**Лаб.2. Заглянем глубже: кластеризация методом k-средних**

В нескольких предыдущих разделах мы рассматривали только одну разновидность машинного обучения без учителя — понижение размерности. В этом разделе мы перейдем к другому классу моделей машинного обучения без учителя — алгоритмам кластеризации. Алгоритмы кластеризации нацелены на то, чтобы найти, исходя из свойств данных, получить оптимальное разбиение или дискретную маркировку групп точек.

В библиотеке Scikit-Learn и других местах имеется множество алгоритмов кластеризации, но, вероятно, наиболее простой для понимания — алгоритм кластеризации *методом k-средних* (k-means clustering), реализованный в классе sklearn.cluster.KMeans . Начнем с обычных импортов:

In[1]: %matplotlib inline

import matplotlib.pyplot as plt

import seaborn as sns; sns.set() # для стилизации графиков

import numpy as np

**Знакомство с методом k-средних**

Алгоритм *k*-средних выполняет поиск заранее заданного количества кластеров в немаркированном многомерном наборе данных. Достигается это с помощью простого представления о том, что такое оптимальная кластеризация.

* «Центр кластера» — арифметическое среднее всех точек, относящихся к этому кластеру.
* Каждая точка ближе к центру своего кластера, чем к центрам других кластеров.

Эти два допущения составляют основу модели метода *k*-средних. Далее мы рассмотрим детальнее, *каким именно образом* алгоритм находит это решение, а пока возьмем простой набор данных и посмотрим на результаты работы метода *k*-средних для него.

Во-первых, сгенерируем двумерный набор данных, содержащий четыре отдельных «пятна». Чтобы подчеркнуть отсутствие учителя в этом алгоритме, мы не будем включать метки в визуализацию (рис. 2.1):

In[2]: from sklearn.d atasets.samples\_generator import make\_blobs

X, y\_true = make\_blobs(n\_samples=300, centers=4,

cluster\_std=0.60, random\_state=0)

plt.scatter(X[:, 0], X[:, 1], s=50);

Визуально выделить здесь четыре кластера не представляет труда. Алгоритм *k*-средних делает это автоматически, используя в библиотеке Scikit-Learn API статистических оценок:

In[3]: from sklearn.cluster import KMeans

kmeans = KMeans(n\_clusters=4)

kmeans.fit(X)

y\_kmeans = kmeans.predict(X)

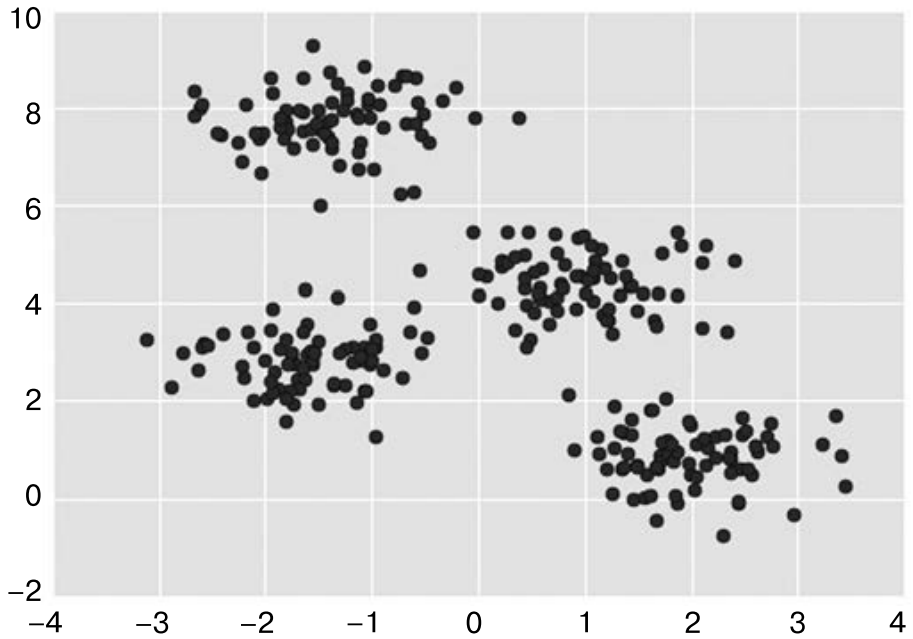


Рис. 2.1. Данные для демонстрации кластеризации

Визуализируем результаты, выведя на график окрашенные в соответствии с метками данные. Нарисуем найденные оценивателем метода *k*-средних центры кластеров (рис. 2.2):

In[4]: plt.scatter(X[:, 0], X[:, 1], c=y\_kmeans, s=50, cmap='viridis')

centers = kmeans.cluster\_centers\_

plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);

