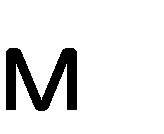
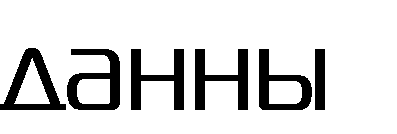
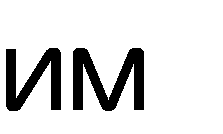
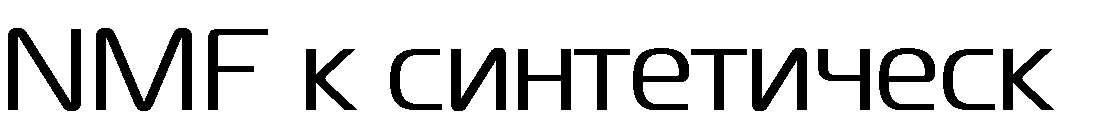
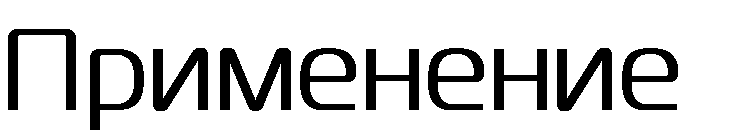
# **ЛБ 13 Факторизация неотрицательных матриц**

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

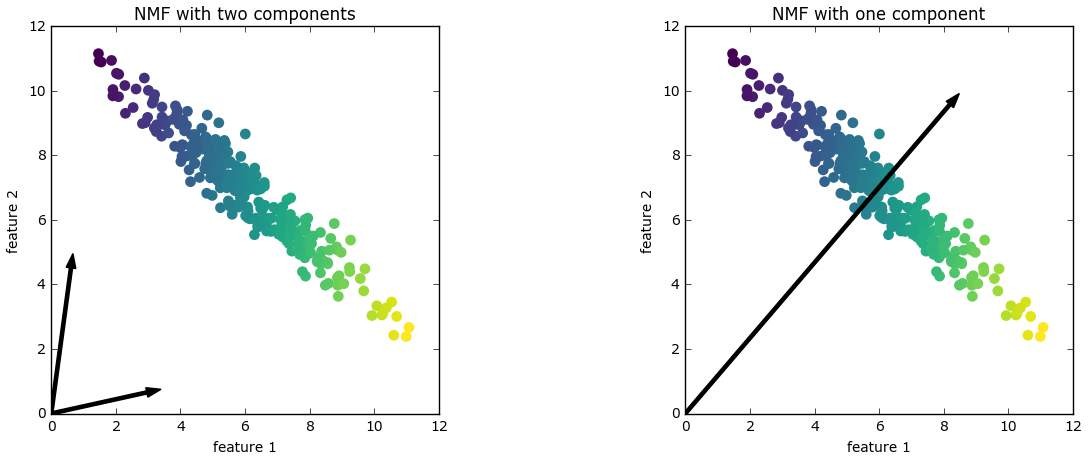
убедиться, что они имеют положительные значения. Это означает, что для NMF расположение данных относительно начала координат (0, 0) имеет реальное значение. Поэтому извлекаемые неотрицательные компоненты можно представить в виде направлений, выходящих из начала координат (0, 0) к данным.

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

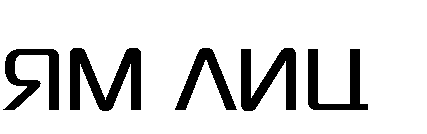
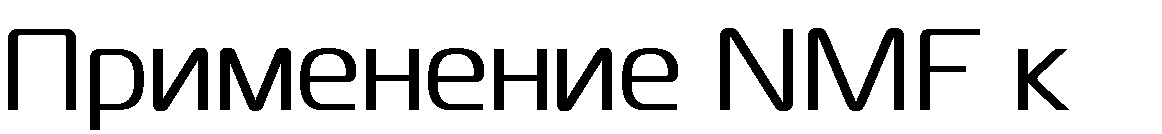
**Рис. 3.13** Компоненты, найденные в результате факторизации неотрицательных матриц с двумя компонентами (слева) и

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



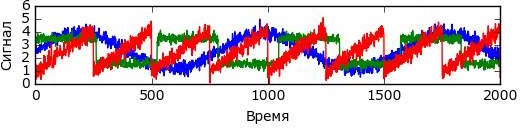
**Рис. 3.17** Лица с большим коэффициентом компоненты 7

