Лабораторная работа №15

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 **PCA**, мы пытаемся записать каждую точку данных в виде взвешенной суммы некоторых компонентов, как показано на рис. 14.8. 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. 14.7, 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.

Применение NMF к синтетическим данным

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. 15.1) показывает результаты применения **NMF** к двумерным синтетическим данным:

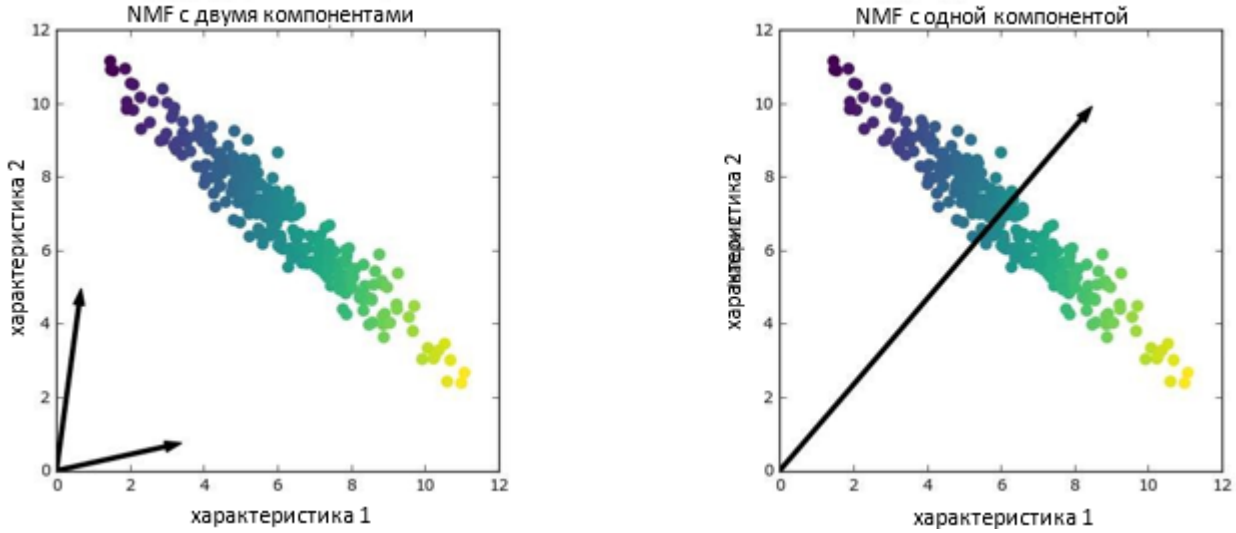

Рис. 15.1 Компоненты, найденные в результате факторизации неотрицательных матриц с двумя компонентами (слева) и одной компонентой (справа)

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 использовани различных случайных значений может привести радикальным изменениям.

Применение NMF к изображением лиц

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. 15.2):

```
mglearn.plots.plot_nmf_faces(X_train, X_test, image_shape)
plt.show()
```

Рис. 15.2 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. 15.3 показывает результат:

```
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': (), 'yti
cks': ()})
for i, (component, ax) in enumerate(zip(nmf.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape))
    ax.set_title("{}. component".format(i))
plt.show()
```

Рис. 15.3 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. 14.7. 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. 15.4 и 15.5):

```
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': (), 'ytic
ks': ()})
```

```
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))
plt.show()

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))
plt.show()
```

Рис. 15.4 Лица с большим коэффициентом компоненты 3

Рис. 15.5 Лица с большим коэффициентом компоненты 7

Как и следовало ожидать, лица с высоким коэффициентом компоненты 3 – это лица, смотрящие вправо (рис. 15.4), тогда как лица с высоким коэффициентом компоненты 7 смотрят влево (рис. 15.5). 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. 15.6):

```
S = mglearn.datasets.make_signals()
plt.figure(figsize=(6, 1))
plt.plot(S, '-')
plt.xlabel("Время")
plt.ylabel("Сигнал")
plt.show()
```

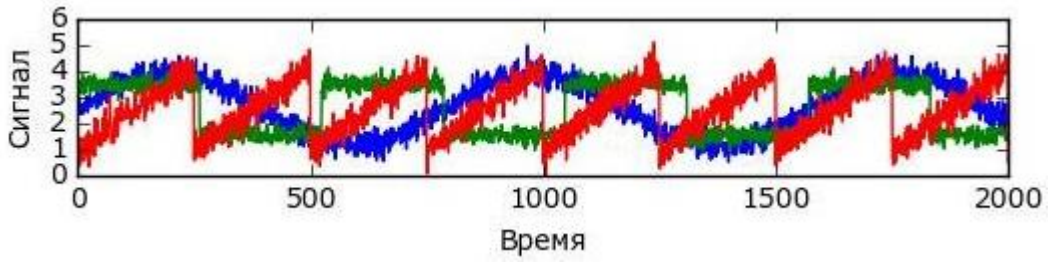

Рис. 15.6 Исходные источники сигнала

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:

```
A = np.random.RandomState(0).uniform(size=(100, 3))
X = np.dot(S, A.T)
print("Форма измерений: {}".format(X.shape))
```

