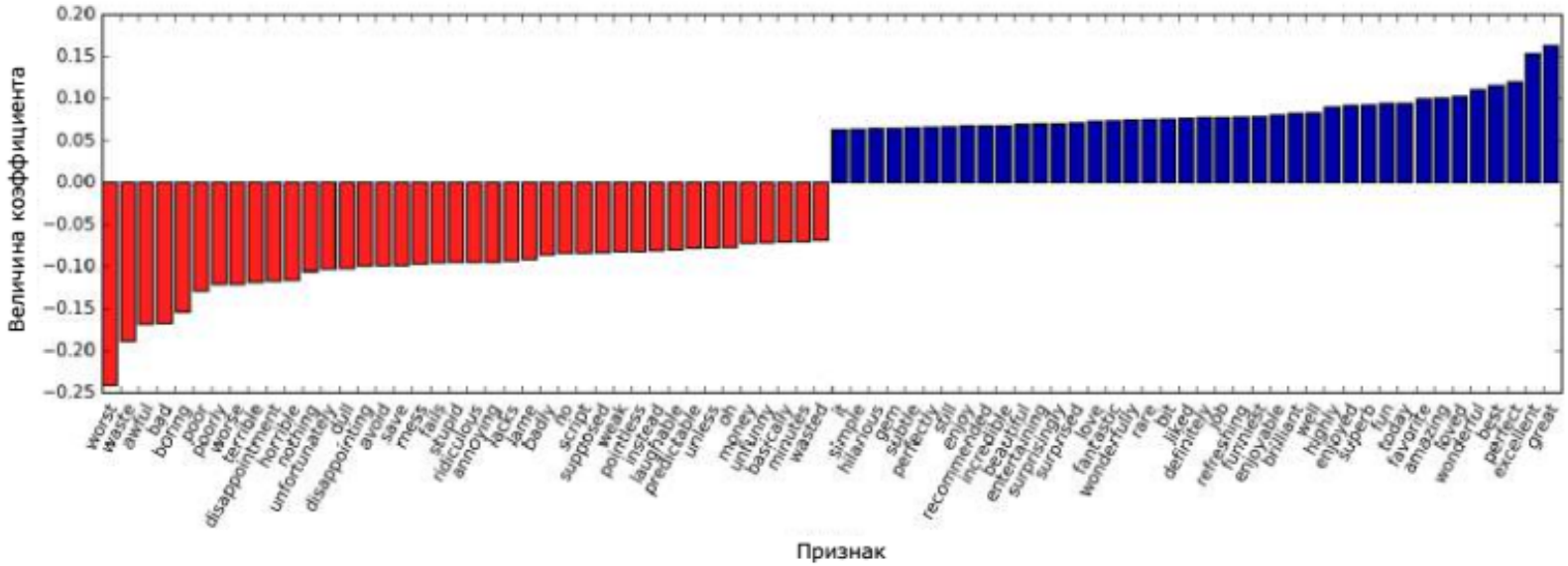
```
Признаки с наименьшими признаками idf:
['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with'
 'was' 'as' 'on' 'movie' 'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all'
 'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they' 'there' 'if' 'his' 'out'
 'just' 'about' 'he' 'or' 'has' 'what' 'some' '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']
```

- слова с низкими значениями idf стали английские стоп-слова типа "the"и "no";

Исследование коэффициентов модели

Рассмотрим коэффициенты, получившие максимальные значения и их слова.

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



Следующая гистограмма показывает 25 наибольших и 25 наименьших коэффициентов модели логистической регрессии, каждый столбик соответствует величине коэффициента.

Модель «мешка слов» для n-грамм

Один из главных недостатков представления «мешок слов»: полное игнорировании порядка слов. Две строки

- «it's bad, not good at all»
- «it's good, not bad at all»

будут иметь одинаковое представление, хотя противоположны по смыслу.

n-грамма

последовательность токенов

- пары токенов - биграммами (bigrams);
- тройки токенов - триграммы (trigrams);
- последовательности из n токенов - n-граммы (n-grams).

Модель «мешка слов» для n-грамм

- 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 nзначений для различных извлекаемых n-грамм.

```
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']
```

Модель «мешка слов» для n-грамм

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']
```

Модель «мешка слов» для n-грамм

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']
```

Модель «мешка слов» для n-грамм

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_word

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]]
```

Модель «мешка слов» для n-грамм

- в большинстве прикладных задач минимальное количество токенов должно быть равно единице;
- добавление биграмм помогает в большинстве случаев;
- включение в анализ более длинных последовательностей, тоже, вероятно, поможет, но это вызовет взрывной рост количества признаков.