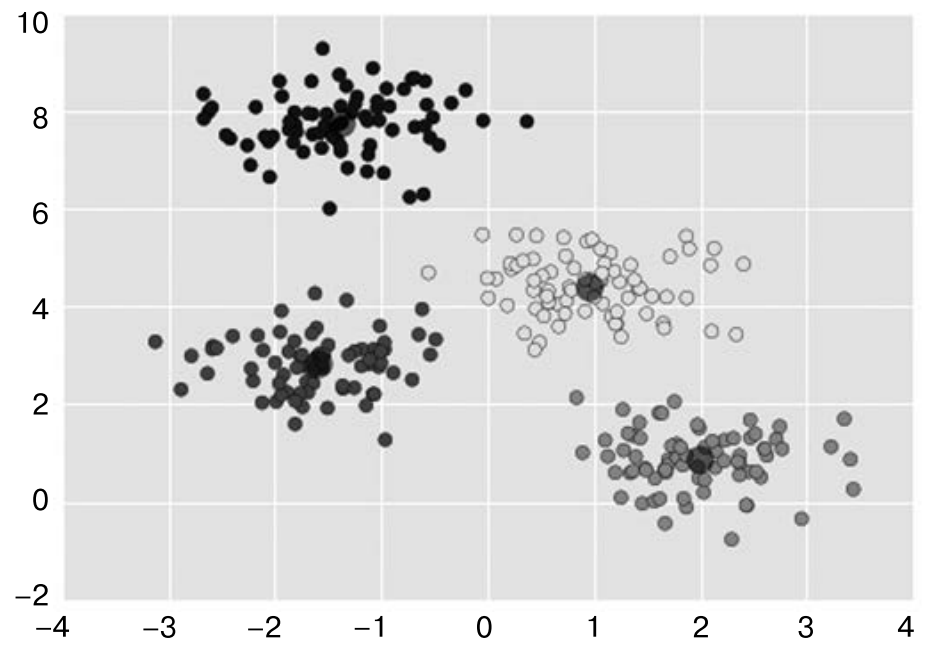


Рис. 2.2. Центры кластеров метода k-средних с окрашенными в разные цвета кластерами

Хорошая новость состоит в том, что алгоритм k-средних (по крайней мере в этом простом случае) задает соответствие точек кластерам очень схоже с тем, как мы бы могли сделать это визуально. Но у вас может возникнуть вопрос: как этому алгоритму­ удалось­ найти кластеры так быстро? В конце концов, количество возможных сочетаний точек и кластеров зависит экспоненциально от числа точек данных, так что поиск методом полного перебора оказался бы чрезвычайно ресурсозатратным. Поиск методом полного перебора здесь не требуется. Вместо него в типичном варианте метода k-средних применяется интуитивно понятный подход, известный под названием *«максимизация математического ожидания»* (expectation-maximization, EM).

**Алгоритм k-средних: максимизация математического ожидания**

Максимизация математического ожидания (EM) — мощный алгоритм, встречающийся во множестве контекстов науки о данных. Метод *k*-средних — особенно простое и понятное приложение этого алгоритма, и мы рассмотрим его здесь вкратце. Подход максимизации математического ожидания состоит из следующей процедуры.

1. Выдвигаем гипотезу о центрах кластеров.
2. Повторяем до достижения сходимости:

* *E-шаг*: приписываем точки к ближайшим центрам кластеров;
* *M-шаг*: задаем новые центры кластеров в соответствии со средними значениями.

E-шаг, или шаг ожидания (expectation), назван так потому, что включает актуализацию математического ожидания того, к каким кластерам относятся точки.

M-шаг, или шаг максимизации (maximization), назван так потому, что включает максимизацию некоторой целевой функции, описывающей расположения центров кластеров. В таком случае максимизация достигается путем простого усреднения данных в кластере.

Этому алгоритму посвящена обширная литература, но если коротко, то можно подвести его итоги следующим образом: при обычных обстоятельствах каждая итерация шагов E и M всегда будет приводить к улучшению оценки показателей кластера. Визуализировать этот алгоритм можно так, как показано на рис. 2.3.

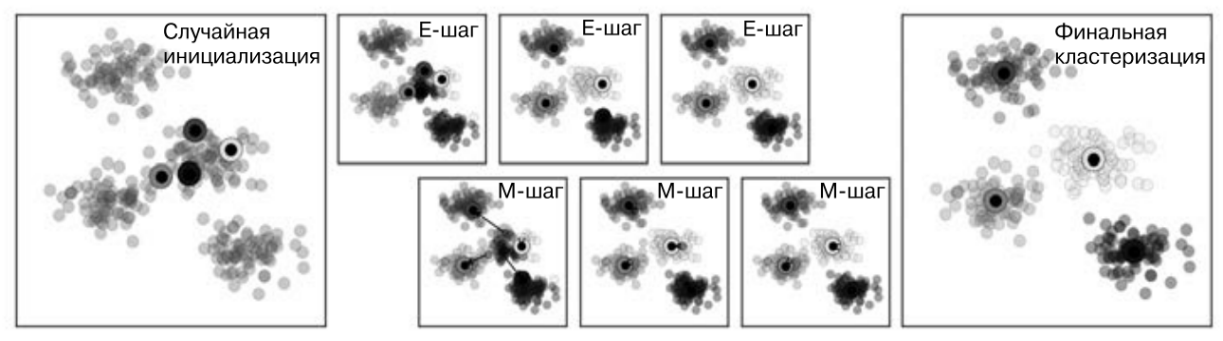


Рис. 2.3. Визуализация EM-алгоритма для метода k-средних

Для показанных на рис. 2.3 конкретных начальных значений кластеры сходятся всего за три итерации. Интерактивную версию рисунка можно увидеть в онлайн-приложении (https://github.com/jakevdp/PythonDataScienceHandbook).