Мы можем использовать **NMF**, чтобы восстановить три сигнала:

```
nmf = NMF(n_components=3, random_state=42)
S_ = nmf.fit_transform(X)
print("Форма восстановленного сигнала: {}".format(S_.shape))
```

Для сравнения мы еще применим **PCA**:

```
pca = PCA(n_components=3)
H = pca.fit_transform(X)
```

Рис. 15.7показывает активность сигнала, обнаруженную с помощью NMF и PCA

```
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], '-')
plt.show()
```

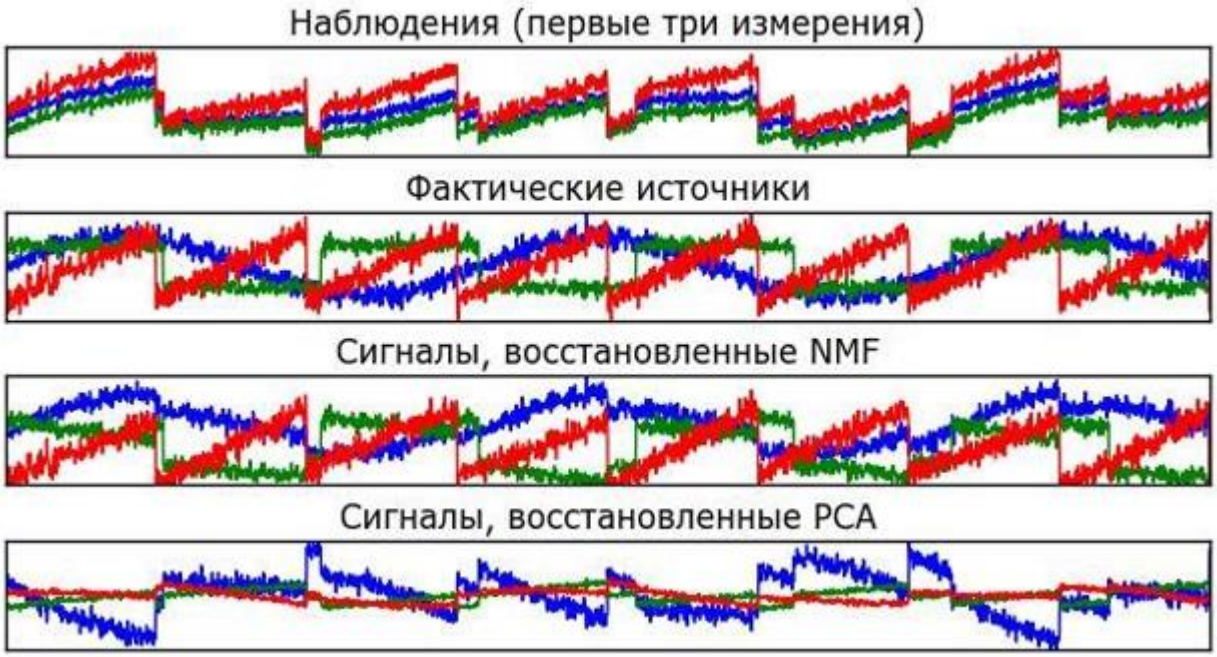

Рис. 15.7 Восстановление первоначальных источников с помощью NMF и PCA

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 большую 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 методам.

Множественное обучение с помощью алгоритма t-SNE

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. Каждая точка данных в этом наборе является изображением цифры в градациях серого. Рис. 15.8 показывает примеры изображений для каждого класса:

```
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)
plt.show()
```

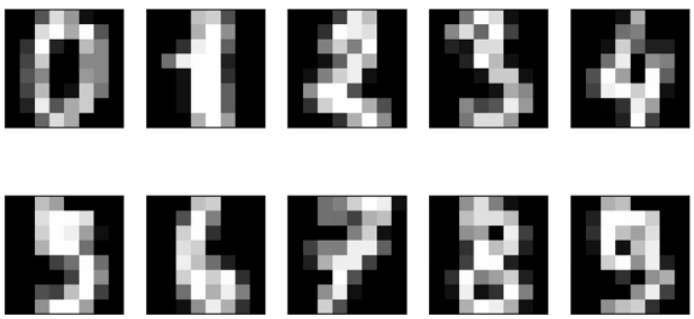

Рис. 15.8 Примеры изображений из набора данных digits

Давайте используем **PCA** для визуализации данных, сведя их к двум измерениям. Мы построим график первых двух главных компонент и отметим цветом класс каждой точки (рис.15.9):

```
# 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", "#4
E655E", "#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")
plt.show()
```

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.