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',

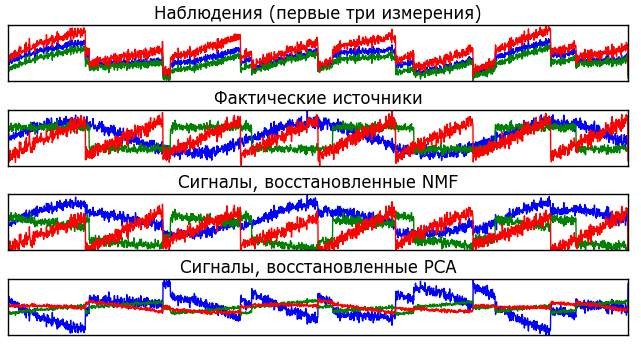
'Сигналы, восстановленные NMF', 'Сигналы, восстановленные PCA']

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:

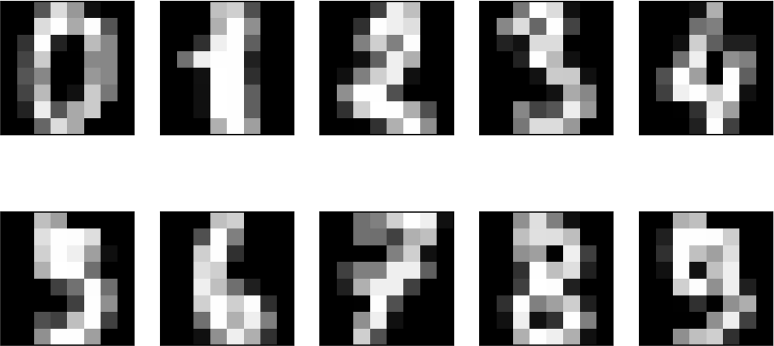
In[43]:

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())

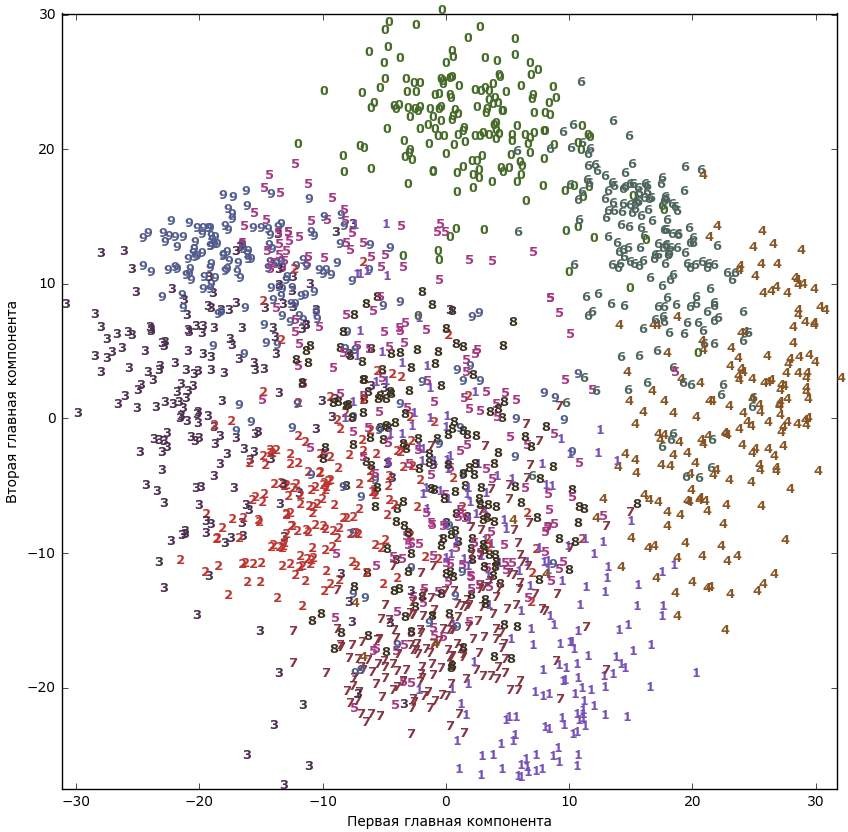
plt.ylim(digits\_pca[:, 1].min(), digits\_pca[:, 1].max()) for i in range(len(digits.data)):

*# 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)

