# **ЛБ 11 Машинное обучение без учителя**

# 

# Неконтролируемые преобразования (unsupervised transformations) –

это алгоритмы, создающие новое представление данных, которое в отличие от исходного представления человеку или алгоритму машинного обучения будет обработать легче. Общераспространенное применение неконтролируемых преобразований – сокращение размерности. Мы берем высокоразмерное представление данных, состоящее из множества признаков, и находим новый способ представления этих данных, обобщая основные характеристики и получая меньшее количество признаков. Общераспространенное применение сокращения размерности

– получение двумерного пространства в целях визуализации.

Еще одно применение неконтролируемых преобразований – поиск компонент, из которых «состоят» данные. Примером такого преобразования является выделение тем из коллекций текстовых документов. Здесь задача состоит в том, чтобы найти неизвестные темы, обсуждаемые в коллекции документов, а также выяснить, какие темы встречаются в каждом документе. Это может быть полезно для отслеживания в социальных сетях обсуждений таких тем, как выборы, контроль огнестрельного оружия или жизнь поп-звезд.

С другой стороны, *алгоритмы кластеризации* (*clustering algorithms*)

разбивают данные на отдельные группы схожих между собой элементов. Рассмотрим пример загрузки фотографий в социальной сети. Часто вы формируете запросы типа «покажите мне все фотографии, на которых изображен Иван Петров». Для выполнения подобных запросов, администрация сайта, возможно, захочет сгруппировать фотографии, на которых изображен один и тот же человек. Однако при этом неизвестно,

на каких загружаемых фотографиях кто показан, и неизвестно, какое количество различных пользователей присутствует на ваших фотографиях. Разумный подход заключался бы в том, чтобы извлечь все лица и разделить их на группы лиц, которые схожи между собой. Будем надеяться, что они соответствуют одному и тому же человеку и изображения в сгруппированном виде будут предъявлены вам.



Главная проблема машинного обучения без учителя – оценка полезности информации, извлеченной алгоритмом. Алгоритмы машинного обучения без учителя, как правило, применяются к данным, которые не содержат никаких меток, таким образом, мы не знаем, каким должен быть правильный ответ. Поэтому очень трудно судить о качестве работы модели. Например, наш гипотетический алгоритм кластеризации мог бы сгруппировать вместе все фотографии лиц в профиль и все фотографии лиц в анфас. Перед нами, несомненно, один из способов разбить коллекцию фотографий лиц на группы, но это совсем не то, что нам нужно. Тем не менее у нас нет никакой возможности «рассказать» алгоритму, что мы ищем, и часто единственный способ оценить результат работы алгоритма машинного обучения без учителя – ручная проверка этого результата.

Как следствие, алгоритмы машинного обучения без учителя часто используются в разведочных целях, когда специалист хочет лучше изучить сами данные. Еще одно общераспространенное применение алгоритмов машинного обучения без учителя заключается в том, что они служат этапом предварительной обработки данных для алгоритмов машинного обучения с учителем. Изучение нового представления данных иногда может повысить правильность алгоритмов машинного обучения с учителем или может привести к снижению времени вычислений и потребления объема памяти.

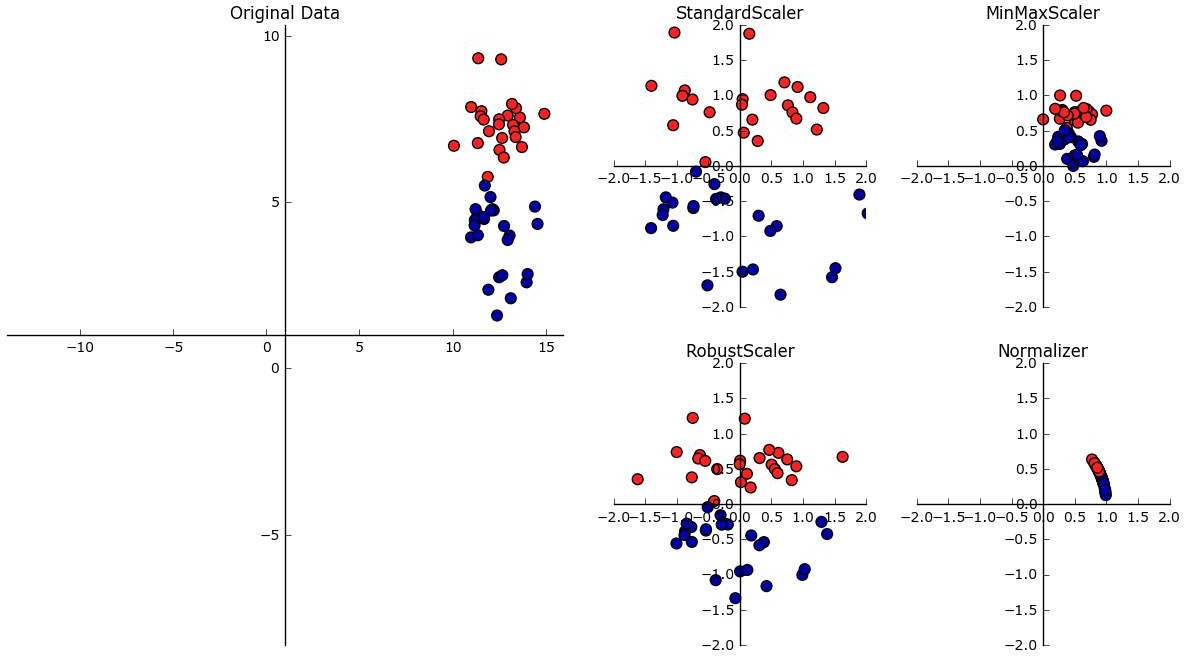
Прежде чем начать знакомство с «реальными» алгоритмами машинного обучения без учителя, мы кратко рассмотрим некоторые простые методы предварительной обработки данных, которые часто могут пригодиться. Хотя предварительная обработка данных и масштабирование часто применяются вместе с алгоритмами контролируемого обучения, методы масштабирования не используют учителя, что делает их методами неконтролируемого обучения.



В предыдущей главе мы видели, что некоторые алгоритмы, например, нейронные сети и SVM, очень чувствительны к масштабированию данных. Поэтому обычной практикой является преобразование признаков с тем, чтобы итоговое представление данных было более подходящим для использования вышеупомянутых алгоритмов. Часто достаточно простого масштабирования признаков и корректировки данных. Программный код (рис. 3.1) показывает простой пример:

In[2]:

mglearn.plots.plot\_scaling()



**Рис. 3.1** Различные способы масштабирования и предварительной обработки данных



Первый график на рис. 3.1 соответствует синтетическому двуклассовому набору данных с двумя признаками. Первый признак (ось *x*) принимает значения в диапазоне от 10 до 15. Второй признак (ось *y*) принимает значения примерно в диапазоне от 1 до 9.

Следующие четыре графика показывают четыре различных способа преобразования данных, которые дают более стандартные диапазоны значений. Применение StandardScaler в scikit-learn гарантирует, что для каждого признака среднее будет равно 0, а дисперсия будет равна 1,

в результате чего все признаки будут иметь один и тот же масштаб. Однако это масштабирование не гарантирует получение каких-то конкретных минимальных и максимальных значений признаков.

RobustScaler аналогичен StandardScaler в том плане, что в результате его применения признаки будут иметь один и тот же масштаб. Однако RobustScaler вместо среднего и дисперсии использует медиану и квартили 23 . Это позволяет RobustScaler игнорировать точки данных, которые сильно отличаются от остальных (например, ошибки измерений). Эти странные точки данных еще называются *выбросами* (*outliers*) и могут стать проблемой для остальных методов масштабирования.

С другой стороны, MinMaxScaler сдвигает данные таким образом, что все признаки находились строго в диапазоне от 0 до 1. Для двумерного набора данных это означает, что все данные помещаются в прямоугольник, образованный осью *х* с диапазоном значений от 0 и 1 и осью *у* с диапазоном значений от 0 и 1.

И, наконец, Normalizer осуществляет совершенно иной вид масштабирования. Он масштабирует каждую точку данных таким образом, чтобы вектор признаков имел евклидову длину 1. Другими словами, он проецирует точку данных на окружность с радиусом 1 (или сферу в случае большого числа измерений). Вектор умножается на инверсию своей длины. Подобная нормализация используется тогда, когда важным является направление (но не длина) вектора признаков.



Теперь, когда мы увидели, что делают различные виды преобразований, давайте применим их, воспользовавшись scikit-learn. Мы будем использовать набор данных cancer, известный нам по главе 2. Методы предварительной обработки обычно применяются перед использованием алгоритма машинного обучения с учителем. Допустим, в качестве примера нам нужно применить ядерный SVM (SVC) к набору данных cancer и использовать MinMaxScaler для предварительной обработки данных. Мы начнем с того, что загрузим наш набор данных и разобьем его на тренировочный и тестовый наборы (обучающий и тестовый наборы нам нужны для оценки качества модели, которую мы построим с

помощью алгоритма контролируемого обучения после предварительной обработки):

In[3]:

from sklearn.datasets import load\_breast\_cancer from sklearn.model\_selection import train\_test\_split cancer = load\_breast\_cancer()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(cancer.data, cancer.target,

random\_state=1)

23 Медиана множества чисел – это такое число *x*, при котором половина значений множества меньше *x*, а другая половина значений больше *x*. Нижний квартиль – это число *х*, ниже которого находится четверть значений, а верхний квартиль – это число *х*, выше которого находится четверть значений.

print(X\_train.shape) print(X\_test.shape)

Out[3]:

(426, 30)

(143, 30)

Напомним, что набор содержит 569 точек данных, каждая из которых представлена 30 признаками. Мы разбиваем набор данных на 426 примеров в обучающей выборке и 143 примера в тестовой выборке.

Как и в случае с моделями контролируемого обучения, построенными ранее, мы сначала импортируем класс, который осуществляет предварительную обработку, а затем создаем его экземпляр:

In[4]:

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

Затем с помощью метода fit мы подгоняем scaler на обучающих данных. Для MinMaxScaler метод fit вычисляет минимальное и максимальное значения каждого признака на обучающем наборе. В отличие от классификаторов и регрессоров, описанных в главе 2, при вызове метода fit scaler работает с данными (X\_train), а ответы (y\_train) не используются:

In[5]:

scaler.fit(X\_train)

Out[5]:

MinMaxScaler(copy=True, feature\_range=(0, 1))

Чтобы применить преобразование, которое мы только что подогнали, то есть фактически *отмасштабировать* (*scale*) обучающие данные, мы воспользуемся методом transform. Метод transform используется в scikit-learn, когда модель возвращает новое представление данных:

In[6]:

*# преобразовываем данные*

X\_train\_scaled = scaler.transform(X\_train)

*# печатаем значения признаков в обучающем наборе до и после масштабирования* print("форма преобразованного массива: {}".format(X\_train\_scaled.shape)) print("min значение признака до масштабирования:\n {}".format(X\_train.min(axis=0))) print("max значение признака до масштабирования:\n {}".format(X\_train.max(axis=0))) print("min значение признака после масштабирования:\n {}".format(

X\_train\_scaled.min(axis=0)))

print("max значение признака после масштабирования:\n {}".format( X\_train\_scaled.max(axis=0)))

Out[6]:

форма преобразованного массива: (426, 30) min значение признака до масштабирования:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [ 6.98 9.71 43.79 143.50 0.05 | 0.02 | 0. | 0. | 0.11 |
| 0.05 0.12 0.36 0.76 6.80 | 0. | 0. | 0. | 0. |
| 0.01 0. 7.93 12.02 50.41 | 185.20 | 0.07 | 0.03 | 0. |
| 0. 0.16 0.06] |  |  |  |  |

max значение признака до масштабирования:

|  |  |  |
| --- | --- | --- |
| [ 28.11 39.28 188.5 | 2501.0 | 0.16 0.29 0.43 0.2 |
| 0.300 0.100 2.87 | 4.88 | 21.98 542.20 0.03 0.14 |
| 0.400 0.050 0.06 | 0.03 | 36.04 49.54 251.20 4254.00 |
| 0.220 0.940 1.17 | 0.29 | 0.58 0.15] |

min значение признака после масштабирования:

[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

max значение признака после масштабирования:

[ 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]

Преобразованные данные имеют такую же форму, что и исходные данные – признаки просто смещены и масштабированы. Видно, что теперь все признаки принимают значения в диапазоне от 0 до 1, как нам и требовалось.

Чтобы применить SVM к масштабированным данным, мы должны преобразовать еще тестовый набор. Это снова делается с помощью вызова метода transform, на этот раз для X\_test:

In[7]:

*# преобразовываем тестовые данные*

X\_test\_scaled = scaler.transform(X\_test)

*# печатаем значения признаков в тестовом наборе после масштабирования*

print("min значение признака после масштабирования:\n{}".format(X\_test\_scaled.min(axis=0))) print("max значение признака после масштабирования:\n{}".format(X\_test\_scaled.max(axis=0)))

|  |  |
| --- | --- |
| Out[7]:  min значение признака после масштабирования: |  |
| [ 0.034 0.023 0.031 0.011 0.141 0.044 0. 0. | 0.154 -0.006 |
| -0.001 0.006 0.004 0.001 0.039 0.011 0. 0. | -0.032 0.007 |
| 0.027 0.058 0.02 0.009 0.109 0.026 0. 0. | -0. -0.002] |
| max значение признака после масштабирования: |  |
| [ 0.958 0.815 0.956 0.894 0.811 1.22 0.88 | 0.933 0.932 1.037 |
| 0.427 0.498 0.441 0.284 0.487 0.739 0.767 | 0.629 1.337 0.391 |
| 0.896 0.793 0.849 0.745 0.915 1.132 1.07 | 0.924 1.205 1.631] |

Возможно, полученные результаты несколько удивят вас: после масштабирования минимальные и максимальные значения признаков в тестовом наборе не равны 0 и 1. Некоторые признаки даже выходят за пределами диапазона 0-1! Объяснить это можно тем, что MinMaxScaler (и все остальные типы масштабирования) всегда применяют одинаковое преобразование к обучающему и тестовому наборам. Это означает, что метод transform всегда вычитает минимальное значение, вычисленное для обучающего набора, и делит на ширину диапазона, вычисленную

также для обучающего набора. Минимальное значение и ширина диапазона для обучающего набора могут отличаться от минимального значения и ширины диапазона для тестового набора.