Алгоритм *k*-средних достаточно прост для того, чтобы уместиться в нескольких

строках кода. Вот простейшая его реализация (рис. 2.4):

In[5]: from sklearn.metrics import pairwise\_distances\_argmin

def find\_clusters(X, n\_clusters, rseed=2):

# 1. Выбираем кластеры случайным образом

rng = np.random.RandomState(rseed)

i = rng.permutation(X.shape[0])[:n\_clusters]

centers = X[i]

while True:

# 2a. Присваиваем метки в соответствии с ближайшим центром

labels = pairwise\_distances\_argmin(X, centers)

# 2b. Находим новые центры, исходя из средних значений точек

new\_centers = np.array([X[labels == i].mean(0)

for i in range(n\_clusters)])

# 2c. Проверяем сходимость

if np.all(centers == new\_centers):

break

centers = new\_centers

return centers, labels

centers, labels = find\_clusters(X, 4)

plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis');

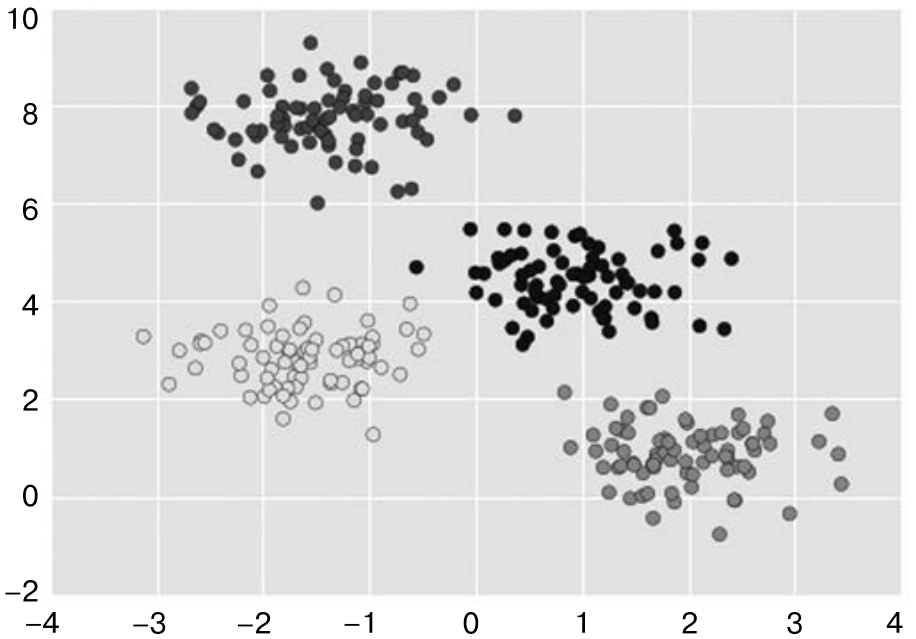


Рис. 2.4. Данные, маркированные с помощью метода *k*-средних

Самые проверенные реализации выполняют «под капотом» немного больше действий, но основное представление о методе максимизации математического

ожидания предыдущая функция дает.

**Предупреждения относительно метода максимизации математического ожи**

**дания.** При использовании алгоритма максимизации математического ожидания следует иметь в виду несколько нюансов.

* *Глобально оптимальный результат может оказаться недостижимым в принципе*. Во-первых, хотя процедура EM гарантированно улучшает результат на каждом шаге, уверенности в том, что она ведет к глобально наилучшему решению, нет. Например, если мы воспользуемся в нашей простой процедуре другим начальным значением для генератора случайных чисел, полученные начальные гипотезы приведут к неудачным результатам (рис. 2.5):

In[6]: centers, labels = find\_clusters(X, 4, rseed=0)

plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis');

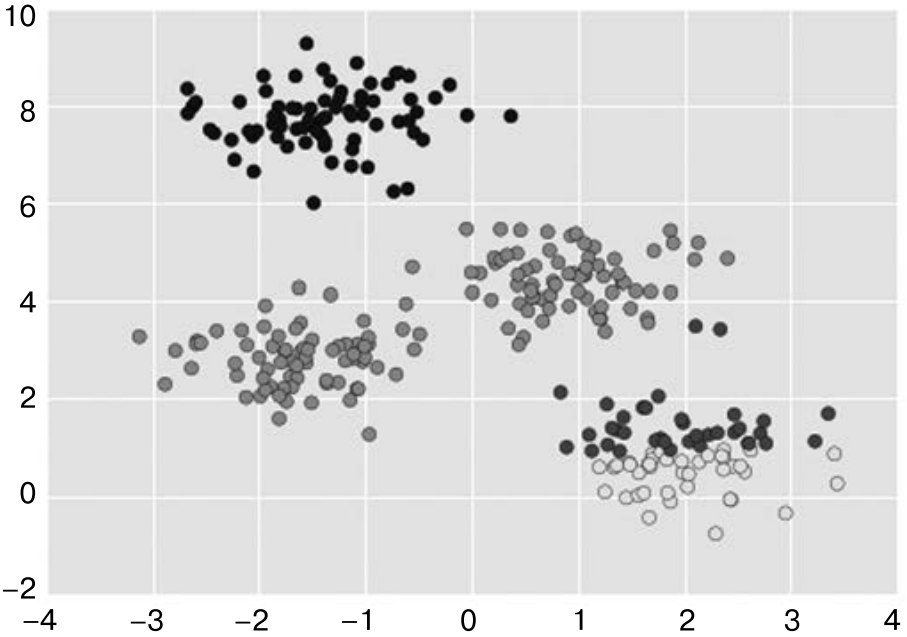


Рис. 2.5. Пример плохой сходимости в методе k-средних

В этом случае EM-метод сошелся к глобально неоптимальной конфигурации, поэтому его часто выполняют для нескольких начальных гипотез, что и делает по умолчанию библиотека Scikit-Learn (это задается с помощью параметра n\_init , по умолчанию имеющего значение 10 ).

