

# Chapter 8 降维

许多机器学习问题涉及到每个训练实例的数千个甚至数百万个特性。所有这些特性不仅使训练变得非常缓慢，而且还会使找到好的解决方案更难，正如你会看到的。这个问题通常被称为 **维数的诅咒（curse of dimensionality）**。

幸运的是，在现实世界中的问题中，通常可以大大减少特性的数量，将一个棘手的问题变成一个易于处理的问题。对于测试对象，考虑MNIST图像（在第3章中介绍）：图像边界上的像素几乎总是白色的，所以你可以从训练集中完全删除这些像素，而不会丢失很多信息。正如我们在上一章中看到的，（图7-6）证实了这些像素对于分类任务完全不重要。此外，两个相邻的两个像素通常高度相关：如果您将它们合并为单个像素（例如，通过取两个像素强度的平均值），您将不会丢失太多信息。

**注意：**降低维度确实会导致一些信息丢失，就像将图像压缩到JPEG会降低其质量一样，所以即使它会加快训练速度，它也可能会让你的系统表现得稍微差一点。它还会使您的管道更加复杂，因此更难维护。因此，我建议您在考虑使用降维方法之前，首先尝试使用原始数据来训练您的系统。在某些情况下，降低训练数据的维数可能会过滤掉一些噪声和不必要的细节，从而产生更高的性能，但一般来说不会；它只会加快训练速度。

除了加速训练外，降维对数据可视化也非常有用。将维数减少到2（或3）可以在图上绘制高维训练集的浓缩视图，并且通常通过视觉检测模式来获得一些重要的见解，比如集群。此外，数据可视化对于将你的结论传达给那些非数据科学家的人至关重要，特别是那些将使用你的研究结果的决策者。

在本章中，我们将首先讨论维度的诅咒，并了解在高维空间中发生了什么。然后，我们将考虑两种主要的降维方法（**投影projection** 和 **流形学习manifold learning**），并且我们将使用三种最流行的降维技术：**PCA**、**随机投影** 和 **局部线性嵌入（LLE）**。

## Setup

```
In [30]: import sys

assert sys.version_info >= (3, 7)
```

```
In [31]: from packaging import version
import sklearn

assert version.parse(sklearn.__version__) >= version.parse("1.0.1")
```

```
In [32]: import matplotlib.pyplot as plt

plt.rc('font', size=14)
plt.rc('axes', labelsizes=14, titlesize=14)
plt.rc('legend', fontsize=14)
plt.rc('xtick', labelsizes=10)
plt.rc('ytick', labelsizes=10)
```

```
In [33]: from pathlib import Path

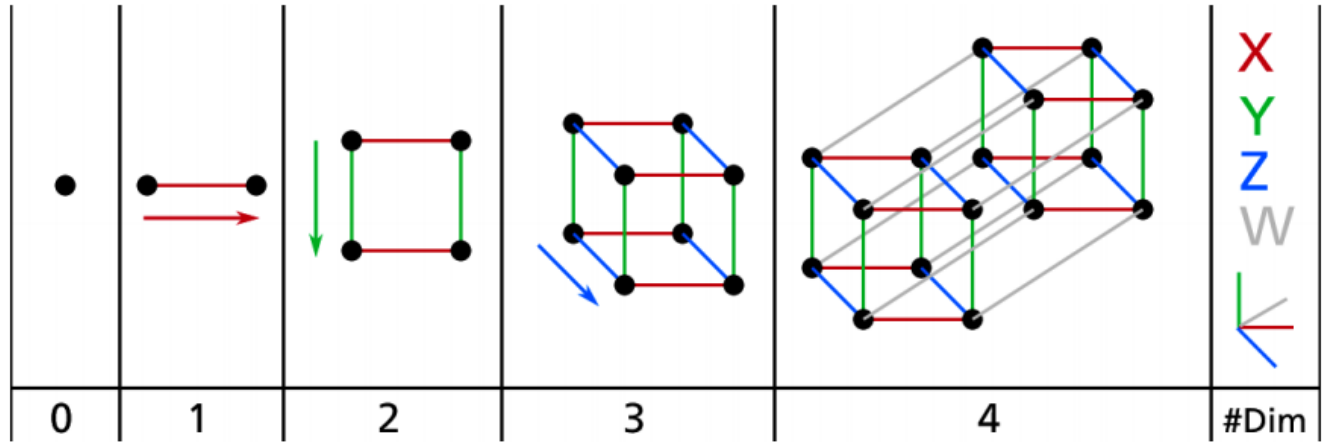
IMAGES_PATH = Path() / "images" / "dim_reduction"
IMAGES_PATH.mkdir(parents=True, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
```

```
path = IMAGES_PATH / f"{fig_id}.{fig_extension}"
if tight_layout:
    plt.tight_layout()
plt.savefig(path, format=fig_extension, dpi=resolution)
```