Чтобы модель контролируемого обучения работала на тестовом наборе, важно преобразовать обучающий и тестовый наборы одинаковым

образом. Следующий пример (рис. 3.2) показывает, что произошло бы, если бы мы использовали минимальное значение и ширину диапазона, отдельно вычисленные для тестового набора:

In[8]:

from sklearn.datasets import make\_blobs

*# создаем синтетические данные*

X, \_ = make\_blobs(n\_samples=50, centers=5, random\_state=4, cluster\_std=2)

*# разбиваем их на обучающий и тестовый наборы*

X\_train, X\_test = train\_test\_split(X, random\_state=5, test\_size=.1)

*# размещаем на графике обучающий и тестовый наборы* fig, axes = plt.subplots(1, 3, figsize=(13, 4)) axes[0].scatter(X\_train[:, 0], X\_train[:, 1],

c=mglearn.cm2(0), label="Обучающий набор", s=60) axes[0].scatter(X\_test[:, 0], X\_test[:, 1], marker='^',

c=mglearn.cm2(1), label="Тестовый набор", s=60) axes[0].legend(loc='upper left')

axes[0].set\_title("Исходные данные")

*# масштабируем данные с помощью MinMaxScaler*

scaler = MinMaxScaler() scaler.fit(X\_train)

X\_train\_scaled = scaler.transform(X\_train) X\_test\_scaled = scaler.transform(X\_test)

*# визуализируем правильно масштабированные данные*

axes[1].scatter(X\_train\_scaled[:, 0], X\_train\_scaled[:, 1], c=mglearn.cm2(0), label="Обучающий набор", s=60)

axes[1].scatter(X\_test\_scaled[:, 0], X\_test\_scaled[:, 1], marker='^', c=mglearn.cm2(1), label="Тестовый набор", s=60)

axes[1].set\_title("Масштабированные данные")

*# масштабируем тестовый набор отдельно*

*# чтобы в тестовом наборе min значение каждого признака было равно 0 # а max значение каждого признака равнялось 1*

*# НЕ ДЕЛАЙТЕ ТАК! Только в ознакомительных целях.* test\_scaler = MinMaxScaler() test\_scaler.fit(X\_test)

X\_test\_scaled\_badly = test\_scaler.transform(X\_test)

*# визуализируем неправильно масштабированные данные*

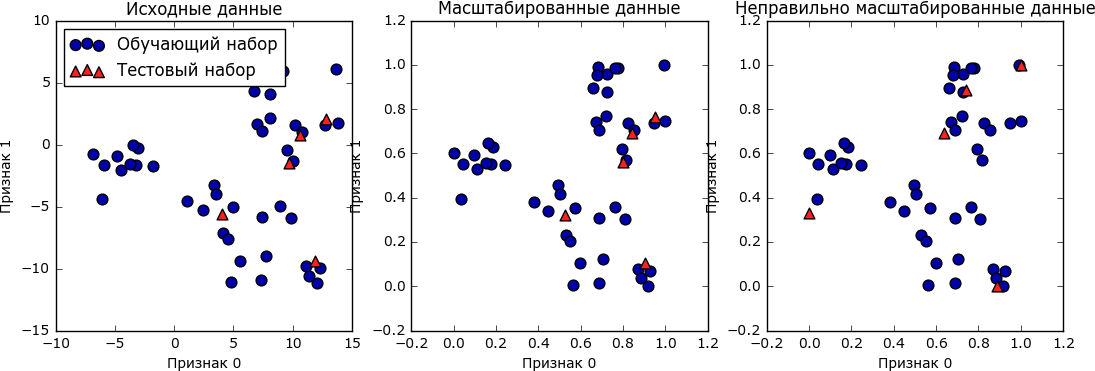
axes[2].scatter(X\_train\_scaled[:, 0], X\_train\_scaled[:, 1], c=mglearn.cm2(0), label="Обучающий набор", s=60)

axes[2].scatter(X\_test\_scaled\_badly[:, 0], X\_test\_scaled\_badly[:, 1], marker='^', c=mglearn.cm2(1), label="Тестовый набор", s=60)

axes[2].set\_title("Неправильно масштабированные данные") for ax in axes:

ax.set\_xlabel("Признак 0")

ax.set\_ylabel("Признак 1")



**Рис. 3.2** Результаты одинакового масштабирования обучающего и тестового наборов (центр) и отдельного масштабирования обучающего и тестового наборов (справа)

Первый график – это немасштабированный двумерный массив данных, наблюдения обучающего набора показаны кружками, а наблюдения тестового набора показаны треугольниками. Второй график

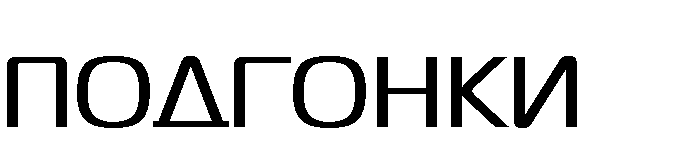
– те же самые данные, но масштабированы с помощью MinMaxScaler. Здесь мы вызвали метод fit для обучающего набора, а затем вызвали метод transform для обучающего и тестового наборов. Как видите, набор данных на втором графике идентичен набору, приведенному на первом

графике, изменились лишь метки осей. Теперь все признаки принимают значения в диапазоне от 0 до 1. Кроме того, видно, что минимальные и максимальные значения признаков в тестовом наборе (треугольники) не равны 0 и 1.

Третий график показывает, что произойдет, если отмасштабируем обучающий и тестовый наборы по отдельности. В этом случае минимальные и максимальные значения признаков в обучающем и тестовом наборах равны 0 и 1. Но теперь набор данных выглядит иначе. Тестовые точки причудливым образом сместились, поскольку масштабированы по-другому. Мы изменили расположение данных произвольным образом. Очевидно, это совсем не то. что нам нужно.

Еще один способ задуматься о неправильности этих действий заключается в том, что представить тестовый набор в виде одной точки. Не существует способа правильно масштабировать единственную точку данных, чтобы с помощью MinMaxScaler получить значения минимума и максимума. Однако размер тестового набора не должен влиять на

обработку данных.



Как правило, вам нужно сначала подогнать модель на некотором наборе данных с помощью метода fit, а затем выполнить преобразование набора с помощью метода transform. Это весьма распространенная задача, которую можно выполнить более эффективно, чем просто вызвать метод fit, а затем вызвать метод transform. Что касается вышеописанного случая, все модели, которые используют метод transform, также позволяют воспользоваться методом fit\_transform. Ниже дан пример использования StandardScaler:

In[9]:

from sklearn.preprocessing import StandardScaler scaler = StandardScaler()

*# последовательно вызываем методы fit и transform (используем цепочку методов)*

X\_scaled = scaler.fit(X).transform(X)

*# тот же самый результат, но более эффективный способ вычислений*

X\_scaled\_d = scaler.fit\_transform(X)

Несмотря на то что применение fit\_transform вовсе не обязательно будет более эффективным для всех моделей, эффективная практика заключается в использовании этого метода для преобразования обучающего набора.



Теперь давайте вернемся к набору данных cancer и посмотрим, как использование MinMaxScaler повлияет на обучение SVC (мы выполняем то же самое масштабирование, что делали в главе 2, но другим способом). Во-первых, давайте для сравнения снова подгоним SVC на исходных данных:

In[10]:

from sklearn.svm import SVC

X\_train, X\_test, y\_train, y\_test = train\_test\_split(cancer.data, cancer.target,

random\_state=0)

svm = SVC(C=100)

svm.fit(X\_train, y\_train)

print("Правильность на тестовом наборе: {:.2f}".format(svm.score(X\_test, y\_test)))

Out[10]:

Правильность на тестовом наборе: 0.63

Теперь давайте отмасштабируем данные с помощью MinMaxScaler

перед тем, как подгонять SVC:

In[11]:

*# предварительная обработка с помощью шкалирования 0-1*

scaler = MinMaxScaler() scaler.fit(X\_train)

X\_train\_scaled = scaler.transform(X\_train) X\_test\_scaled = scaler.transform(X\_test)

*# построение SVM на масштабированных обучающих данных*

svm.fit(X\_train\_scaled, y\_train)

*# оценка правильности для масштабированного тестового набора*

print("Правильность на масштабированном тестовом наборе: {:.2f}".format( svm.score(X\_test\_scaled, y\_test)))

Out[11]:

Правильность на масштабированном тестовом наборе: 0.97

Как мы уже видели ранее, эффект масштабирования данных весьма существенен. Хотя масштабирование данных не предполагает каких-либо сложных математических расчетов, эффективная практика заключается в том, чтобы использовать методы масштабирования, предлагаемые scikit-learn, а не создавать их заново самостоятельно, поскольку легко

ошибиться даже в этих простых вычислениях.

Кроме того, можно легко заменить один алгоритм предварительной обработки на другой, сменив имя используемого класса, поскольку все классы предварительной обработки имеют один и тот же интерфейс, состоящий из методов fit и transform:

In[12]:

*# предварительная обработка с помощью масштабирования*

*# нулевым средним и единичной дисперсией*

from sklearn.preprocessing import StandardScaler scaler = StandardScaler()

scaler.fit(X\_train)

X\_train\_scaled = scaler.transform(X\_train) X\_test\_scaled = scaler.transform(X\_test)

*# построение SVM на масштабированных обучающих данных*

svm.fit(X\_train\_scaled, y\_train)

*# оценка правильности для масштабированного тестового набора*

print("Правильность SVM на тестовом наборе: {:.2f}".format(svm.score(X\_test\_scaled, y\_test)))

Out[12]:

Правильность SVM на тестовом наборе: 0.96

Теперь узнав, как работают простые преобразования, выполняющие предварительную обработку данных, давайте перейдем к более интересным преобразованиям, использующим машинное обучение без учителя.



Как мы уже говорили ранее, преобразование данных с помощью неконтролируемого обучения может быть обусловлено многими причинами. Наиболее распространенные причины – визуализация,

сжатие данных, а также поиск такого представления данных, которое даст больше информации в ходе дальнейшей обработки.

Одним из самых простых и наиболее широко используемых алгоритмов контролируемого обучения является анализ главных компонент (principal component analysis, PCA). Кроме того, мы рассмотрим еще два алгоритма: факторизацию неотрицательных матриц (non-negative matrix factorization, NMF), которая обычно используется для выделения признаков, и стохастическое вложение соседей с распределением Стьюдента (t-distributed stochastic neighbor embedding, t-SNE), которое обычно используется для визуализации с использованием двумерных диаграмм рассеяния.

# **ЛБ 12 Анализ главных компонент (PSA)**

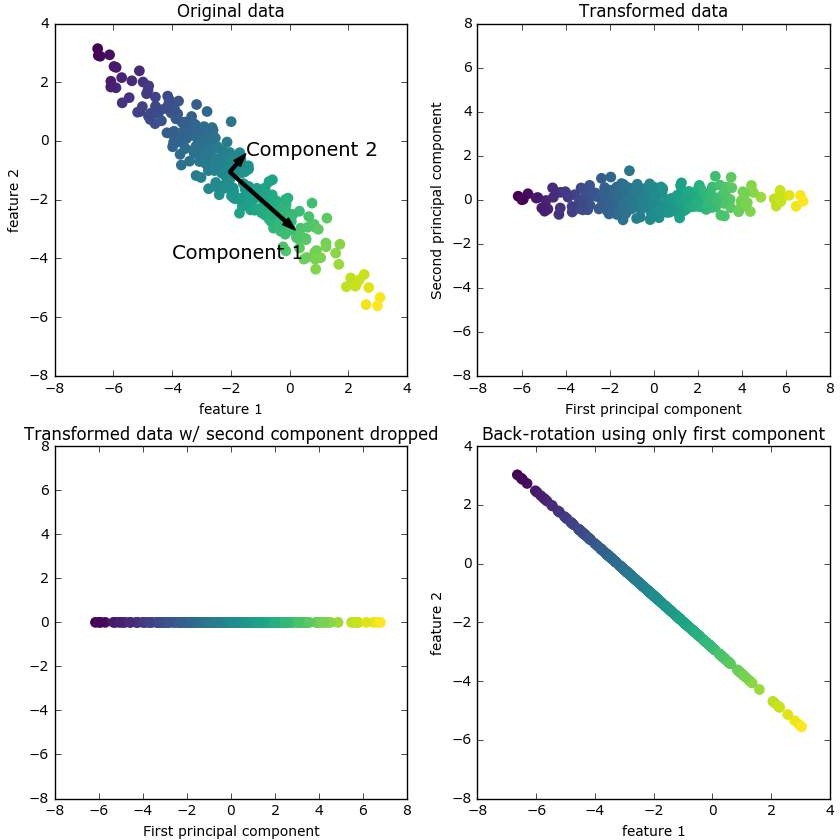
Анализ главных компонент представляет собой метод, который осуществляет вращение данных с тем, чтобы преобразованные признаки не коррелировали между собой. Часто это вращение сопровождается выбором подмножества новых признаков в зависимости от их важности с точки зрения интерпретации данных. Следующий пример (рис. 3.3) иллюстрирует результат применения PCA к синтетическому двумерному массиву данных:

In[13]:

mglearn.plots.plot\_pca\_illustration()

Первый рис. (вверху слева) показывает исходные точки данных, выделенные цветом для лучшей дискриминации. Алгоритм начинает работу с того, что сначала находит направление максимальной дисперсии, помеченное как «компонента 1». Речь идет о направлении (или векторе) данных, который содержит б*о*льшую часть информации, или другими словами, направление, вдоль которого признаки коррелируют друг с другом сильнее всего. Затем алгоритм находит направление, которое содержит наибольшее количество информации, и при этом ортогонально (расположено под прямым углом) первому направлению. В двумерном пространстве существует только одна возможная ориентация, расположенная под прямым углом, но в пространствах большей размерности может быть (бесконечно) много ортогональных направлений. Хотя эти две компоненты изображаются в виде стрелок, на самом деле не имеет значения, где начало, а где конец, мы могли бы нарисовать первую компоненту, выходящую из центра в верхний левый угол, а не в нижний правый. Направления, найденные с помощью этого алгоритма, называются *главными компонентами* (*principal components*), поскольку они являются основными

направлениями дисперсии данных. В целом максимально возможное количество главных компонент равно количеству исходных признаков.