* Количество кластеров следует выбирать заранее. Еще одна часто встречающаяся проблема с методом *k*-средних заключается в том, что ему необходимо сообщить, какое количество кластеров вы ожидаете: он не умеет вычислять количество кластеров на основе данных. Например, если предложить алгоритму выделить шесть кластеров, он с радостью это сделает и найдет шесть оптималь ных кластеров (рис. 2.6):

In[7]: labels = KMeans(6, random\_state=0).fit\_predict(X)

plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis');

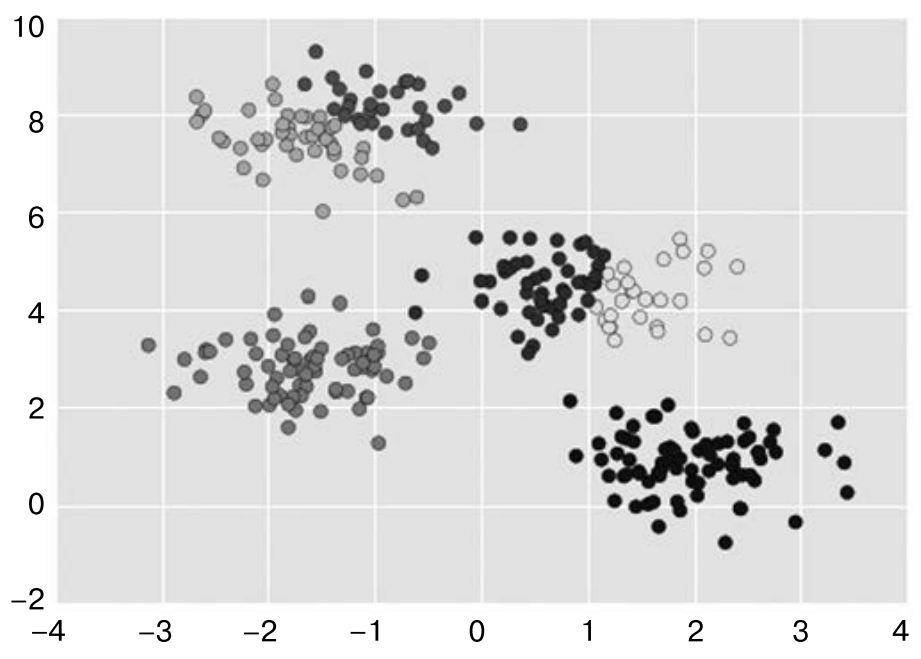


Рис. 2.6. Пример неудачного выбора количества кластеров

Осмысленный ли результат получен — сказать трудно. Один из полезных в этом случае и интуитивно довольно понятных подходов, который мы не станем обсуждать подробно, — силуэтный анализ (http://scikit-learn.org/stable/auto\_examples/cluster/plot\_kmeans\_silhouette\_analysis.html ).

В качестве альтернативы можно воспользоваться более сложным алго ритмом кластеризации с лучшим количественным показателем зависимости качества аппроксимации от количества кластеров (см. смесь Гауссовых распределений, в следующей работе «Заглянем глубже: смеси Гауссовых распределений») или с *возможностью* выбора приемлемого количества кластеров (например, методы DBSCAN, сдвиг среднего или распространения аффинности (affinity propagation), находящиеся в подмодуле sklearn.cluster ).

* Применение метода k-средних ограничивается случаем линейных границ кластеров. Базовое допущение модели k-средних (точки должны быть ближе к центру их собственного кластера, чем других) означает, что этот алгоритм зачастую будет неэффективен в случае сложной геометрии кластеров.

В частности, границы между кластерами в методе k-средних всегда будут линейными, а значит, он будет плохо работать в случае более сложных границ.

Рассмотрим следующие данные и найденные для них при обычном подходе

метода *k*-средних метки кластеров (рис. 2.7):

In[8]: from sklearn.datasets import make\_moons

X, y = make\_moons(200, noise=.05, random\_state=0)

In[9]: labels = KMeans(2, random\_state=0).fit\_predict(X)

plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis');