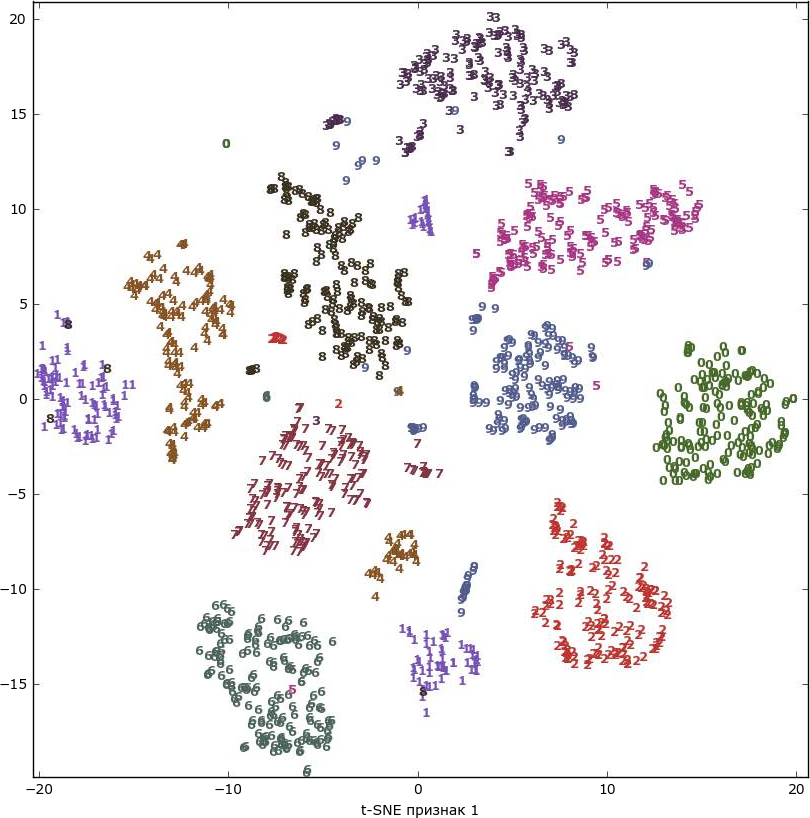
plt.ylim(digits\_tsne[:, 1].min(), digits\_tsne[:, 1].max() + 1) for i in range(len(digits.data)):

*# 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**

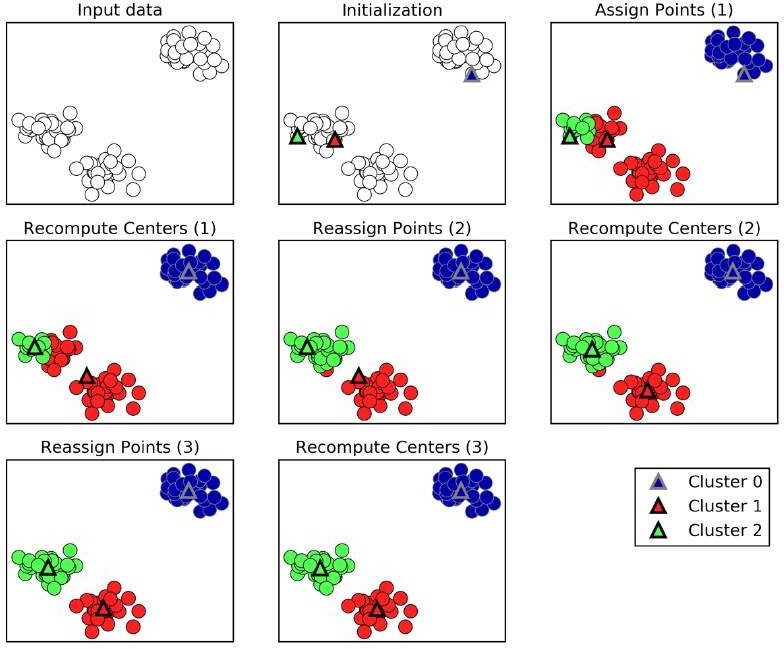
Как мы уже говорили выше, *кластеризация* (*clustering*) является задачей разбиения набора данных на группы, называемые кластерами. Цель – разделить данные таким образом, чтобы точки, находящие в одном и том же кластере, были очень схожи друг с другом, а точки, находящиеся в разных кластерах, отличались 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

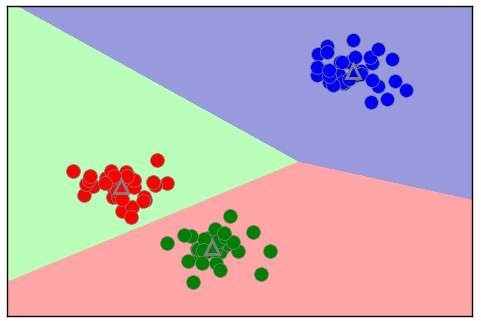
**Рис. 3.23** Исходные данные и этапы алгоритма *k*-средних

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\_:

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.