Исходные данные

Преобразованные данные

Первая главная компонента

Обратное вращение c использованием лишь первой компоненты

Первая главная компонента

признак 1

Преобразованные данные с удаленной второй компонентой

признак 1

признак 2

Вторая главная компонента

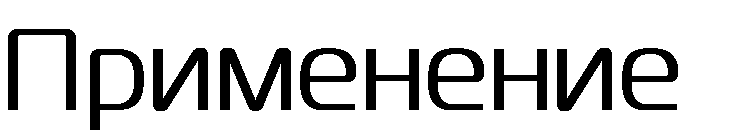
**Рис. 3.3** Преобразование данных с помощью PCA

признак 2

Второй график (вверху справа) показывает те же самые данные, но теперь повернутые таким образом, что первая главная компонента совпадает с осью х, а вторая главная компонента совпадает с осью у. Перед вращением из каждого значения данных вычитается среднее, таким образом, преобразованные данные центрированы около нуля. В новом представлении данных, найденном с помощью PCA, две оси становятся некоррелированными. Это означает, что в новом представлении все элементы корреляционной матрицы данных, кроме диагональных, будут равны нулю.

Мы можем использовать PCA для уменьшения размерности, сохранив лишь несколько главных компонент. В данном примере мы можем оставить лишь первую главную компоненту, как показано на третьем графике рис. 3.3 (внизу слева). Это уменьшит размерность данных: из двумерного массива данных получаем одномерный массив данных. Однако следует отметить, что вместо того, чтобы оставить лишь один из исходных признаков, мы находим наиболее интересное направление (выходящее из верхнего левого угла в нижний правый на первом графике) и оставляем это направление, т.е. первую главную компоненту. И, наконец, мы можем отменить вращение и добавить обратно среднее значение к значениям данных. В итоге получим данные, показанные на последнем графике рис. 3.3. Эти точки располагаются в пространстве исходных признаков, но мы оставили лишь информацию, содержащуюся в первой главной компоненте. Это преобразование иногда используется, чтобы удалить эффект шума из данных или показать, какая часть

информации сохраняется при использовании главных компонент.



Одним из наиболее распространенных применений PCA является

визуализация высокоразмерных наборов данных. Как мы видели в главе 1, довольно сложно построить диаграммы рассеяния для данных, которые включают больше двух признаков. Для набора данных Iris мы смогли построить матрицу диаграмм рассеяния (рис. 1.3 в главе 1), которая дала нам частичное представление о данных, показав все возможные комбинации двух признаков. Но если мы захотим взглянуть на набор данных Breast Cancer, использование матрицы диаграмм рассеяния будет затруднительным. Этот набор данных содержит 30 признаков, которые привели бы к 30 \* 14 = 420 диаграммам рассеяния! Мы никогда не сможем детально просмотреть все эти графики, не говоря уже об их интерпретации.

Впрочем, можно воспользовать более простой визуализацией, вычислив гистограммы распределения значений признаков для двух классов, доброкачественных и злокачественных опухолей (рис. 3.4):

In[14]:

fig, axes = plt.subplots(15, 2, figsize=(10, 20)) malignant = cancer.data[cancer.target == 0] benign = cancer.data[cancer.target == 1]

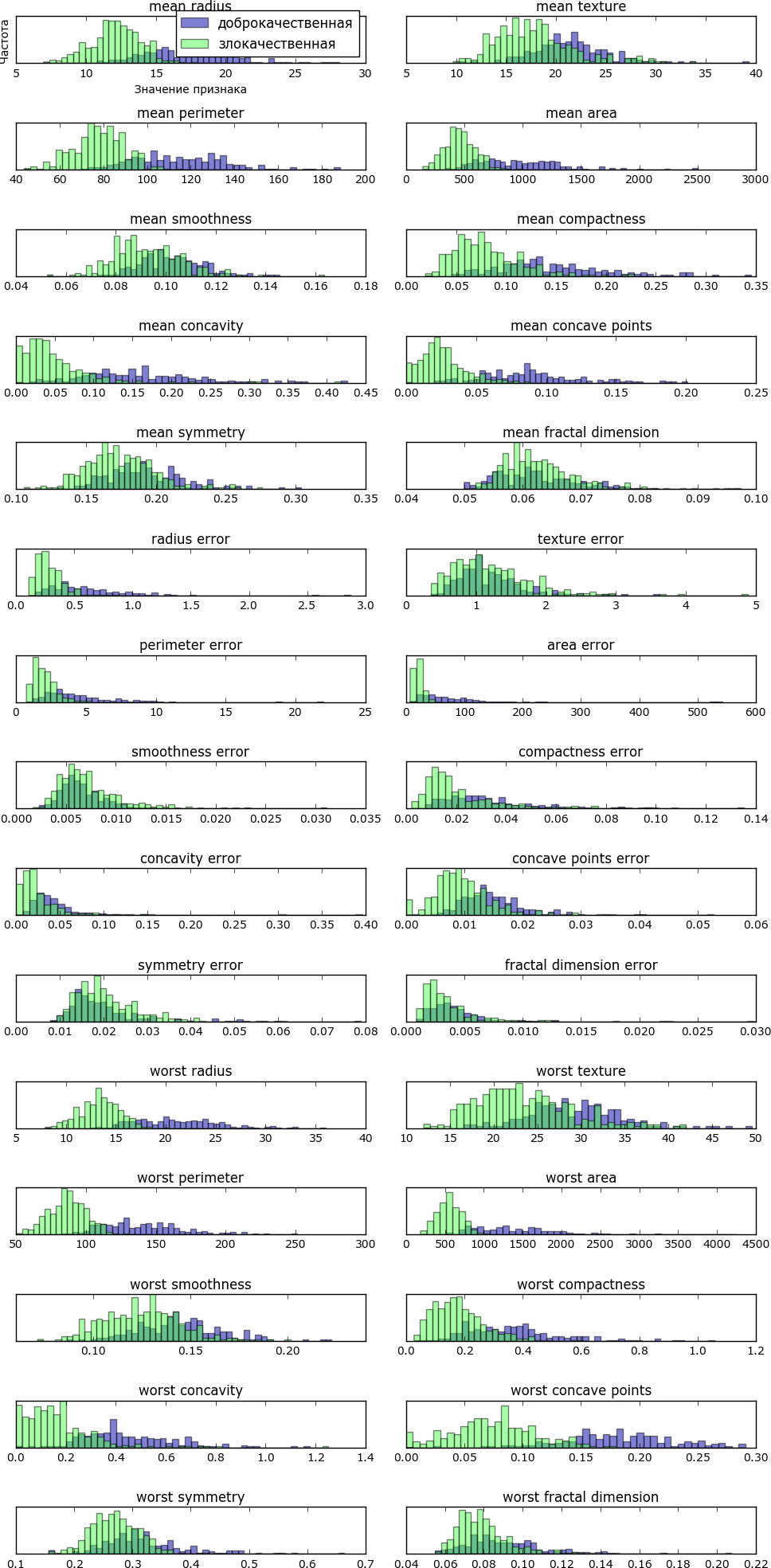
ax = axes.ravel() for i in range(30):

\_, bins = np.histogram(cancer.data[:, i], bins=50)

ax[i].hist(malignant[:, i], bins=bins, color=mglearn.cm3(0), alpha=.5) ax[i].hist(benign[:, i], bins=bins, color=mglearn.cm3(2), alpha=.5) ax[i].set\_title(cancer.feature\_names[i])

ax[i].set\_yticks(()) ax[0].set\_xlabel("Значение признака") ax[0].set\_ylabel("Частота")

ax[0].legend(["доброкачественная", "злокачественная"], loc="best") fig.tight\_layout()



**Рис. 3.4** Преобразование данных с помощью PCA

В данном случае мы строим для каждого признака гистограмму, подсчитывая частоту встречаемости точек данных в пределах границ интервалов (этот интервал еще называют бином). Каждый график содержит две наложенные друг на друга гистограммы, первая – для всех точек, относящихся к классу «доброкачественная опухоль» (синий цвет), а вторая – для всех точек, относящихся к классу «злокачественная опухоль» (зеленый цвет). Это дает нам некоторое представление о распределении каждого признака по двум классам и позволяет нам строить предположения о том, какие признаки лучше всего дискриминируют злокачественные и доброкачественные опухоли. Например, признак «smoothness error», похоже, довольно малоинформативен, потому что две гистограммы, построенные для данного признака, большей частью накладываются друг на друга, в то время признак «worst concave points» кажется весьма информативным, поскольку гистрограммы, построенные для этого признака, практически не накрывают друг друга.

Однако этот график не дает нам никакой информации о взаимодействии между переменными и взаимосвязях между признаками и классами зависимой переменной. Используя PCA, мы можем учесть главные взаимодействия и получить несколько более полную картину. Мы можем найти первые две главные компоненты и визуализировать данные в этом новом двумерном пространстве с помощью одной диаграммы рассеяния.

Перед тем, как применить PCA, мы отмасштабируем наши данные таким образом, чтобы каждый признак имел единичную дисперсию, воспользовавшись StandardScaler:

In[15]:

from sklearn.datasets import load\_breast\_cancer cancer = load\_breast\_cancer()

scaler = StandardScaler() scaler.fit(cancer.data)

X\_scaled = scaler.transform(cancer.data)

Обучение PCA и его применение так же просто, как применение преобразований, выполняющихся в ходе предварительной обработки. Мы создаем экземпляр объекта PCA, находим главные компоненты, вызвав метод fit, а затем применяем вращение и снижение размерности, вызвав метод transform. По умолчанию PCA лишь поворачивает (и смещает) данные, но сохраняет все главные компоненты. Чтобы

уменьшить размерность данных, нам нужно указать, сколько компонент мы хотим сохранить при создании объекта PCA:

In[16]:

from sklearn.decomposition import PCA

*# оставляем первые две главные компоненты*

pca = PCA(n\_components=2)

*# подгоняем модель PCA на наборе данных breast cancer*

pca.fit(X\_scaled)

*# преобразуем данные к первым двум главным компонентам*

X\_pca = pca.transform(X\_scaled)

print("Форма исходного массива: {}".format(str(X\_scaled.shape)))

print("Форма массива после сокращения размерности: {}".format(str(X\_pca.shape)))

Out[16]:

Форма исходного массива: (569, 30)

Форма массива после сокращения: (569, 2)

Теперь мы можем построить график первых двух главных компонент (рис. 3.5):

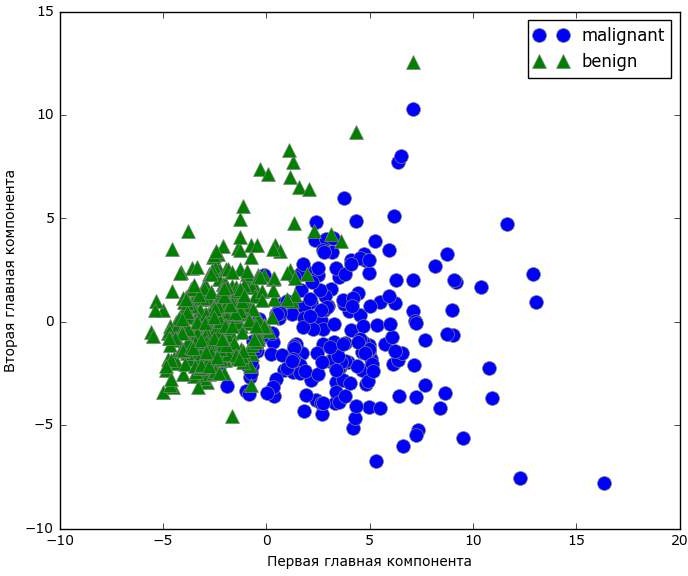
In[17]:

*# строим график первых двух главных компонент, классы выделены цветом*

plt.figure(figsize=(8, 8))

mglearn.discrete\_scatter(X\_pca[:, 0], X\_pca[:, 1], cancer.target) plt.legend(cancer.target\_names, loc="best") plt.gca().set\_aspect("equal")

plt.xlabel("Первая главная компонента") plt.ylabel("Вторая главная компонента")



**Рис. 3.5** Двумерная диаграмма рассеяния для набора данных Breast Cancer с использованием первых двух главных компонент

Важно отметить, что РСА является методом машинного обучения без учителя и не использует какой-либо информации о классах при поиске поворота. Он просто анализирует корреляционные связи в данных. Для точечного графика, показанного здесь, мы построили график, где по оси x отложена первая главная компонента, по оси y – вторая главная компонента, а затем воспользовались информацией о классах, чтобы выделить точки разным цветом. Вы можете увидеть, что в рассматриваемом двумерном пространстве эти два класса разделены достаточно хорошо. Это наводит на мысль, что даже линейный классификатор (который проведет прямую линию в этом пространстве) сможет достаточно хорошо разделить два класса. Кроме того, мы можем увидеть, что случаи злокачественных опухолей (синие точки) более распространены, чем случаи доброкачественных опухолей (зеленые точки) – что отчасти было видно на гистограммах рис. 3.4.