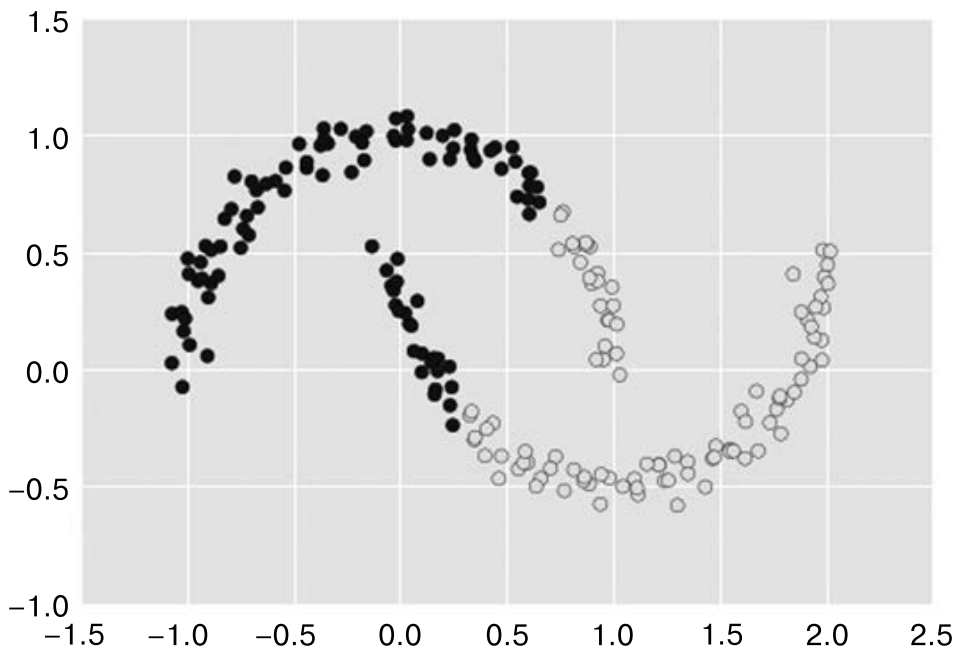


Рис. 2.7. Неудача метода k-средних в случае нелинейных границ

Здесь можно попробовать воспользоваться некоторой уловкой, называемой ядерное преобразование, чтобы метод *k*-средних стал распознавать нелинейные границы. Более подробно ядерное преобразование или *ядерный трюк* (kernel trick) мы будем рассматривать в работе «Метод опорных векторов».

Одна из версий ядерного метода *k*-средних реализована в библиотеке Scikit-Learn в оценивателе SpectralClustering . Она использует граф ближайших соседей для вычисления представления данных более высокой размерности, после чего задает соответствие меток с помощью алгоритма *k*-средних (рис. 2.8):

In[10]:from sklearn.cluster import SpectralClustering

model = SpectralClustering(n\_clusters=2,

affinity='nearest\_neighbors',

assign\_labels='kmeans')

labels = model.fit\_predict(X)

plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis');

Как видим, метод *k*-средних с помощью этого ядерного преобразования способен обнаруживать более сложные нелинейные границы между кластерами.

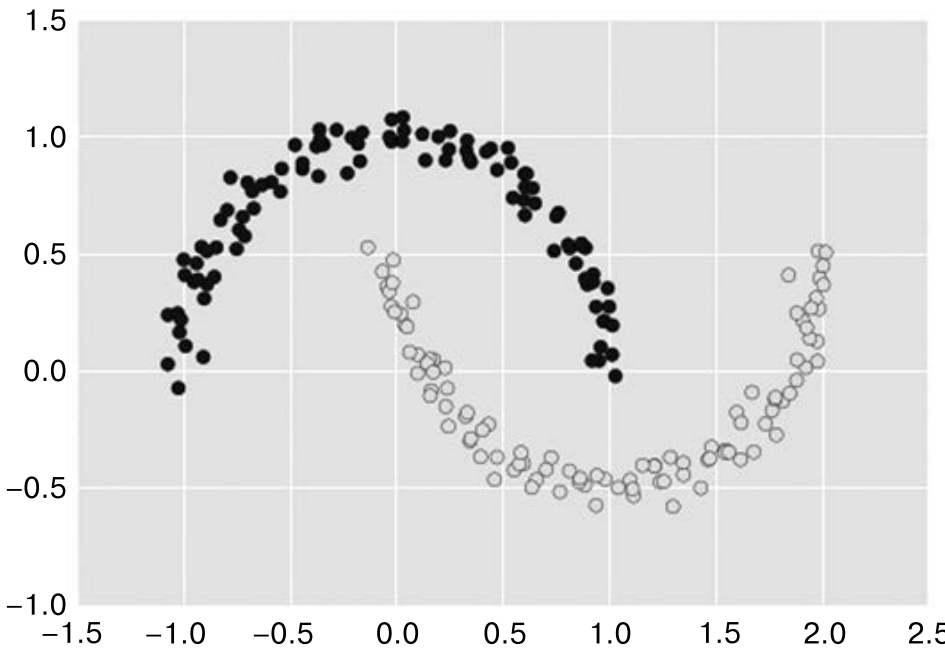


Рис. 2.8. Нелинейные границы, обнаруженные с помощью SpectralClustering

* *Метод k-средних работает довольно медленно в случае большого количества выборок*. Алгоритм может работать довольно медленно при росте числа выборок, ведь при каждой итерации методу k-средних необходимо обращаться к каждой точке в наборе данных. Можно ли смягчить это требование относительно использования всех данных при каждой итерации? Например, можно применить лишь подмножество данных для корректировки центров кластеров на каждом шаге. Эта идея лежит в основе пакетных алгоритмов *k-*средних, один из которых реализован в классе sklearn.cluster.MiniBatchKMeans . Их интерфейс не отличается от обычного KMeans . В дальнейшем мы рассмотрим пример их использования.

**Примеры**

При соблюдении некоторой осторожности в плане вышеупомянутых ограничений можно успешно использовать метод k-средних во множестве ситуаций. Рассмотрим несколько примеров.

**Пример 1: применение метода *k-*средних для рукописных цифр**

Для начала рассмотрим применение метода *k-*средних к тем же простым данным по цифрам, которые мы уже видели в лабораторной работе 1. Мы попробуем воспользоваться методом k-средних для распознания схожих цифр *без использования информации об исходных метках*. Это напоминает первый шаг извлечения смысла нового набора данных, для которого отсутствует какая-либо априорная информации о метках.