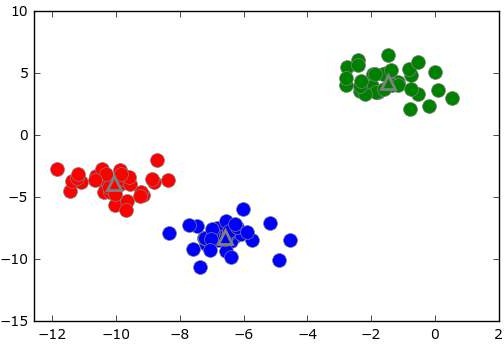
В случае с кластеризацией, которую мы только что построили для двумерного синтетического набора данных, это означает, что мы не должны придавать значения тому факту, что одной группе был присвоен 0, а другой – 1. Повторный запуск алгоритма может привести к совершенно иной нумерации кластеров в силу случайного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

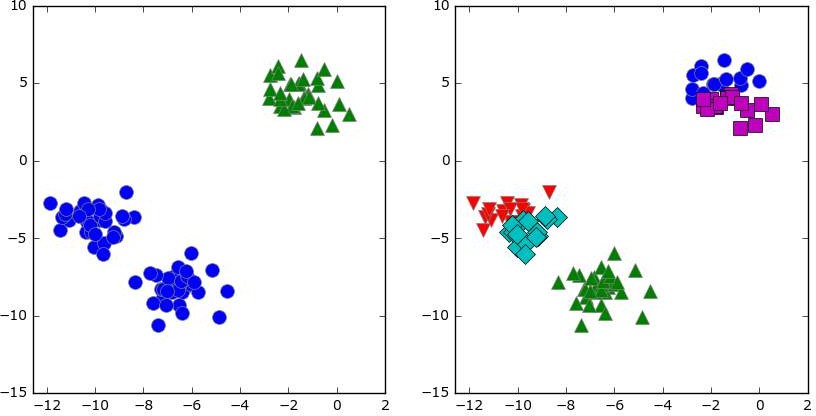
mglearn.discrete\_scatter(X[:, 0], X[:, 1], assignments, ax=axes[0])

*# использование пяти центров кластеров:*

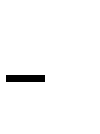
kmeans = KMeans(n\_clusters=5) kmeans.fit(X)

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)

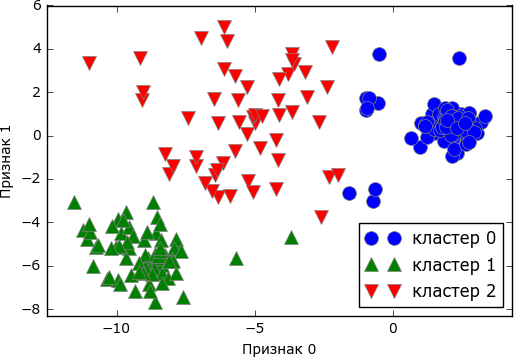
y\_pred = KMeans(n\_clusters=3, random\_state=0).fit\_predict(X\_varied)

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)

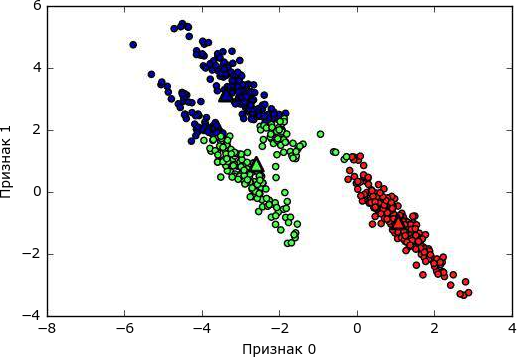
*# строим график принадлежности к кластерам и центров кластеров*

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)

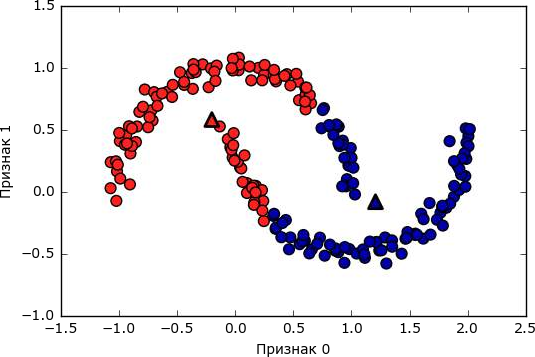
*# группируем данные в два кластера* kmeans = KMeans(n\_clusters=2) kmeans.fit(X)

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.