Недостаток PCA заключается в том, что эти две оси графика часто бывает сложно интерпретировать. Главные компоненты соответствуют направлениям данных, поэтому они представляют собой комбинации исходных признаков. Однако, как мы скоро увидим, эти комбинации обычно очень сложны. Сами главные компоненты могут быть сохранены в атрибуте components\_ объекта PCA в ходе подгонки:

In[18]:

print("форма главных компонент: {}".format(pca.components\_.shape))

Out[18]:

форма главных компонент: (2, 30)

Каждая строка в атрибуте components\_ соответствует одной главной компоненте и они отсортированы по важности (первой приводится первая главная компонента и т.д.). Столбцы соответствуют атрибуту исходных признаков для объекта PCA в этом примере, «mean radius»,

«mean texture» и т.д. Давайте посмотрим на содержимое атрибута

components\_:

components\_:

In[19]:

print("компоненты PCA:\n{}".format(pca.components\_))

Out[19]:

компоненты PCA:

[[ 0.219 0.104 0.228 0.221 0.143 0.239 0.258 0.261 0.138 0.064

0.206 0.017 0.211 0.203 0.015 0.17 0.154 0.183 0.042 0.103

0.228 0.104 0.237 0.225 0.128 0.21 0.229 0.251 0.123 0.132]

[-0.234 -0.06 -0.215 -0.231 0.186 0.152 0.06 -0.035 0.19 0.367

-0.106 0.09 -0.089 -0.152 0.204 0.233 0.197 0.13 0.184 0.28

-0.22 -0.045 -0.2 -0.219 0.172 0.144 0.098 -0.008 0.142 0.275]]

Кроме того, с помощью тепловой карты (рис. 3.6) мы можем визуализировать коэффициенты, чтобы упростить их интерпретацию:

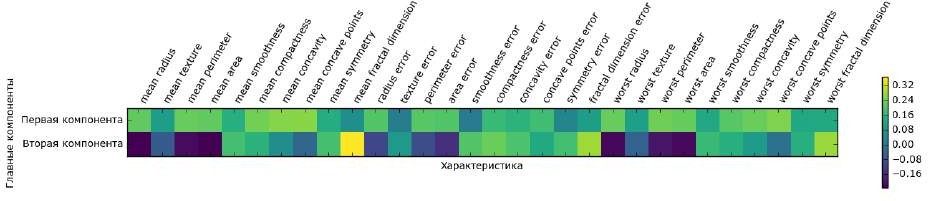
In[20]:

plt.matshow(pca.components\_, cmap='viridis')

plt.yticks([0, 1], ["Первая компонента", "Вторая компонента"]) plt.colorbar()

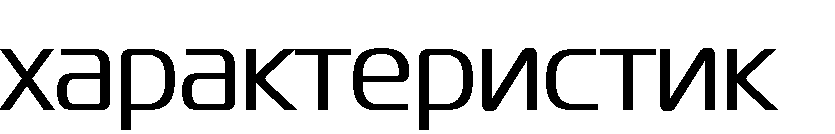
plt.xticks(range(len(cancer.feature\_names)), cancer.feature\_names, rotation=60, ha='left')

plt.xlabel("Характеристика") plt.ylabel("Главные компоненты")



**Рис. 3.6** Тепловая карта первых двух главных компонент для набора данных рака Breast Cancer

Вы можете увидеть, что в первой компоненте коэффициенты всех признаков имеют одинаковый знак (они положительные, но, как мы уже говорили ранее, не имеет значения, какое направление указывает стрелка). Это означает, что существует общая корреляция между всеми признаками. Высоким значениям одного признака будут соответствовать высокие значения остальных признаков. Во второй компоненте коэффициенты признаков имеют разные знаки. Обе компоненты включают все 30 признаков. Смешивание всех признаков – это как раз то, что усложняет интерпретацию осей на рис. 3.6.



Еще одно применение PCA, о котором мы уже упоминали ранее, – это

выделение признаков. Идея, лежащая в основе выделения признаков, заключается в поиске нового представления данных, которое в отличие от исходного лучше подходит для анализа. Отличный пример, показывающий, что выделение признаков может быть полезно, – это работа с изображениями. Изображения состоят из пикселей, обычно хранящихся в виде интенсивностей красной, зеленой и синей составляющих цвета (RGB). Объекты в изображениях, как правило, состоят из тысяч пикселей и лишь все вместе эти пиксели приобретают смысл.

Мы приведем очень простой пример того, как можно применить выделение признаков к изображениям с помощью PCA. Для этого мы воспользуемся набором данных Labeled Faces in the Wild. Этот набор данных содержит изображения лиц знаменитостей, загруженных из Интернета, и включает в себя лица политиков, певцов, актеров и спортсменов с начала 2000-х годов. Мы преобразуем эти фотографии в

оттенки серого, а также уменьшим их для более быстрой обработки. Вы можете увидеть некоторые изображения на рис. 3.7:

In[21]:

from sklearn.datasets import fetch\_lfw\_people

people = fetch\_lfw\_people(min\_faces\_per\_person=20, resize=0.7) image\_shape = people.images[0].shape

fix, axes = plt.subplots(2, 5, figsize=(15, 8),

subplot\_kw={'xticks': (), 'yticks': ()})

for target, image, ax in zip(people.target, people.images, axes.ravel()): ax.imshow(image)

ax.set\_title(people.target\_names[target])



**Рис. 3.7** Некоторые изображения из набора данных Labeled Faces in the Wild

Получаем 3023 изображения размером 87 x 65 пикселей, принадлежащие 62 различным людям:

In[22]:

print("форма массива изображений лиц: {}".format(people.images.shape)) print("количество классов: {}".format(len(people.target\_names)))

Out[22]:

форма массива изображений лиц: (3023, 87, 65)

количество классов: 62

Однако данные немного асимметричны. Как вы можете здесь увидеть, он содержит большое количество изображений Джорджа Буша и Колина Пауэлла:

In[23]:

*# вычисляем частоту встречаемости каждого ответа*

counts = np.bincount(people.target)

*# печатаем частоты рядом с ответами*

for i, (count, name) in enumerate(zip(counts, people.target\_names)): print("{0:25} {1:3}".format(name, count), end=' ')

if (i + 1) % 3 == 0:

print()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Out[23]: |  | | | | |
| Alejandro Toledo | 39 | Alvaro Uribe | 35 | Amelie Mauresmo | 21 |
| Andre Agassi | 36 | Angelina Jolie | 20 | Ariel Sharon | 77 |
| Arnold Schwarzenegger | 42 | Atal Bihari Vajpayee | 24 | Bill Clinton | 29 |
| Carlos Menem | 21 | Colin Powell | 236 | David Beckham | 31 |
| Donald Rumsfeld | 121 | George Robertson | 22 | George W Bush | 530 |
| Gerhard Schroeder | 109 | Gloria Macapagal Arroyo | 44 | Gray Davis | 26 |
| Guillermo Coria | 30 | Hamid Karzai | 22 | Hans Blix | 39 |
| Hugo Chavez | 71 | Igor Ivanov | 20 | Jack Straw | 28 |
| Jacques Chirac | 52 | Jean Chretien | 55 | Jennifer Aniston | 21 |
| Jennifer Capriati | 42 | Jennifer Lopez | 21 | Jeremy Greenstock | 24 |
| Jiang Zemin | 20 | John Ashcroft | 53 | John Negroponte | 31 |
| Jose Maria Aznar | 23 | Juan Carlos Ferrero | 28 | Junichiro Koizumi | 60 |
| Kofi Annan | 32 | Laura Bush | 41 | Lindsay Davenport | 22 |
| Lleyton Hewitt | 41 | Luiz Inacio Lula da Silva | 48 | Mahmoud Abbas | 29 |
| Megawati Sukarnoputri | 33 | Michael Bloomberg | 20 | Naomi Watts | 22 |
| Nestor Kirchner | 37 | Paul Bremer | 20 | Pete Sampras | 22 |
| Recep Tayyip Erdogan | 30 | Ricardo Lagos | 27 | Roh Moo-hyun | 32 |
| Rudolph Giuliani | 26 | Saddam Hussein | 23 | Serena Williams | 52 |
| Silvio Berlusconi | 33 | Tiger Woods | 23 | Tom Daschle | 25 |
| Tom Ridge | 33 | Tony Blair | 144 | Vicente Fox | 32 |
| Vladimir Putin | 49 | Winona Ryder | 24 |  |  |

Чтобы данные стали менее асимметричными, мы будем рассматривать не более 50 изображений каждого человека (в противном случае выделение признаков будет перегружено большим количеством изображений Джорджа Буша):

In[24]:

mask = np.zeros(people.target.shape, dtype=np.bool) for target in np.unique(people.target):

mask[np.where(people.target == target)[0][:50]] = 1

X\_people = people.data[mask] y\_people = people.target[mask]

*# для получения большей стабильности масштабируем шкалу оттенков серого так, чтобы значения*

*# были в диапазоне от 0 до 1 вместо использования шкалы значений от 0 до 255*

X\_people = X\_people / 255.

Общая задача распознавания лиц заключается в том, чтобы спросить, не принадлежит ли незнакомое фото уже известному человеку из базы данных. Она применяется при составлении фотоколлекций, в социальных сетях и программах обеспечения безопасности. Один из способов решения этой задачи заключается в построении классификатора, в котором каждый человек представляет собой отдельный класс. Однако изображения, записанные в базах лиц, обычно принадлежат большому количеству самых различных людей и при этом очень мало фотографий принадлежат одному и тому же человеку (то есть очень мало обучающих примеров, принадлежащих одному классу). Для

большинства классификаторов это представляет проблему. Кроме того, часто необходимо добавить фотографии новых людей, при этом не перестраивая заново огромную модель.

Самое простое решение – использовать классификатор одного ближайшего соседа, который ищет лицо, наиболее схожее с классифицируемым. Этот классификатор в принципе может работать только с одним обучающим примером в классе. Давайте посмотрим, насколько хорошо здесь сработает KNeighborsClassifier:

In[25]:

from sklearn.neighbors import KNeighborsClassifier *# разбиваем данные на обучающий и тестовый наборы* X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_people, y\_people, stratify=y\_people, random\_state=0)

*# строим KNeighborsClassifier с одним соседом* knn = KNeighborsClassifier(n\_neighbors=1) knn.fit(X\_train, y\_train)

print("Правильность на тестовом наборе для 1-nn: {:.2f}".format(knn.score(X\_test, y\_test)))

Out[25]:

Правильность на тестовом наборе для 1-nn: 0.27

Мы получаем правильность 26.6%, на самом деле это неплохо для классификационной задачи с 62 классами (случайное угадывание даст вам правильность около 1/62 = 1.61%), но и не так велико. Мы правильно распознаем лишь каждое четвертое изображение человека.