```
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']
```

Модель «мешка слов» для n-грамм для киноотзывов

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.

Модель «мешка слов» для n-грамм для киноотзывов

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.

Модель «мешка слов» для n-грамм для киноотзывов

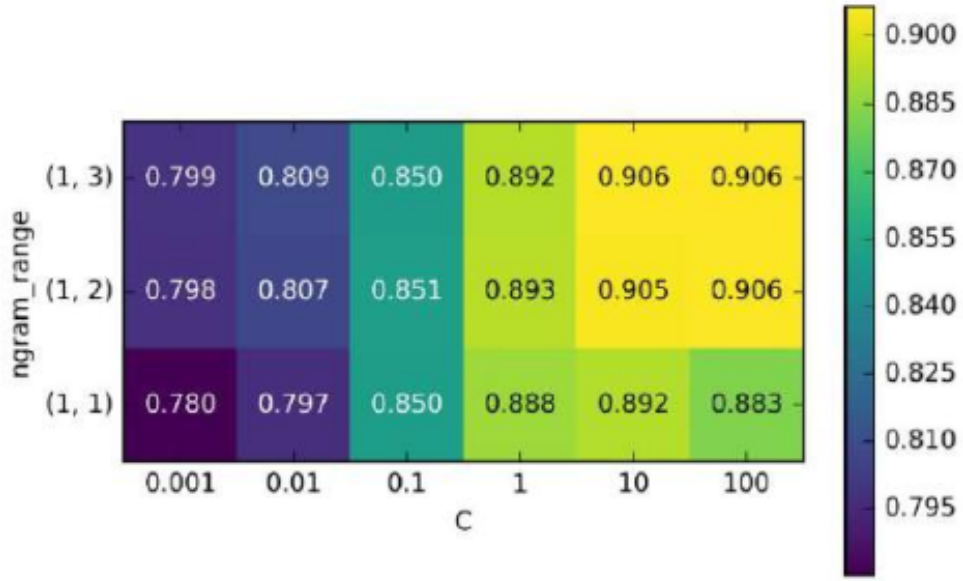
Мы можем представить правильность перекрестной проверки в виде функции параметров ngram_range и C, использовав теплокарту:

```
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)
```

На теплокарте видно, что использование биграмм довольно значительно увеличивает качество модели, тогда как добавление триграмм дает очень небольшое преимущество с точки зрения правильности.

Модель «мешка слов» для n-грамм для киноотзывов

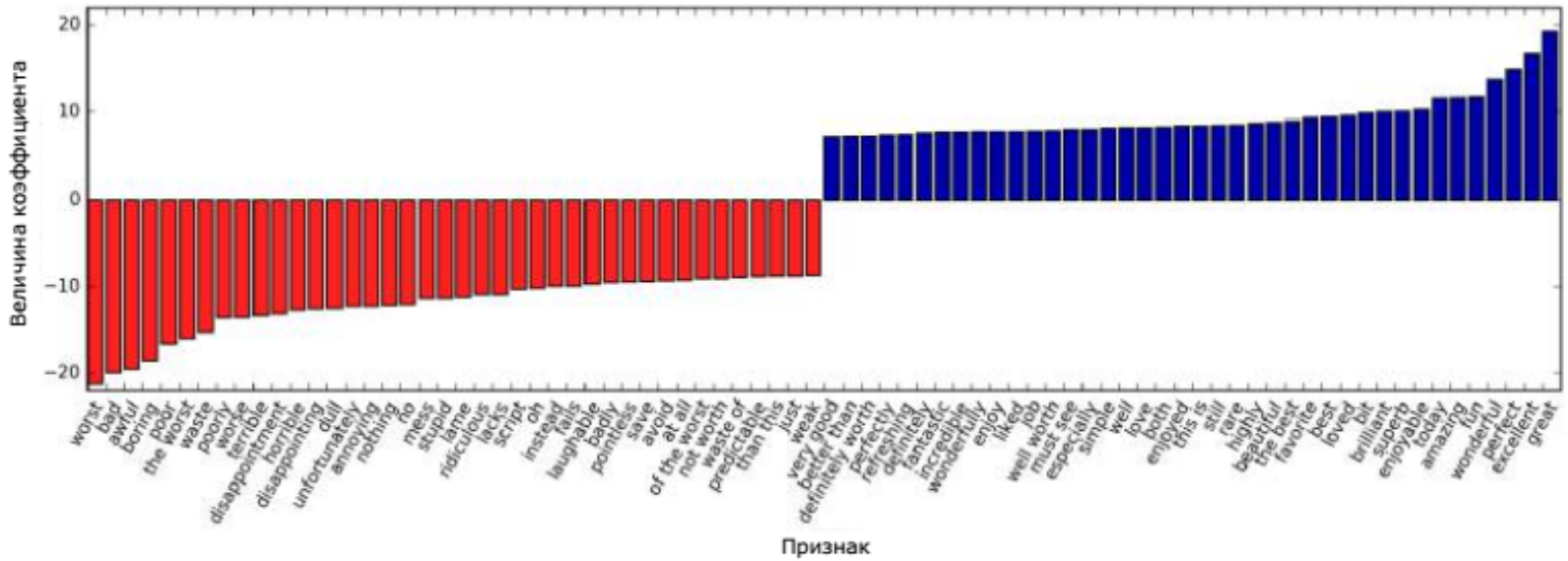
Визуализируем наиболее важные коэффициенты наилучшей модели, которая включает юниграммы, биграммы и триграммы:



Модель «мешка слов» для n-грамм для киноотзывов

Визуализируем наиболее важные коэффициенты наилучшей модели, которая включает юниграммы, биграммы и триграммы:

```
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)
```

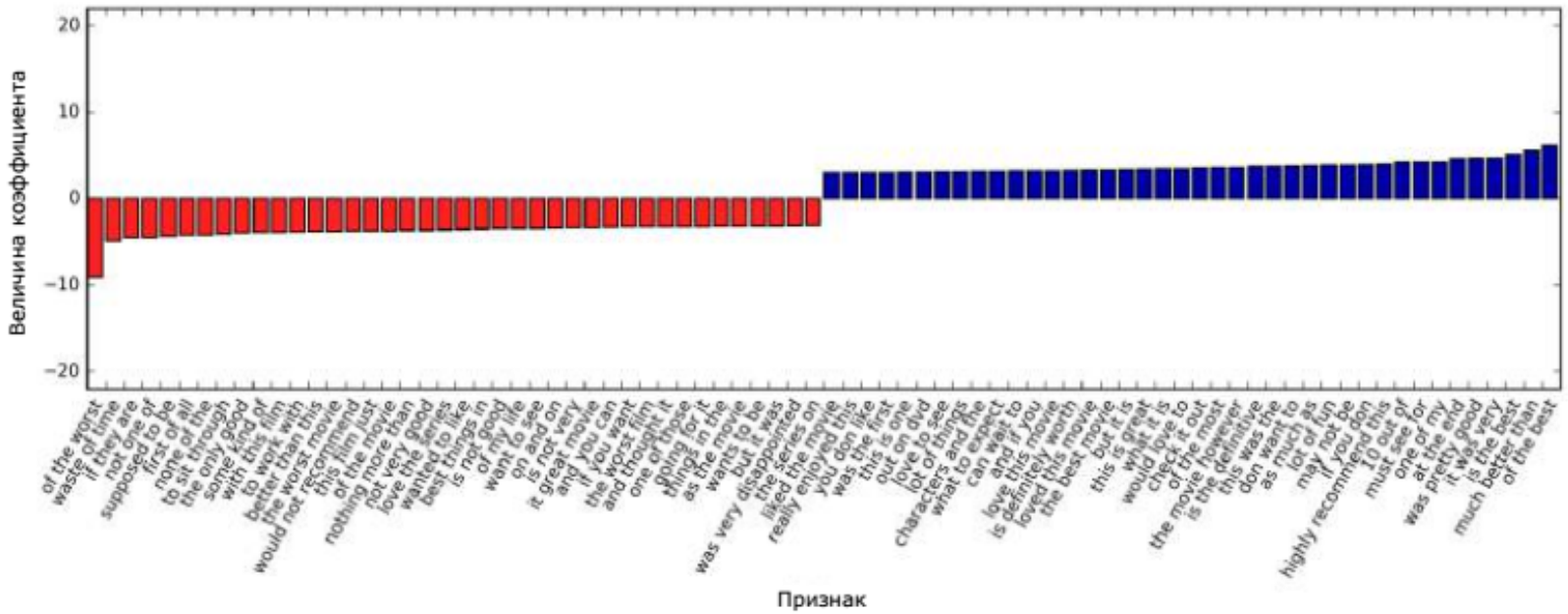


У нас появились весьма интересные признаки со словом «worth», которые отсутствовали в юниграммной модели: "not worth" признак

Модель «мешка слов» для n-грамм для киноотзывов

Визуализируем только триграммы, чтобы лучше понять, какие признаки являются полезными.

```
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)
```



Влияние триграммных признаков по сравнению с важностью