Начнем с загрузки цифр, а затем перейдем к поиску кластеров с помощью алгоритма KMeans . Напомним, что набор данных по цифрам состоит из 1797 выборок с 64 признаками, где каждый из признаков представляет собой яркость одного пиксела в изображении размером 8 × 8:

In[11]:from sklearn.datasets import load\_digits

digits = load\_digits()

digits.data.shape

Out[11]: (1797, 64)

Кластеризация выполняется так же, как и ранее:

In[12]: kmeans = KMeans(n\_clusters=10, random\_state=0)

clusters = kmeans.fit\_predict(digits.data)

kmeans.cluster\_centers\_.shape

Out[12]: (10, 64)

В результате мы получили десять кластеров в 64-мерном пространстве. Обратите внимание, что и центры кластеров представляют собой 64-мерные точки, а значит, их можно интерпретировать как «типичные» цифры в кластере. Посмотрим, что представляют собой эти центры кластеров (рис. 2.9):

In[13]: fig, ax = plt.subplots(2, 5, figsize=(8, 3))

centers = kmeans.cluster\_centers\_.reshape(10, 8, 8)

for axi, center in zip(ax.flat, centers):

axi.set(xticks=[], yticks=[])

axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)

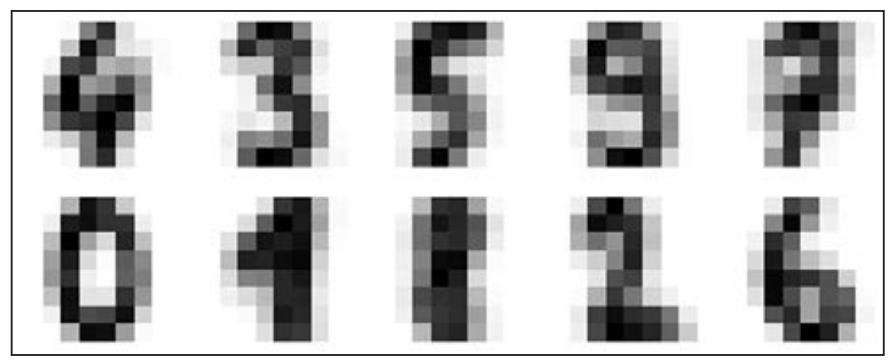


Рис. 2.9. Центры кластеров

Как видим, алгоритм KMeans даже без меток способен определить кластеры, чьи центры представляют собой легко узнаваемые цифры, возможно, за исключением 1 и 8.

В силу того что алгоритм *k-*средних ничего не знает о сущности кластеров, метки 0–9 могут оказаться перепутаны местами. Исправить это можно, задав соответствие всех полученных меток кластеров имеющимся в них фактическим меткам:

In[14]: from scipy.stats import mode

labels = np.zeros\_like(clusters)

for i in range(10):

mask = (clusters == i)

labels[mask] = mode(digits.target[mask])[0]

Теперь можно проверить, насколько точно кластеризация без учителя определила подобие цифр в наших данных:

In[15]: from sklearn.metrics import accuracy\_score

accuracy\_score(digits.target, labels)

Out[15]: 0.79354479688369506

С помощью простого алгоритма *k-*средних мы определили правильную группировку для почти 80 % исходных цифр! Посмотрим на матрицу различий

(рис. 2.10):

In[16]: from sklearn.metrics import confusion\_matrix

mat = confusion\_matrix(digits.target, labels)

sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,

xticklabels=digits.target\_names,

yticklabels=digits.target\_names)

plt.xlabel('true label')

plt.ylabel('predicted label');

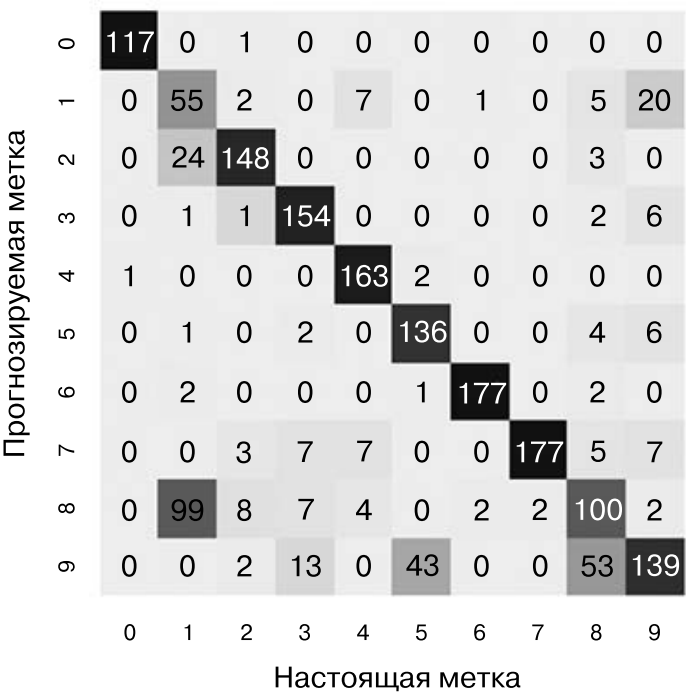


Рис. 2.10. Матрица различий для классификатора методом *k-*средних

Как и можно было ожидать, судя по визуализированным ранее центрам кластеров, больше всего путаницы между единицами и восьмерками. Но все равно, с помощью метода *k-*средних мы фактически создали классификатор цифр без каких-либо известных меток!