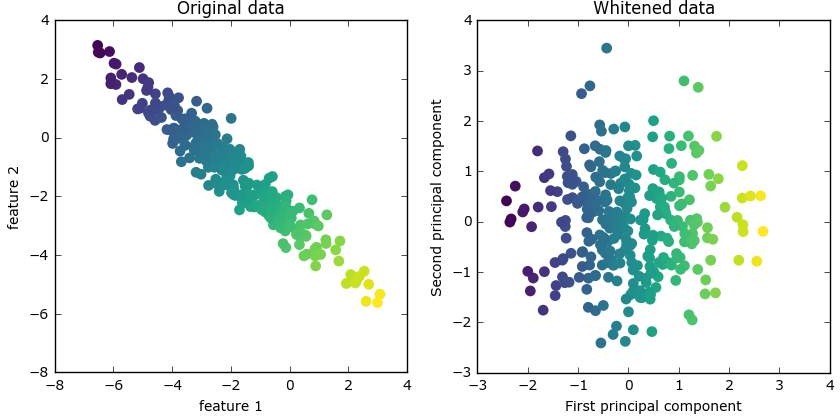
И вот именно здесь применяется PCA. Вычисление расстояний в исходном пиксельном пространстве – довольно неудачный способ измерить сходство между лицами. Используя пиксельное представление для сопоставления двух изображений, мы сравниваем значение каждого отдельного пикселя по шкале градаций серого со значением пикселя в соответствующем положении на другом изображении. Это представление довольно сильно отличается от интерпретации изображений лиц людьми и крайне трудно выделить характеристики лица с использованием этого исходного представления. Например, использование пиксельных расстояний означает, что смещение лица на один пиксель вправо соответствует резкому изменению, дающему совершенно другое представление данных. Мы рассчитываем на то, что использование расстояний вдоль главных компонент может улучшить правильность. Здесь мы воспользуемся опцией PCA *выбеливание* (*whitening*), которая преобразует компоненты к одному и тому же масштабу. Операция

выбеливания аналогична применению StandardScaler после преобразования. Повторно используя данные, приведенные на рис. 3.3,

выбеливание не только поворачивает данные, но и масштабирует их таким образом, чтобы центральный график представлял собой окружность вместо эллипса (см. рис. 3.8):

In[26]:

mglearn.plots.plot\_pca\_whitening()



Исходные данные

Выбеленные данные

характеристика 1

Первая главная компонента

характеристика 2

Вторая главная компонента

**Рис. 3.8** Преобразование данных с использованием выбеливания

Мы подгоняем объект PCA на обучающих данных и извлекаем первые 100 главных компонент. Затем мы преобразуем обучающие и тестовые данные:

In[27]:

pca = PCA(n\_components=100, whiten=True, random\_state=0).fit(X\_train) X\_train\_pca = pca.transform(X\_train)

X\_test\_pca = pca.transform(X\_test)

print("обучающие данные после PCA: {}".format(X\_train\_pca.shape)) Out[27]:

обучающие данные после PCA: (1537, 100)

Новые данные содержат 100 новых признаков, первые 100 главных компонент. Теперь мы можем использовать новое представление, чтобы классифицировать наши изображения, используя классификатор одного ближайшего соседа:

In[28]:

knn = KNeighborsClassifier(n\_neighbors=1) knn.fit(X\_train\_pca, y\_train)

print("Правильность на тестовом наборе: {:.2f}".format(knn.score(X\_test\_pca, y\_test)))

Out[28]:

Правильность на тестовом наборе: 0.36

Наша правильность улучшилась весьма значительно, с 26.6% до 35.7%, это подтверждает наше предположение о том, что главные компоненты могут дать лучшее представление данных.

Работая с изображениями, мы можем легко визуализировать найденные главные компоненты. Вспомним, что компоненты соответствуют направлениям в пространстве входных данных. Пространство входных данных здесь представляет собой изображения в градациях серого размером 87x65 пикселей, поэтому направления внутри этого пространства также являются изображениями в градациях серого размером 87x65 пикселей.

Давайте посмотрим на первые несколько главных компонент (рис.

3.9):

In[29]:

print("форма pca.components\_: {}".format(pca.components\_.shape))

Out[29]:

форма pca.components\_: (100, 5655)

In[30]:

fix, axes = plt.subplots(3, 5, figsize=(15, 12),

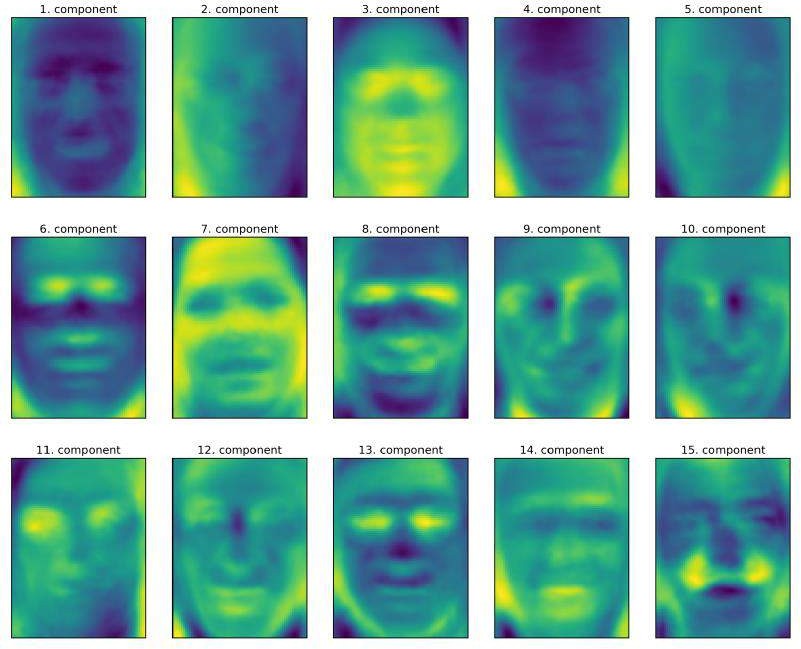
subplot\_kw={'xticks': (), 'yticks': ()})

for i, (component, ax) in enumerate(zip(pca.components\_, axes.ravel())): ax.imshow(component.reshape(image\_shape),

cmap='viridis')

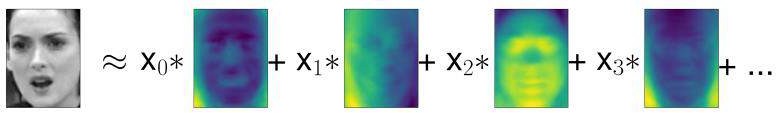
ax.set\_title("{}. component".format((i + 1)))

Несмотря на то что мы, конечно, не сможем понять весь содержательный смысл этих компонент, мы можем догадаться, какие характеристики изображений лиц были выделены некоторыми компонентами. Похоже, что первая компонента главным образом кодирует контраст между лицом и фоном, а вторая компонента кодирует различия в освещенности между правой и левой половинами лица и т.д. Хотя это представление данных в отличие от исходных значений пикселей немного содержательнее, оно по-прежнему весьма далеко от того, как человек привык воспринимать лицо. Поскольку модель PCA основана на пикселях, выравнивание изображения лица (положения глаз, подбородка и носа) и освещенность оказывают сильное влияние на степень сходства двух пиксельных изображений. Однако выравнивание и освещенность, вероятно, будут совсем не теми характеристиками, которые человек будет воспринимать в первую очередь. Когда людей просят оценить сходство между лицами, они в большей степени руководствуются такими признаками, как возраст, пол, выражение лица и прическа, то есть признаками, которые трудно выделить, исходя из интенсивностей пикселей. Важно помнить, что, как правило, алгоритмы в отличие от человека интерпретируют данные (в частности, визуальные данные, например, изображения популярных людей) совершенно по- другому.



**Рис. 3.9** Собственные векторы первых 15 компонент для набора лиц

Впрочем, давайте вернемся к конкретному случаю использования PCA. Мы кратко рассказали о преобразовании PCA как способе поворота данных с последующим удалением компонент, имеющих низкую дисперсию. Еще одна полезная интерпретация заключается в том, чтобы попытаться вычислить значения новых признаков, полученные после поворота PCA, таким образом, мы можем записать тестовые точки в виде взвешенной суммы главных компонент (см. рис. 3.10).



**Рис. 3.10** Схематическое изображение PCA, осуществляющего разложение изображения на взвешенную сумму компонент

Здесь *x* , *x* и т.д. являются коэффициентами главных компонент для

1

2

конкретной точки данных, другими словами, они представляют собой изображение в новом пространстве, полученном в результате вращения. Еще один способ понять, что делает модель PCA – реконструировать исходные данные, используя лишь некоторые компоненты. На третьем графике рис. 3.3 мы удалили вторую компоненту, затем мы отменили вращение и добавили обратно среднее значение, чтобы получить новые точки в исходном пространстве с удаленной второй компонентой, как показано на последнем графике рис. 3.3. Мы можем выполнить аналогичное преобразование для лиц, сократив данные за счет использования лишь некоторых главных компонент и вернувшись затем в исходное пространство. Это возвращение в пространство исходных признаков можно выполнить с помощью метода inverse\_transform.

Здесь мы визуализируем результаты реконструкции некоторых лиц,

используя 10, 50, 100, 500 и 2000 компонент (рис. 3.11):

In[32]:

mglearn.plots.plot\_pca\_faces(X\_train, X\_test, image\_shape)



**Рис. 3.11** Реконструкция трех изображений лица с помощью постепенного увеличения числа главных компонент

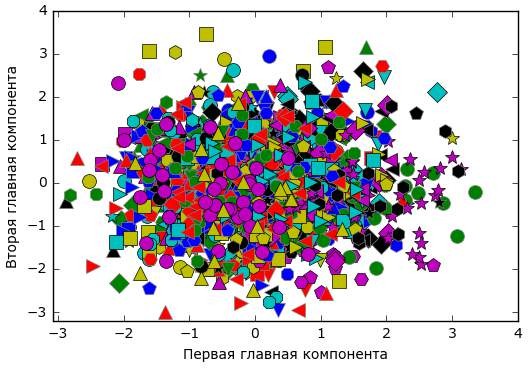
Вы можете увидеть, что, когда мы используем лишь первые 10 главных компонент, фиксируется лишь общая суть картинки, например, ориентация лица и освещенность. По мере увеличения количества используемых компонент сохраняется все больше деталей изображения. Это соответствует включению большего числа слагаемых в сумму, показанную на рис. 3.10. Использование числа компонент, равного числу имеющихся пикселей, означало бы, что мы, осуществив поворот, сохранили всю информацию и можем идеально реконструировать изображение.

Кроме того, мы можем применить PCA для визуализации всех лиц набора на диаграмме рассеяния, воспользовавшись первыми двумя главными компонентами (рис. 3.12). Для этого мы выделим классы, соответствующие лицам, с помощью определенного цвета и формы (аналогично тому, что делали для набора данных cancer):

In[33]:

mglearn.discrete\_scatter(X\_train\_pca[:, 0], X\_train\_pca[:, 1], y\_train) plt.xlabel("Первая главная компонента")

plt.ylabel("Вторая главная компонента")



**Рис. 3.12** Диаграмма рассеяния для набора лиц, использующая первые две главные компоненты (см. рис. 3.5 с соответствующим изображением для набора данных cancer)

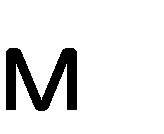
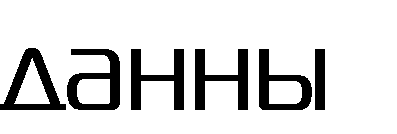
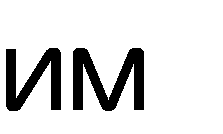
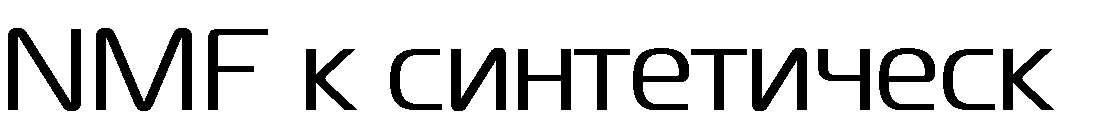
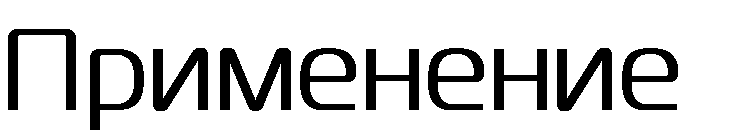
Из рис. видно, когда мы используем лишь первые две главные компоненты, все данные представляют собой просто одно большое скопление данных без видимого разделения классов. Данный факт неудивителен, учитывая, что даже при использовании 10 компонент, как

уже было показано ранее на рис. 3.11, PCA фиксирует самые общие характеристики лиц.

# **ЛБ 13 Факторизация неотрицательных матриц**

Факторизация неотрицательных матриц – еще один алгоритм машинного обучения без учителя, цель которого – выделить полезные характеристики. Он работает так же, как PCA, а также его можно использовать для уменьшения размерности. Как и в РСА, мы пытаемся записать каждую точку данных в виде взвешенной суммы некоторых компонентов, как показано на рис. 3.10. Однако, если в PCA нам нужно получить ортогональные компоненты, объясняющие максимально возможную долю дисперсии данных, то в NMF нам нужно получить неотрицательные компоненты и коэффициенты, то есть нам нужны компоненты и коэффициенты, которые больше или равны нулю. Поэтому этот метод может быть применен только к тем данным, в которых характеристики имеют неотрицательные значения, поскольку неотрицательная сумма неотрицательных компонентов не может быть отрицательной.

Процесс разложения данных на неотрицательную взвешенную сумму особенно полезен для данных, созданных в результате объединения (или наложения) нескольких независимых источников, например, аудиотреков с голосами нескольких людей, музыки с большим количеством инструментов. В таких ситуациях NMF может найти исходные компоненты, которые лежат в основе объединенных данных. В целом NMF позволяет получить более интерпретабельные компоненты, чем PCA, поскольку отрицательные компоненты и коэффициенты могут привести к получению трудных для интерпретации взаимокомпенсирующих эффектов. Например, собственные лица на рис. 3.9, содержат как положительные, так и отрицательные характеристики, и, как мы уже упоминали в описании PCA, знаки имеют фактически произвольный характер. Перед тем, как применить NMF к набору лиц, давайте заново посмотрим на наши синтетические данные.