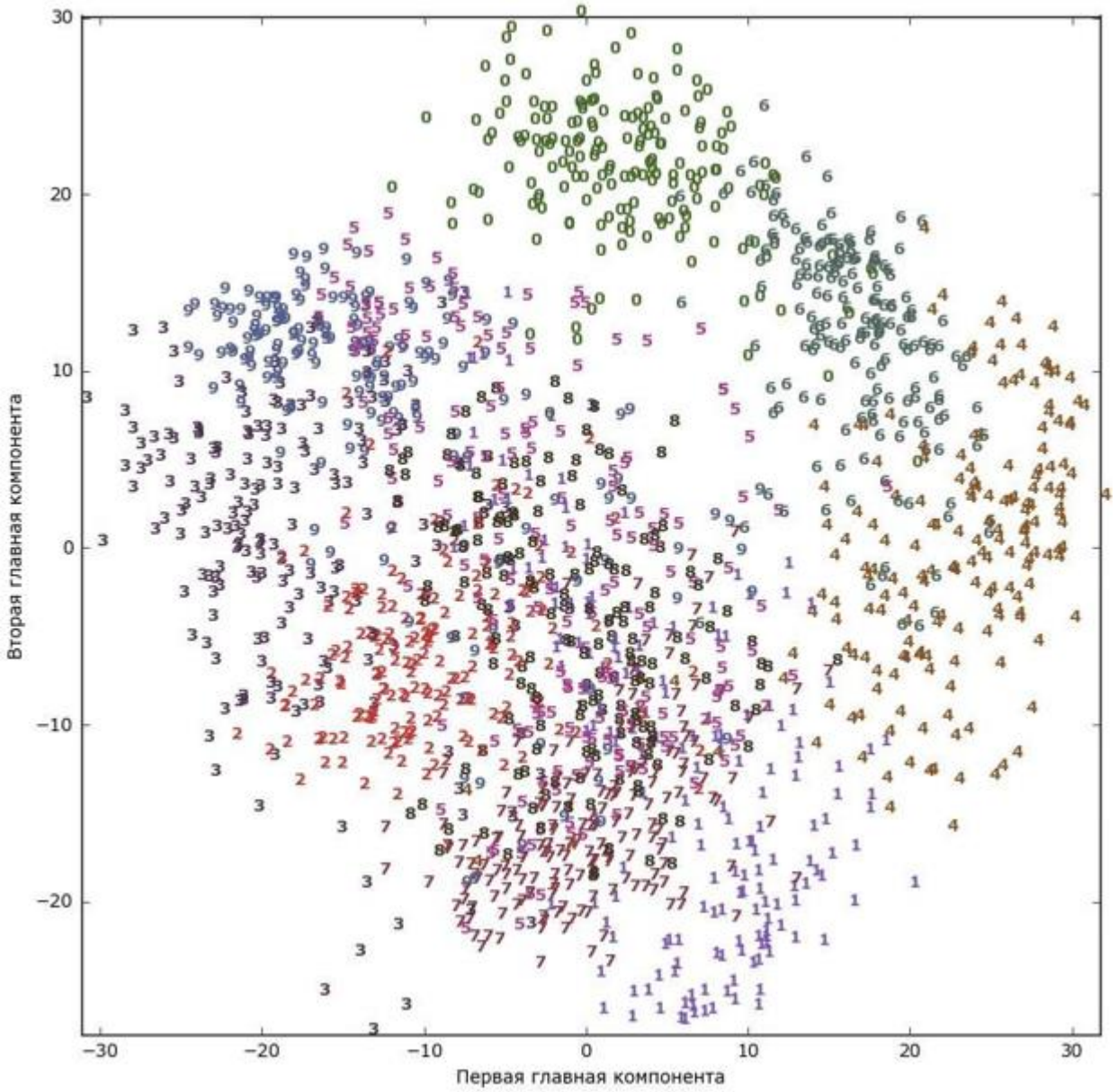

Рис. 15.9 Диаграмма рассеяния для набора данных digits, использующая первые две главные компоненты

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. 15.10):

```
from sklearn.manifold import TSNE

tsne = TSNE(random_state=42)
# используем метод fit_transform вместо fit, т.к. класс TSNE не испол
ьзует метод transform
digits_tsne = tsne.fit_transform(digits.data)
```

```
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")
plt.show()
```