Попробуем пойти еще дальше. Воспользуемся для предварительной обработки данных до выполнения *k-*средних алгоритмом стохастического вложения соседей на основе распределения Стьюдента (t-SNE). t-SNE — нелинейный алгоритм вложения, особенно хорошо умеющий сохранять точки внутри кластеров. Посмотрим, как он работает:

In[17]: from sklearn.manifold import TSNE

# Проекция данных: выполнение этого шага займет несколько секунд

tsne = TSNE(n\_components=2, init='pca', random\_state=0)

digits\_proj = tsne.fit\_transform(digits.data) # Расчет кластеров

kmeans = KMeans(n\_clusters=10, random\_state=0)

clusters = kmeans.fit\_predict(digits\_proj) # Перестановка меток местами

labels = np.zeros\_like(clusters)

for i in range(10):

mask = (clusters == i)

labels[mask] = mode(digits.target[mask])[0]

# Оценка точности

accuracy\_score(digits.target, labels)

Out[17]: 0.93356149137451305

Точность, даже без использования меток, составляет почти 94 %. При разумном использовании алгоритмы машинного обучения без учителя демонстрируют отличные результаты: они могут извлекать информацию из набора данных даже тогда, когда сделать это вручную или визуально очень непросто.

**Пример 2: использование метода k-средних для сжатия цветов**

Одно из интересных приложений кластеризации — сжатие цветов в изображениях. Например, допустим, что у нас есть изображение с миллионами цветов. Почти во всех изображениях большая часть цветов не используется и цвета многих пикселов изображения будут похожи или даже совпадать.

Например, рассмотрим изображение, показанное на рис. 2.11, взятом из модуля datasets библиотеки Scikit-Learn (для работы следующего кода у вас должен быть установлен пакет pillow языка Python):

In[18]: # Обратите внимание: для работы этого кода

# должен быть установлен пакет pillow

from sklearn.datasets import load\_sample\_image

china = load\_sample\_image("china.jpg")

ax = plt.axes(xticks=[], yticks=[])

ax.imshow(china);



Рис. 2.11. Исходное изображение

Само изображение хранится в трехмерном массиве размера ( высота , ширина , RGB ), содержащем вклад в цвет по красному/синему/зеленому каналам в виде целочисленных значений от 0 до 255:

In[19]: china.shape

Out[19]: (427, 640, 3)

Этот набор пикселов можно рассматривать как облако точек в трехмерном цветовом пространстве. Изменим форму данных на [n\_samples × n\_features] и масштабируем шкалу цветов так, чтобы они располагались между 0 и 1:

In[20]: data = china / 255.0 # используем шкалу 0...1

data = data.reshape(427 \* 640, 3)

data.shape

Out[20]: (273280, 3)

Визуализируем эти пикселы в данном цветовом пространстве, используя подмножество из 10 000 пикселов для быстроты работы (рис. 2.12):

In[21]:

def plot\_pixels(data, title, colors=None, N=10000):

if colors is None:

colors = data

# Выбираем случайное подмножество

rng = np.random.RandomState(0)

i = rng.permutation(data.shape[0])[:N]

colors = colors[i]

R, G, B = data[i].T

fig, ax = plt.subplots(1, 2, figsize=(16, 6))

ax[0].scatter(R, G, color=colors, marker='.')

ax[0].set(xlabel='Red', ylabel='Green', xlim=(0, 1), ylim=(0, 1))

ax[1].scatter(R, B, color=colors, marker='.')

ax[1].set(xlabel='Red', ylabel='Blue', xlim=(0, 1), ylim=(0, 1))

fig.suptitle(title, size=20);

In[22]: plot\_pixels(data, title='Input color space: 16 million possible colors') # Исходное цветовое пространство: 16 миллионов