В отличие от PCA, чтобы применить NMF к данным, мы должны

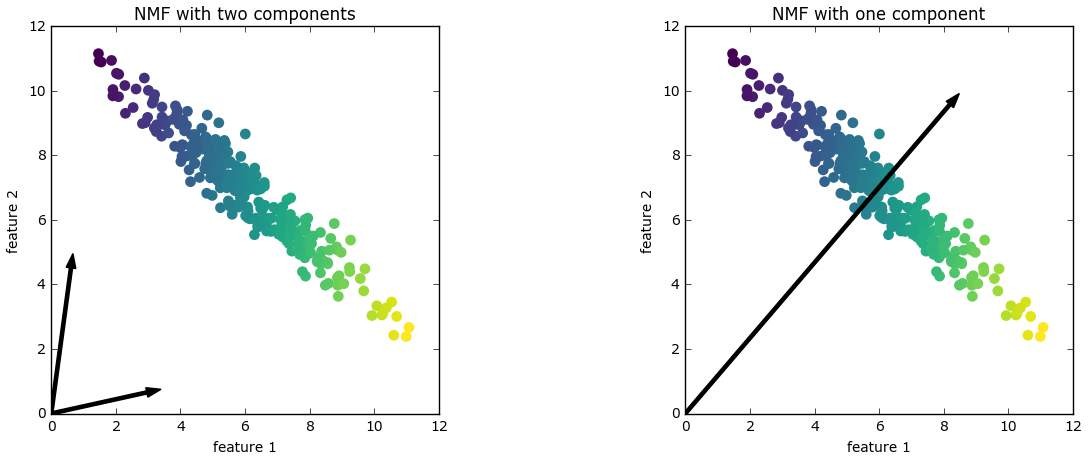
убедиться, что они имеют положительные значения. Это означает, что для NMF расположение данных относительно начала координат (0, 0) имеет реальное значение. Поэтому извлекаемые неотрицательные компоненты можно представить в виде направлений, выходящих из начала координат (0, 0) к данным.

Следующий пример (рис. 3-13) показывает результаты применения

NMF к двумерным синтетическим данным:

In[34]:

mglearn.plots.plot\_nmf\_illustration()



NMF с двумя компонентами

NMF с одной компонентой

характеристика 1

характеристика 1

характеристика 2

характеристика 2

**Рис. 3.13** Компоненты, найденные в результате факторизации неотрицательных матриц с двумя компонентами (слева) и

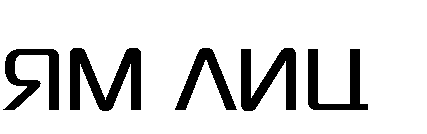
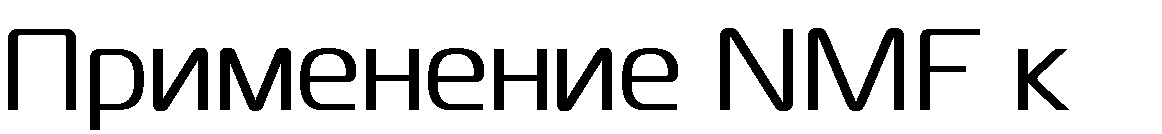
одной компонентой (справа)

Для NMF с двумя компонентами (график слева) ясно, что все точки данных можно записать в виде комбинации положительных значений этих двух компонент. Если количества компонент достаточно для того, чтобы полностью реконструировать данные (количество компонент совпадает с количеством характеристик), алгоритм будет выбирать направления, указывающие на экстремальные значения данных.

При использовании лишь одной компоненты NMF выделяет компоненту, которая указывает на среднее значение как значение, лучше всего объясняющее данные. Видно, что в отличие от PCA уменьшение числа компонент удаляет не только некоторые направления, но и создает совершенно другой набор компонент! Кроме того, компоненты NMF не упорядочены каким-либо определенным образом, поэтому здесь нет такого понятия, как «первая неотрицательная компонента»: все компоненты играют одинаковую роль.

NMF использует случайную инициализацию, поэтому разные стартовые значения дают различные результаты. В относительно простых случаях (например, синтетические данные с двумя компонентами), где все данные можно прекрасно объяснить, случайность мало влияет на результат (хотя она может изменить порядок или масштаб компонент). В более сложных ситуациях использование

различных случайных значений может привести к радикальным изменениям.



Теперь давайте применим NMF к набору данных Labeled Faces in the

Wild, который мы использовали ранее. Основной параметр NMF – количество извлекаемых компонент. Как правило, количество извлекаемых компонент меньше количества входных характеристик (в противном случае, данные можно объяснить, представив каждый пиксель отдельной компонентой).

Во-первых, давайте выясним, как количество компонент влияет на качество восстановления данных с помощью NMF (рис. 3.14):

In[35]:

mglearn.plots.plot\_nmf\_faces(X\_train, X\_test, image\_shape)



**Рис. 3.14** Реконструкция трех изображений лица с помощью постепенного увеличения числа компонент

Качество обратно преобразованных данных аналогично качеству, полученному с помощью PCA, но немного хуже. Это вполно ожидаемо, поскольку PCA находит оптимальные направления с точки зрения реконструкции данных. NMF же, как правило, используется не из-за своей способности реконструировать или представлять данные, а скорее из-за того, что позволяет находить интересные закономерности в данных.

Для начала давайте попробуем извлечь лишь несколько компонент (скажем, 15). Рис. 3.15 показывает результат:

In[36]:

from sklearn.decomposition import NMF

nmf = NMF(n\_components=15, random\_state=0) nmf.fit(X\_train)

X\_train\_nmf = nmf.transform(X\_train) X\_test\_nmf = nmf.transform(X\_test)

fix, axes = plt.subplots(3, 5, figsize=(15, 12),

subplot\_kw={'xticks': (), 'yticks': ()})

for i, (component, ax) in enumerate(zip(nmf.components\_, axes.ravel())): ax.imshow(component.reshape(image\_shape))

ax.set\_title("{}. component".format(i))



**Рис. 3.15** Компоненты, найденные NMF для набора лиц

(использовалось 15 компонент)

Все эти компоненты являются положительными и поэтому похожи на прототипы лиц гораздо больше, чем компоненты PCA, показанные на рис. 3.9. Например, четко видно, что компонента 3 показывает лицо, немного повернутое вправо, тогда как компонента 7 показывает лицо, немного повернутое влево. Давайте посмотрим на изображения, для которых эти компоненты имеют наибольшие значения (показаны на рис. 3.16 и 3.17):

In[37]:

compn = 3

*# сортируем по 3-й компоненте, выводим первые 10 изображений*

inds = np.argsort(X\_train\_nmf[:, compn])[::-1] fig, axes = plt.subplots(2, 5, figsize=(15, 8),

subplot\_kw={'xticks': (), 'yticks': ()}) for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):

ax.imshow(X\_train[ind].reshape(image\_shape))

compn = 7

*# сортируем по 7-й компоненте, выводим первые 10 изображений*

inds = np.argsort(X\_train\_nmf[:, compn])[::-1] fig, axes = plt.subplots(2, 5, figsize=(15, 8),

subplot\_kw={'xticks': (), 'yticks': ()}) for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):

ax.imshow(X\_train[ind].reshape(image\_shape))



**Рис. 3.16** Лица с большим коэффициентом компоненты 3



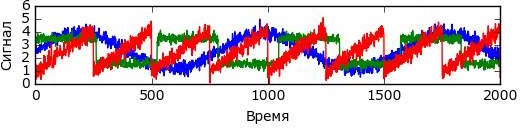
**Рис. 3.17** Лица с большим коэффициентом компоненты 7

Как и следовало ожидать, лица с высоким коэффициентом компоненты 3 – это лица, смотрящие вправо (рис. 3.16), тогда как лица с высоким коэффициентом компоненты 7 смотрят влево (рис. 3.17). Как уже упоминалось ранее, выделение паттернов, аналогичных рассматриваемым изображениям, лучше всего работает в отношении данных с аддитивной структурой, включая аудиоданные, данные экспрессии генов и текстовые данные. Давайте рассмотрим еще один пример на основе синтетических данных, чтобы увидеть, как это будет выглядеть.

Допустим, нас интересует сигнал, который представляет собой комбинацию трех различных источников (рис. 3.18):

In[38]:

S = mglearn.datasets.make\_signals() plt.figure(figsize=(6, 1)) plt.plot(S, '-') plt.xlabel("Время") plt.ylabel("Сигнал")



**Рис. 3.18** Исходные источники сигнала

К сожалению, мы не можем наблюдать исходные сигналы, лишь аддитивную смесь (сумму) всех трех сигналов. Необходимо восстановить

исходные компоненты из этой смеси. Предположим, у нас есть различные способы фиксировать характеристики этого смешанного сигнала (скажем, у нас есть 100 измерительных приборов), каждый из которых дает нам серию измерений:

In[39]:

A = np.random.RandomState(0).uniform(size=(100, 3)) X = np.dot(S, A.T)

print("Форма измерений: {}".format(X.shape))

Out[39]:

Форма измерений: (2000, 100)

Мы можем использовать NMF, чтобы восстановить три сигнала:

In[40]:

nmf = NMF(n\_components=3, random\_state=42) S\_ = nmf.fit\_transform(X)

print("Форма восстановленного сигнала: {}".format(S\_.shape))

Out[40]:

Форма восстановленного сигнала: (2000, 3)

Для сравнения мы еще применим PCA:

In[41]:

pca = PCA(n\_components=3) H = pca.fit\_transform(X)

Рис. 3.19 показывает активность сигнала, обнаруженную с помощью

NMF и PCA:

In[42]:

models = [X, S, S\_, H]

names = ['Наблюдения (первые три измерения)', 'Фактические источники',

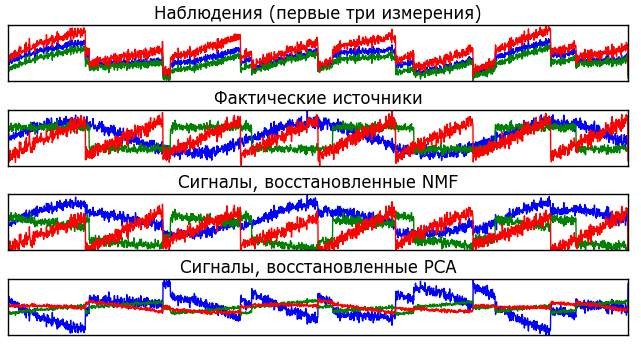
'Сигналы, восстановленные NMF', 'Сигналы, восстановленные PCA']

fig, axes = plt.subplots(4, figsize=(8, 4), gridspec\_kw={'hspace': .5},

subplot\_kw={'xticks': (), 'yticks': ()})

for model, name, ax in zip(models, names, axes): ax.set\_title(name)

ax.plot(model[:, :3], '-')



**Рис. 3.19** Восстановление первоначальных источников с помощью NMF и PCA

Этот график включает в себя наблюдения по первым 3 измерениям X. Как вы можете увидеть, NMF довольно хорошо выделил первоначальные источники, тогда как PCA потерпел неудачу и использовал первую компоненту, чтобы объяснить б*о*льшую часть дисперсии данных. Помните о том, что компоненты, полученные с помощью NMF, не упорядочены. В этом примере порядок компонент NMF точно такой же,

как в исходном сигнале (см. цвет трех кривых), но это носит чисто случайный характер.

Существует множество других алгоритмов, которые можно использовать для разложения каждой точки данных на взвешенную сумму компонент, как это делают PCA и NMF. Обсуждение всех этих алгоритмов выходит за рамки этой книги, а описание ограничений, накладываемых на компоненты и коэффициенты, часто предполагает знание теории вероятностей. Если вас заинтересовал тот или иной алгоритм выделения паттернов, мы рекомендуем вам изучить разделы

руководства scikit-learn, посвященные *анализу независимых компонент* (*independent component analysis*, *ICA*), *факторному анализу* (*factor analysis*, *FA*) и *разреженному кодированию* (*sparse coding*) с *обучением словаря* (*dictionary learning*). Информацию обо всех этих методах можно найти на странице, посвященной [декомпозиционным](http://scikit-learn.org/stable/modules/decomposition.html) [методам](http://scikit-learn.org/stable/modules/decomposition.html).



Хотя PCA часто выступает в качестве приоритетного метода, преобразующего данные таким образом, что можно визуализировать их с помощью диаграммы рассеяния, сам характер метода (вращение данных, а затем удаление направлений, объясняющих незначительную дисперсию данных) ограничивает его полезность, как мы уже убедились на примере диаграммы рассеяния для набора данных Labeled Faces in the Wild. Существует класс алгоритмов визуализации, называемых *алгоритмами множественного обучения* (*manifold learning algorithms*), которые используют гораздо более сложные графические представления данных и позволяют получить визуализации лучшего качества. Особенно полезным является алгоритм t-SNE.

Алгоритмы множественного обучения в основном направлены на визуализацию и поэтому редко используются для получения более двух новых характеристик. Некоторые из них, в том числе t-SNE, создают новое представление обучающих данных, но при этом не осуществляют преобразования новых данных. Это означает, что данные алгоритмы нельзя применить к тестовому набору, они могут преобразовать лишь те данные, на которых они были обучены. Множественное обучение может использоваться для разведочного анализа данных, но редко используется в тех случаях, когда конечной целью является применение модели машинного обучения с учителем. Идея, лежащая в основе алгоритма t- SNE, заключается в том, чтобы найти двумерное представление данных, сохраняющее расстояния между точками наилучшим образом. t-SNE начинает свою работу со случайного двумерного представления каждой точки данных, а затем пытается сблизить точки, которые в пространстве исходных признаков находятся близко друг к другу, и отдаляет друг от друга точки, которые находятся далеко друг от друга. При этом t-SNE уделяет большее внимание сохранению расстояний между точками, близко расположенными друг к другу. Иными словами, он пытается сохранить информацию, указывающую на то, какие точки являются соседями друг другу.