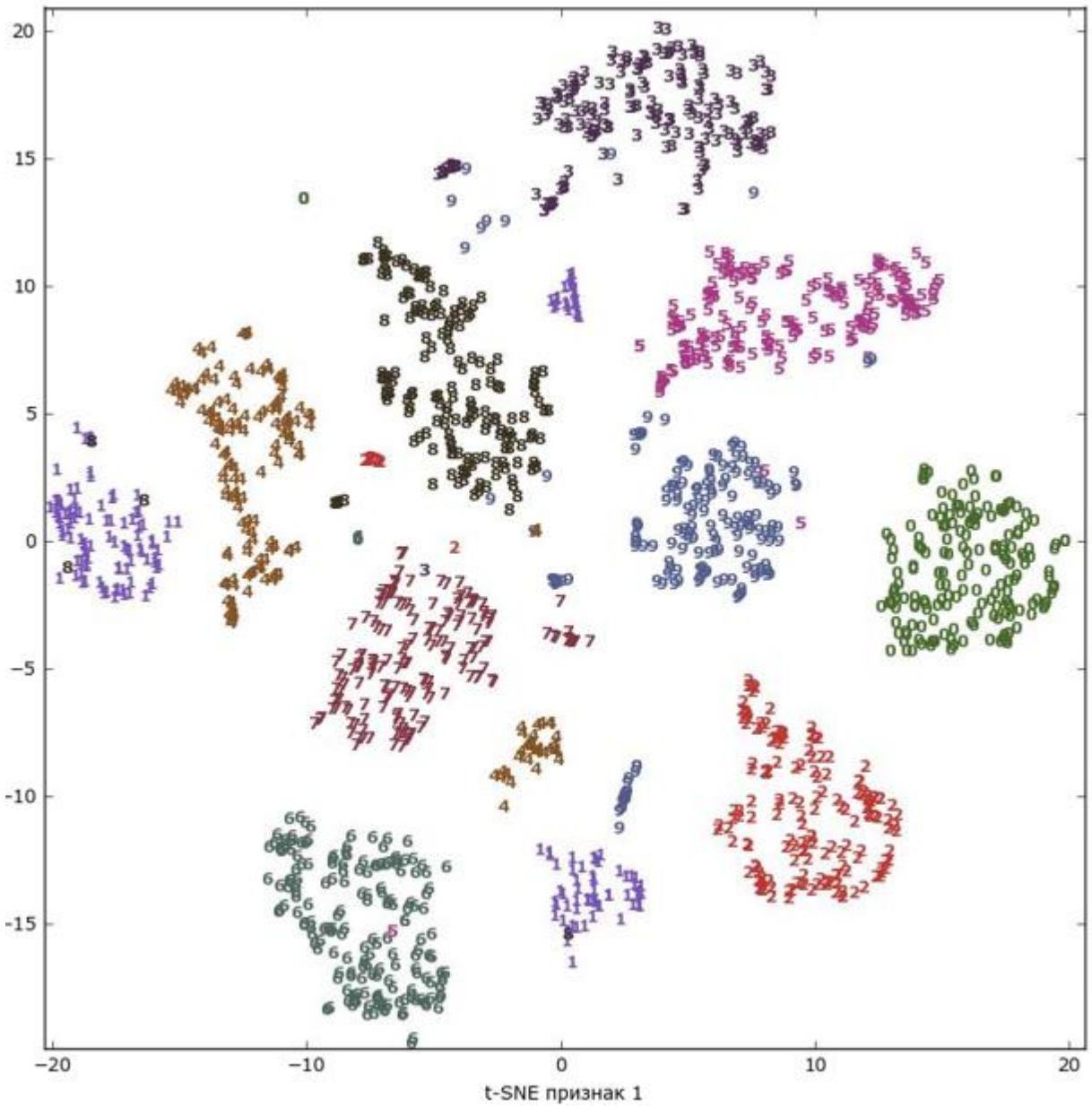

Рис. 15.10 Диаграмма рассеяния для набора данных digits, которая использует первые две главные компоненты, найденные с помощью t-SNE

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.

Код к лабораторной работе:

```
import mglearn
import sklearn
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split


mglearn.plots.plot_nmf_illustration()
plt.show()

# mglearn.plots.plot_nmf_faces(X_train, X_test, image_shape)
# plt.show()



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))
plt.show()


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))
plt.show()

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))
plt.show()


S = mglearn.datasets.make_signals()
plt.figure(figsize=(6, 1))
```

```
plt.plot(S, '-')
plt.xlabel("Время")
plt.ylabel("Сигнал")
plt.show()

A = np.random.RandomState(0).uniform(size=(100, 3))
X = np.dot(S, A.T)
print("Форма измерений: {}".format(X.shape))

nmf = NMF(n_components=3, random_state=42)
S_ = nmf.fit_transform(X)
print("Форма восстановленного сигнала: {}".format(S_.shape))
pca = PCA(n_components=3)
H = pca.fit_transform(X)

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], '-')
plt.show()

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)
plt.show()

# 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", "#3A3
120", "#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")
plt.show()

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)

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")
plt.show()
```