# возможных цветов

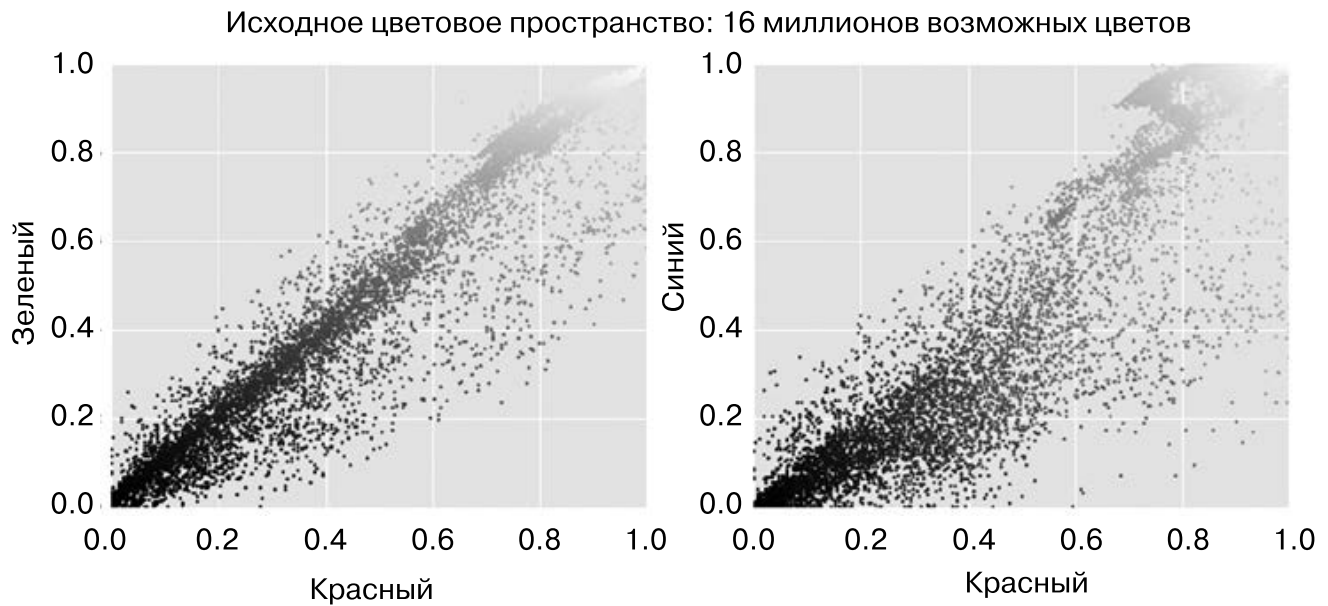


Рис. 2.12. Шестнадцать кластеров в цветовом пространстве RGB

Теперь уменьшим количество цветов с 16 миллионов до 16 путем кластеризацииметодом *k-*средних на пространстве пикселов. Так как наш набор данных очень велик, воспользуемся мини-пакетным методом *k-*средних, который вычисляет результат гораздо быстрее, чем стандартный метод *k-*средних путем работы с подмножествами набора данных (рис. 2.13):

In[23]: from sklearn.cluster import MiniBatchKMeans

kmeans = MiniBatchKMeans(16)

kmeans.fit(data)

new\_colors = kmeans.cluster\_centers\_[kmeans.predict(data)]

plot\_pixels(data, colors=new\_colors, title="Reduced color space: 16 colors")

# Редуцированное цветовое пространство: 16 цветов



Рис. 2.13. Шестнадцать кластеров в цветовом пространстве RGB

В результате исходные пикселы перекрашиваются в другие цвета: каждый пиксел получает цвет ближайшего центра кластера. Рисуя эти новые цвета в пространстве изображения вместо пространства пикселов, видим эффект от перекрашивания (рис. 2.14).

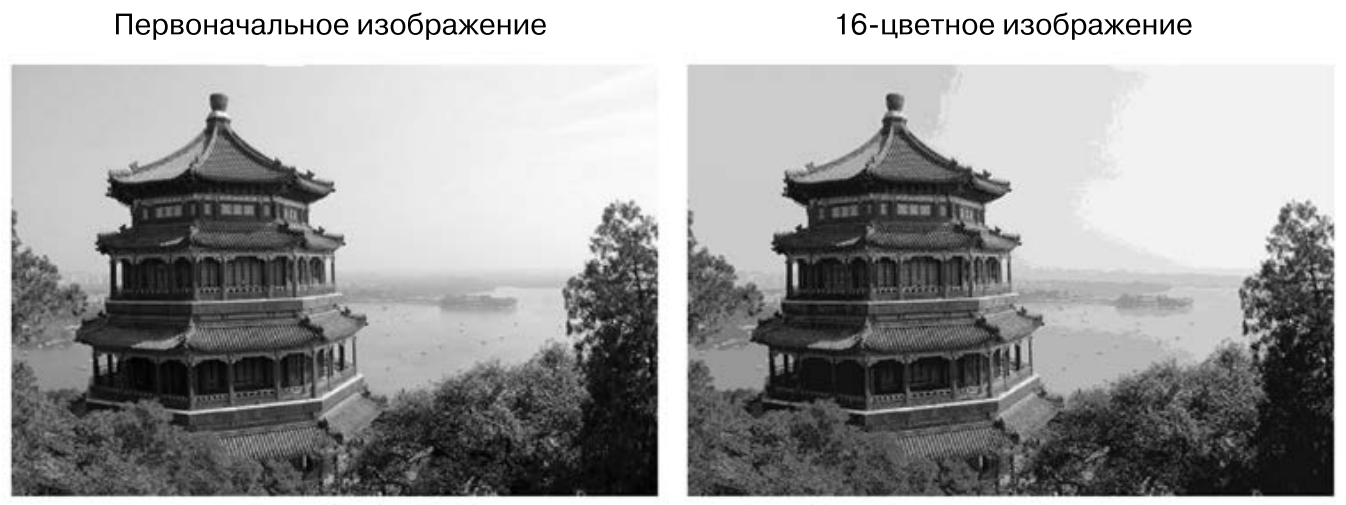


Рис. 2.14. Полноцветное изображение (слева) по сравнению с 16-цветным (справа)

In[24]:

china\_recolored = new\_colors.reshape(china.shape)

fig, ax = plt.subplots(1, 2, figsize=(16, 6),

subplot\_kw=dict(xticks=[], yticks=[]))

fig.subplots\_adjust(wspace=0.05)

ax[0].imshow(china)

ax[0].set\_title('Original Image', size=16) # Первоначальное изображение

ax[1].imshow(china\_recolored)

ax[1].set\_title('16-color Image', size=16); # 16-цветное изображение

Некоторые детали в изображении справа утрачены, но изображение в целом остается вполне узнаваемым. Коэффициент сжатия этого изображения — почти 1 миллион! Существуют лучшие способы сжатия информации в изображениях, но этот пример демонстрирует возможности творческого подхода к методам машинного обучения без учителя, таким как метод *k-*средних.