Мы применим алгоритм множественного обучения t-SNE к набору данных рукописных цифр, который включен в scikit-learn.24 Каждая точка данных в этом наборе является изображением цифры в градациях серого. Рис. 3.20 показывает примеры изображений для каждого класса:

24 Не следует путать с гораздо большим набором данных MNIST.

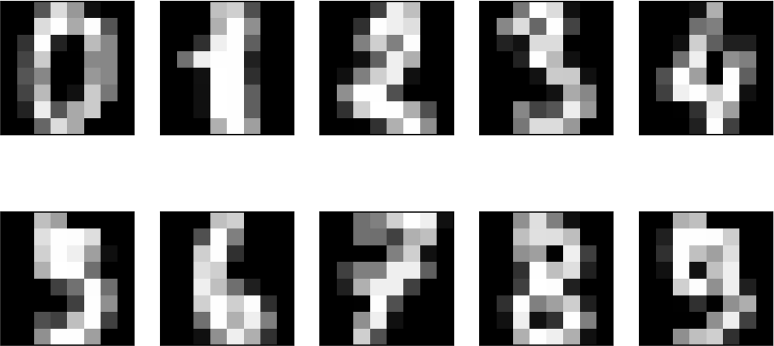
In[43]:

from sklearn.datasets import load\_digits digits = load\_digits()

fig, axes = plt.subplots(2, 5, figsize=(10, 5),

subplot\_kw={'xticks':(), 'yticks': ()}) for ax, img in zip(axes.ravel(), digits.images):

ax.imshow(img)



**Рис. 3.20** Примеры изображений из набора данных digits

Давайте используем PCA для визуализации данных, сведя их к двум измерениям. Мы построим график первых двух главных компонент и отметим цветом класс каждой точки (см. рис. 3-21):

In[44]:

*# строим модель PCA*

pca = PCA(n\_components=2) pca.fit(digits.data)

*# преобразуем данные рукописных цифр к первым двум компонентам*

digits\_pca = pca.transform(digits.data)

colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525", "#A83683", "#4E655E", "#853541", "#3A3120", "#535D8E"]

plt.figure(figsize=(10, 10))

plt.xlim(digits\_pca[:, 0].min(), digits\_pca[:, 0].max())

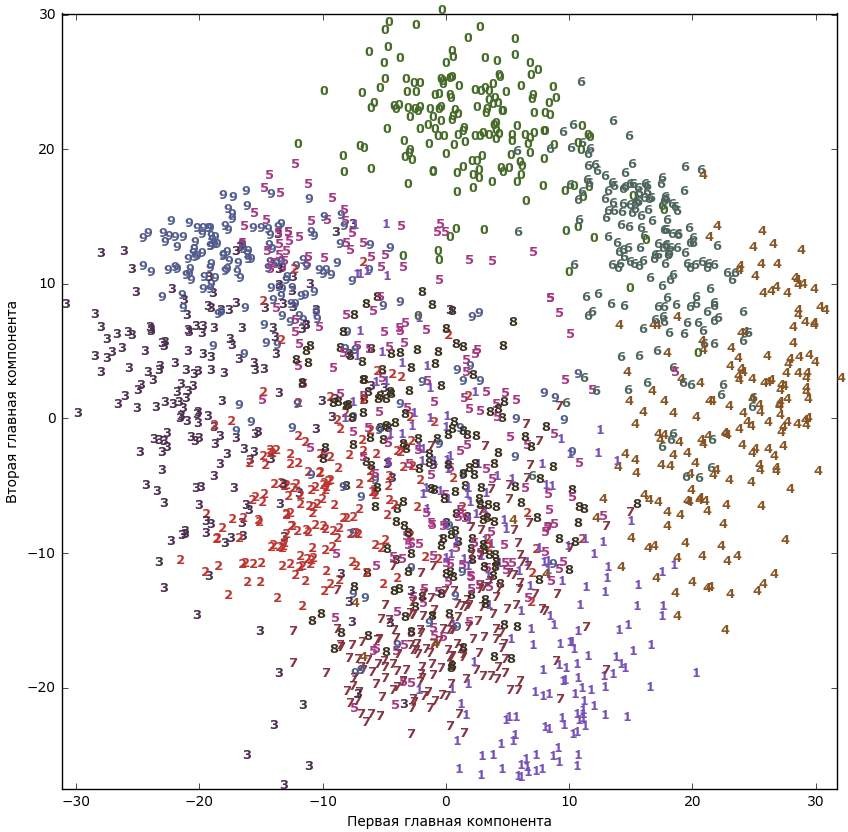
plt.ylim(digits\_pca[:, 1].min(), digits\_pca[:, 1].max()) for i in range(len(digits.data)):

*# строим график, где цифры представлены символами вместо точек*

plt.text(digits\_pca[i, 0], digits\_pca[i, 1], str(digits.target[i]), color = colors[digits.target[i]],

fontdict={'weight': 'bold', 'size': 9}) plt.xlabel("Первая главная компонента") plt.ylabel("Вторая главная компонента")

Здесь мы вывели фактические классы цифр в виде символов, чтобы визуально показать расположение каждого класса. Цифры 0, 6 и 4 относительно хорошо разделены с помощью первых двух главных компонент, хотя по-прежнему перекрывают друг друга. Большинство остальных цифр значительно перекрывают друг друга.



**Рис. 3.21** Диаграмма рассеяния для набора данных digits, использующая первые две главные компоненты

Давайте применим t-SNE к этому же набору данных и сравним результаты. Поскольку t-SNE не поддерживает преобразование новых данных, в классе TSNE нет метода transform. Вместо этого мы можем вызвать метод fit\_transform, который построит модель и немедленно вернет преобразованные данные (см. рис. 3.22):

In[45]:

from sklearn.manifold import TSNE tsne = TSNE(random\_state=42)

*# используем метод fit\_transform вместо fit, т.к. класс TSNE не использует метод transform*

digits\_tsne = tsne.fit\_transform(digits.data)

In[46]:

plt.figure(figsize=(10, 10))

plt.xlim(digits\_tsne[:, 0].min(), digits\_tsne[:, 0].max() + 1)

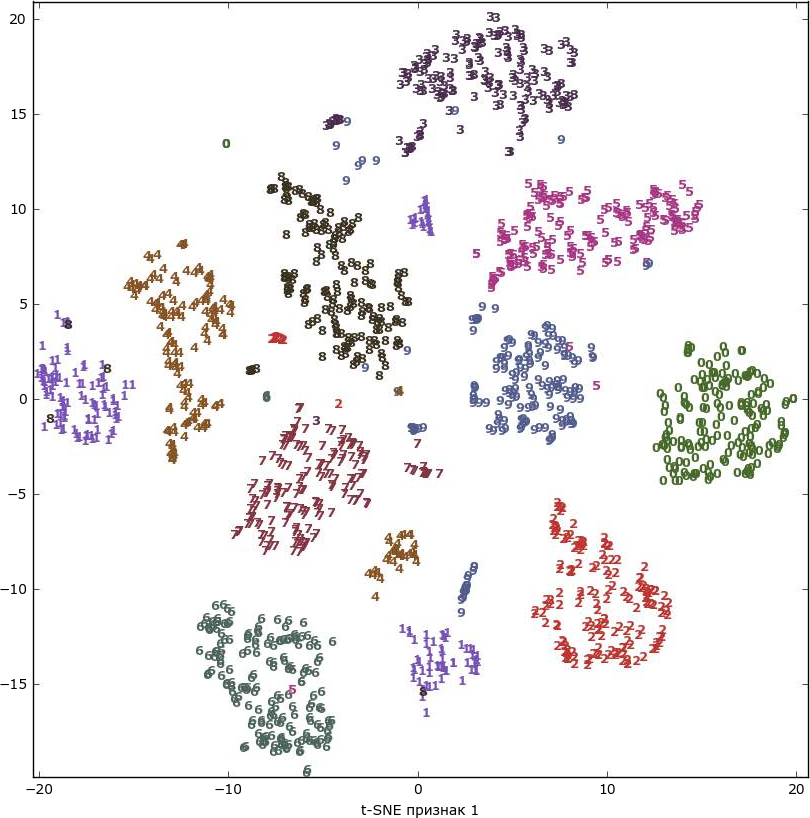
plt.ylim(digits\_tsne[:, 1].min(), digits\_tsne[:, 1].max() + 1) for i in range(len(digits.data)):

*# строим график, где цифры представлены символами вместо точек*

plt.text(digits\_tsne[i, 0], digits\_tsne[i, 1], str(digits.target[i]), color = colors[digits.target[i]],

fontdict={'weight': 'bold', 'size': 9}) plt.xlabel("t-SNE признак 0")

plt.xlabel("t-SNE признак 1")



**Рис. 3.22** Диаграмма рассеяния для набора данных digits, которая использует первые две главные компоненты, найденные с помощью t-SNE

Результат, полученный с помощью t-SNE, весьма примечателен. Все классы довольно четко разделены. Единицы и девятки в некоторой степени распались, однако большинство классов образуют отдельные сплоченные группы. Имейте в виду, что этот метод не использует

информацию о метках классов: он является полностью неконтролируемым. Тем не менее он может найти двумерное представление данных, которое четко разграничивает классы, используя лишь информацию о расстояниях между точками данных в исходном пространстве.

Алгоритм t-SNE имеет некоторые настраиваемые параметры, хотя, как правило, дает хорошее качество, когда используются настройки по умолчанию. Вы можете поэкспериментировать с параметрами perplexity и early\_exaggeration, но эффекты от их применения обычно незначительны.

# **ЛБ 13 Кластеризация**

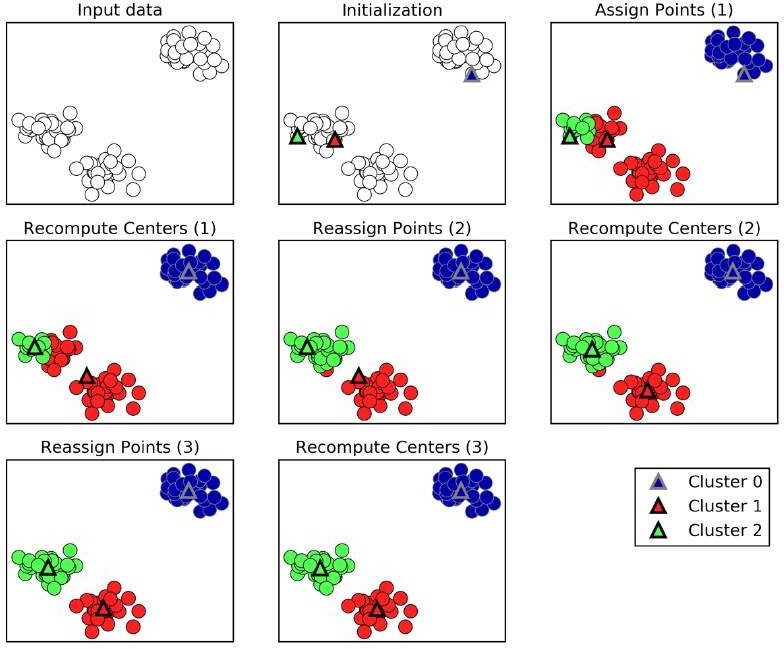
Как мы уже говорили выше, *кластеризация* (*clustering*) является задачей разбиения набора данных на группы, называемые кластерами. Цель – разделить данные таким образом, чтобы точки, находящие в одном и том же кластере, были очень схожи друг с другом, а точки, находящиеся в разных кластерах, отличались друг от друга. Как и алгоритмы классификации, алгоритмы кластеризации присваивают (или прогнозируют) каждой точке данных номер кластера, которому она принадлежит.



Кластеризация *k*-средних – один из самых простых и наиболее часто используемых алгоритмов кластеризации. Сначала выбирается число кластеров *k*. После выбора значения *k* алгоритм *k*-средних отбирает точки, которые будут представлять *центры кластеров* (*cluster centers*). Затем для каждой точки данных вычисляется его евклидово расстояние до каждого центра кластера. Каждая точка назначается ближайшему центру кластера. Алгоритм вычисляет *центроиды* (*centroids*) – центры тяжести кластеров. Каждый центроид – это вектор, элементы которого представляют собой средние значения характеристик, вычисленные по всем точкам кластера. Центр кластера смещается в его центроид. Точки заново назначаются ближайшему центру кластера. Этапы изменения центров кластеров и переназначения точек итеративно повторяются до тех пор, пока границы кластеров и расположение центроидов не перестанут изменяться, т.е. на каждой итерации в каждый кластер будут попадать одни и те же точки данных. Следующий пример (рис. 3.23) иллюстрирует работу алгоритма на синтетическом наборе данных

In[47]:

mglearn.plots.plot\_kmeans\_algorithm()



Входные данные

Инициализация

Назначение точек (1)

Пересчет центров (1)

Переназначение точек (2)

Пересчет центров (2)

Переназначение точек (3)

Пересчет центров (3)

Кластер 0

Кластер 1

Кластер 2

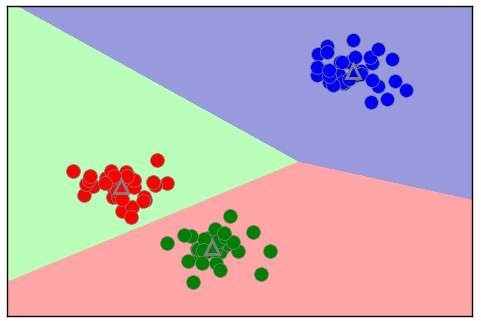
**Рис. 3.23** Исходные данные и этапы алгоритма *k*-средних

Центры кластеров представлены в виде треугольников, в то время как точки данных отображаются в виде окружностей. Цвета указывают принадлежность к кластеру. Мы указали, что ищем три кластера, поэтому алгоритм был инициализирован с помощью случайного выбора трех точек данных в качестве центров кластеров (см. «Инициализация»). Затем запускается итерационный алгоритм. Во-первых, каждая точка данных назначается ближайшему центру кластера (см. «Назначение точек (1)»). Затем центры кластеров переносятся в центры тяжести кластеров (см. «Пересчет центров (1)»). Затем процесс повторяется еще два раза. После третьей итерации принадлежность точек кластерным центрам не изменилась, поэтому алгоритм останавливается.

Получив новые точки данных, алгоритм k-средних будет присваивать каждую точку данных ближайшему центру кластера. Следующий пример (рис. 3.24) показывает границы центров кластеров, процесс вычисления которых был приведен на рис. 3.23:

In[48]:

mglearn.plots.plot\_kmeans\_boundaries()



**Рис. 3.24** Центры кластеров и границы кластеров, найденные с помощью алгоритма *k*-средних

Применить алгоритм *k*-средних, воспользовавшись библиотекой scikit-learn, довольно просто. Здесь мы применяем его к синтетическим данным, которые использовали для построения предыдущих графиков. Мы создаем экземпляр класса KMeans и задаем количество выделяемых кластеров.25 Затем мы вызываем метод fit и передаем ему в качестве аргумента данные:

In[49]:

from sklearn.datasets import make\_blobs from sklearn.cluster import KMeans

*# генерируем синтетические двумерные данные*

X, y = make\_blobs(random\_state=1)

*# строим модель кластеризации* kmeans = KMeans(n\_clusters=3) kmeans.fit(X)

Во время работы алгоритма каждой точке обучающих данных X присваивается метка кластера. Вы можете найти эти метки в атрибуте kmeans.labels\_:

25 Если вы не зададите количество выделяемых кластеров, то значение n\_clusters по умолчанию будет равно 8. При этом нет никаких конкретных причин, в силу которых вы должны использовать именно это значение.

In[50]:

print("Принадлежность к кластерам:\n{}".format(kmeans.labels\_))

|  |  |
| --- | --- |
| Out[50]:  Принадлежность | к кластерам: |
| [1 2 2 2 0 0 0 | 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2 |
| 2 2 0 0 2 1 2 | 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1 |
| 1 2 0 0 1 2 1 | 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1] |

Поскольку мы задали три кластера, кластеры пронумерованы от 0 до

2.

Кроме того, вы можете присвоить метки кластеров новым точкам с

помощью метода predict. В ходе прогнозирования каждая новая точка назначается ближайшему центру кластера, но существующая модель не меняется. Запуск метода predict на обучающем наборе возвращает тот же самый результат, что содержится в атрибуте labels\_:

In[51]:

print(kmeans.predict(X))

Out[51]:

[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2

2 2 0 0 2 1 2 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1

1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]

Вы можете увидеть, что кластеризация немного похожа на классификацию в том плане, что каждый элемент получает метку. Однако нет никаких оснований утверждать, что данная метка является истинной и поэтому сами по себе метки не несут никакого априорного смысла. Давайте вернемся к примеру с кластеризацией изображений лиц, который мы обсуждали ранее. Возможно, что кластер 3, найденный с помощью алгоритма, содержит лишь лица вашего друга. Впрочем, вы можете узнать это только после того, как взгляните на фотографии, а само число 3 является произвольным. Единственная информация, которую дает вам алгоритм, – это то, что все лица, отнесенные к кластеру 3, схожи между собой.

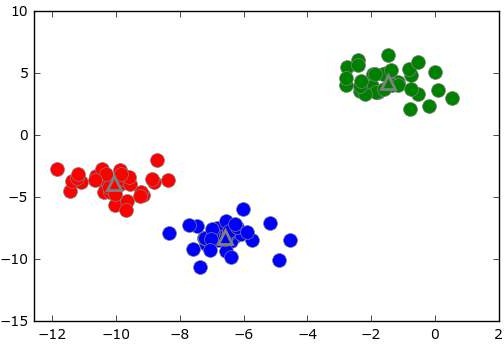
В случае с кластеризацией, которую мы только что построили для двумерного синтетического набора данных, это означает, что мы не должны придавать значения тому факту, что одной группе был присвоен 0, а другой – 1. Повторный запуск алгоритма может привести к совершенно иной нумерации кластеров в силу случайного характера инициализации.

Ниже приводится новый график для тех же самых данных (рис. 3.25). Центры кластеров записаны в атрибуте cluster\_centers\_ и мы наносим их на график в виде треугольников:

In[52]:

mglearn.discrete\_scatter(X[:, 0], X[:, 1], kmeans.labels\_, markers='o') mglearn.discrete\_scatter(

kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], [0, 1, 2], markers='^', markeredgewidth=2)



**Рис. 3.25** Принадлежность к кластерам и центры кластеров, найденные с помощью алгоритма *k*-средних, *k*=3

Кроме того, мы можем увеличить или уменьшить количество центров кластеров (рис. 3.26):

In[53]:

fig, axes = plt.subplots(1, 2, figsize=(10, 5))

*# использование двух центров кластеров:*

kmeans = KMeans(n\_clusters=2) kmeans.fit(X)

assignments = kmeans.labels\_

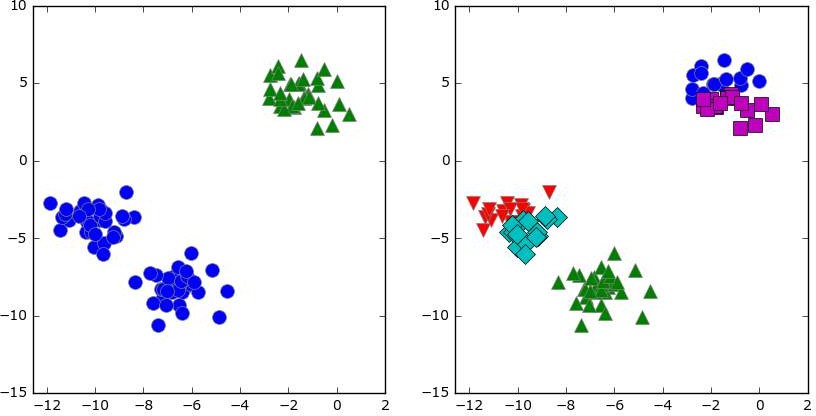
mglearn.discrete\_scatter(X[:, 0], X[:, 1], assignments, ax=axes[0])

*# использование пяти центров кластеров:*

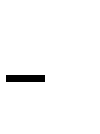
kmeans = KMeans(n\_clusters=5) kmeans.fit(X)

assignments = kmeans.labels\_

mglearn.discrete\_scatter(X[:, 0], X[:, 1], assignments, ax=axes[1])



**Рис. 3.26** Принадлежность к кластерам, найденная с помощью алгоритма *k*-средних, *k*=3 (слева) и *k*=5 (справа)



Даже если вы знаете «правильное» количество кластеров для

конкретного набора данных, алгоритм *k*-средних не всегда может выделить их. Каждый кластер определяется исключительно его центром, это означает, что каждый кластер имеет выпуклую форму. В результате этого алгоритм *k*-средних может описать относительно простые формы. Кроме того, алгоритм *k*-средних предполагает, что все кластеры в определенном смысле имеют одинаковый «диаметр», он всегда проводит границу между кластерами так, чтобы она проходила точно посередине между центрами кластеров. Это иногда может привести к неожиданным результатам, как показано на рис. 3.27:

In[54]:

X\_varied, y\_varied = make\_blobs(n\_samples=200,

cluster\_std=[1.0, 2.5, 0.5], random\_state=170)

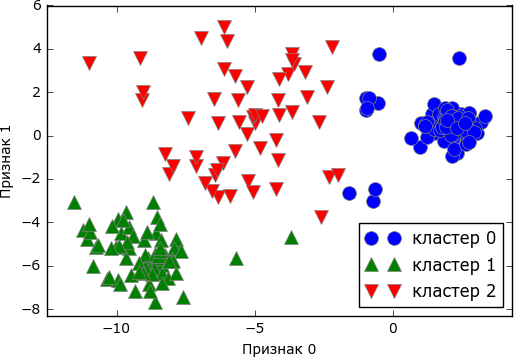
y\_pred = KMeans(n\_clusters=3, random\_state=0).fit\_predict(X\_varied)

mglearn.discrete\_scatter(X\_varied[:, 0], X\_varied[:, 1], y\_pred)

plt.legend(["кластер 0", "кластер 1", "кластер 2"], loc='best')

plt.xlabel("Признак 0")

plt.ylabel("Признак 1")



**Рис. 3.27** Принадлежность к кластерам, найденная с помощью алгоритма

*k*-средних, при этом кластеры имеют разные плотности

Можно было бы ожидать плотную область в нижнем левом углу, которая рассматривалась бы в качестве первого кластера, плотную область в верхнем правом углу в качестве второго кластера и менее плотную область в центре в качестве третьего кластера. Вместо этого, у кластера 0 и кластера 1 есть несколько точек, которые сильно удалены от всех остальных точек этих кластеров, «тянущихся» к центру.

Кроме того, алгоритм *k*-средних предполагает, что все направления одинаково важны для каждого кластера. Следующий график (рис. 3.28) показывает двумерный набор данных с тремя четко обособленными группами данных. Однако эти группы вытянуты по диагонали. Поскольку алгоритм *k*-средних учитывает лишь расстояние до ближайшего центра кластера, он не может обработать данные такого рода:

In[55]:

*# генерируем случайным образом данные для кластеризации*

X, y = make\_blobs(random\_state=170, n\_samples=600) rng = np.random.RandomState(74)

*# преобразуем данные так, чтобы они были вытянуты по диагонали*

transformation = rng.normal(size=(2, 2)) X = np.dot(X, transformation)

*# группируем данные в три кластера* kmeans = KMeans(n\_clusters=3) kmeans.fit(X)

y\_pred = kmeans.predict(X)

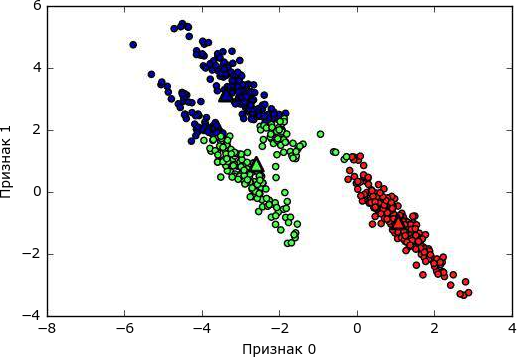
*# строим график принадлежности к кластерам и центров кластеров*

plt.scatter(X[:, 0], X[:, 1], c=y\_pred, cmap=mglearn.cm3)

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], marker='^', c=[0, 1, 2], s=100, linewidth=2, cmap=mglearn.cm3)

plt.xlabel("Признак 0")

plt.ylabel("Признак 1")



**Рис. 3.28** Алгоритм *k*-средних не позволяет выявить несферические кластеры

Кроме того, алгоритм *k*-средних плохо работает, когда кластеры имеют более сложную форму, как в случае с данными two\_moons, с которыми мы столкнулись в главе 2 (см. рис. 3.29):

In[56]:

*# генерируем синтетические данные two\_moons (на этот раз с меньшим количеством шума)*

from sklearn.datasets import make\_moons

X, y = make\_moons(n\_samples=200, noise=0.05, random\_state=0)

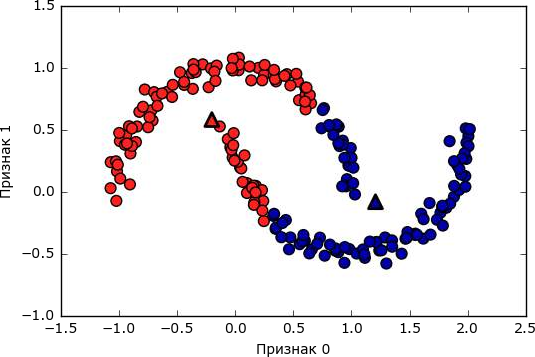
*# группируем данные в два кластера* kmeans = KMeans(n\_clusters=2) kmeans.fit(X)

y\_pred = kmeans.predict(X)

*# строим график принадлежности к кластерам и центров кластеров* plt.scatter(X[:, 0], X[:, 1], c=y\_pred, cmap=mglearn.cm2, s=60) plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1],

marker='^', c=[mglearn.cm2(0), mglearn.cm2(1)], s=100, linewidth=2) plt.xlabel("Признак 0")

plt.ylabel("Признак 1")



**Рис. 3.29** Алгоритм *k*-средних не позволяет выявить кластеры более сложной формы

В данном случае мы понадеялись на то, что алгоритм кластеризации сможет обнаружить два кластера в форме полумесяцев. Однако определить их с помощью алгоритма *k*-средних не представляется возможным.