## 1. 维度的诅咒（The Curse of Dimensionality）

我们已经习惯于生活在三维中，当我们试图想象一个高维空间时，我们的直觉失败了。即使是一个基本的4D超立方体在我们的脑海中也非常很难想象，更不用说一个在1000维空间中的200维椭球体弯曲了。



事实证明，许多事物在高维空间中的表现非常不同。例如，如果您在一个单位正方形（ $1 \times 1$ 的正方形）中随机选择一个点，它只有大约 0.4% 的机会位于距边界小于 0.001 的位置（换句话说，随机点不太可能在任何维度上都是“极端的”）。但在10000维单位超立方体中，这个概率大于99.999999%。高维超立方体中的大多数点都非常靠近边界。

这里有一个更麻烦的区别：如果你在一个单位平方中随机选取两个点，这两个点之间的距离平均大约为 0.52。如果你在一个三维单位立方体中选择两个随机点，平均距离大约是0.66。但是在一个100万维的超立方体中随机选取的两个点呢？信不信由你，平均距离将约为408.25！这是违反直觉的：当两点都位于同一个单位的超立方体内时，它们怎么能相距如此之远呢？在高维空间中只有足够的空间。因此，高维数据集面临着非常稀疏的风险：大多数训练实例很可能彼此相距遥远。这也意味着一个新的实例可能会远离任何训练实例，这使得预测远不如在更低维度上的预测可靠，因为它们将基于更大的外推。

简而言之，训练集拥有的维度越多，其过拟合的风险就越大。

理论上，解决维数灾难的一种方法是增加训练集的大小以达到足够的训练实例密度。不幸的是，在实践中，达到给定密度所需的训练实例数量随维数呈指数增长。只有 100 个特征——比 MNIST 问题中的要少得多——都在 0 到 1 之间，你需要比可观察宇宙中的原子更多的训练实例，以便训练实例之间的平均误差在 0.1 以内，假设它们是均匀地分布在所有维度上。

## 2. 降维的主要方法（Main Approaches for Dimensionality Reduction）

在我们深入研究特定的降维算法之前，让我们来看看两种主要的降维方法：投影和流形学习。

### 2.1 投影（Projection）

在大多数现实世界的问题中，训练实例并不是均匀地分布到所有维度上的。许多特性几乎是恒定的，而其他特性则是高度相关的（如前面讨论的MNIST）。因此，所有的训练实例都位于（或接近）高维空间的一个更



```

w1, w2 = np.linalg.solve(Vt[:2, :2], Vt[:2, 2]) # projection plane coefs
z = w1 * (x1 - pca.mean_[0]) + w2 * (x2 - pca.mean_[1]) - pca.mean_[2] # plane
X3D_above = X[X[:, 2] >= X3D_inv[:, 2]] # samples above plane
X3D_below = X[X[:, 2] < X3D_inv[:, 2]] # samples below plane

fig = plt.figure(figsize=(9, 9))
ax = fig.add_subplot(111, projection="3d")

# plot samples and projection lines below plane first
ax.plot(X3D_below[:, 0], X3D_below[:, 1], X3D_below[:, 2], "ro", alpha=0.3)
for i in range(m):
    if X[i, 2] < X3D_inv[i, 2]:
        ax.plot([X[i][0], X3D_inv[i][0]],
                [X[i][1], X3D_inv[i][1]],
                [X[i][2], X3D_inv[i][2]], ":", color="#F88")

ax.plot_surface(x1, x2, z, alpha=0.1, color="b") # projection plane
ax.plot(X3D_inv[:, 0], X3D_inv[:, 1], X3D_inv[:, 2], "b+") # projected samples
ax.plot(X3D_inv[:, 0], X3D_inv[:, 1], X3D_inv[:, 2], "b.")

# now plot projection lines and samples above plane
for i in range(m):
    if X[i, 2] >= X3D_inv[i, 2]:
        ax.plot([X[i][0], X3D_inv[i][0]],
                [X[i][1], X3D_inv[i][1]],
                [X[i][2], X3D_inv[i][2]], "r--")

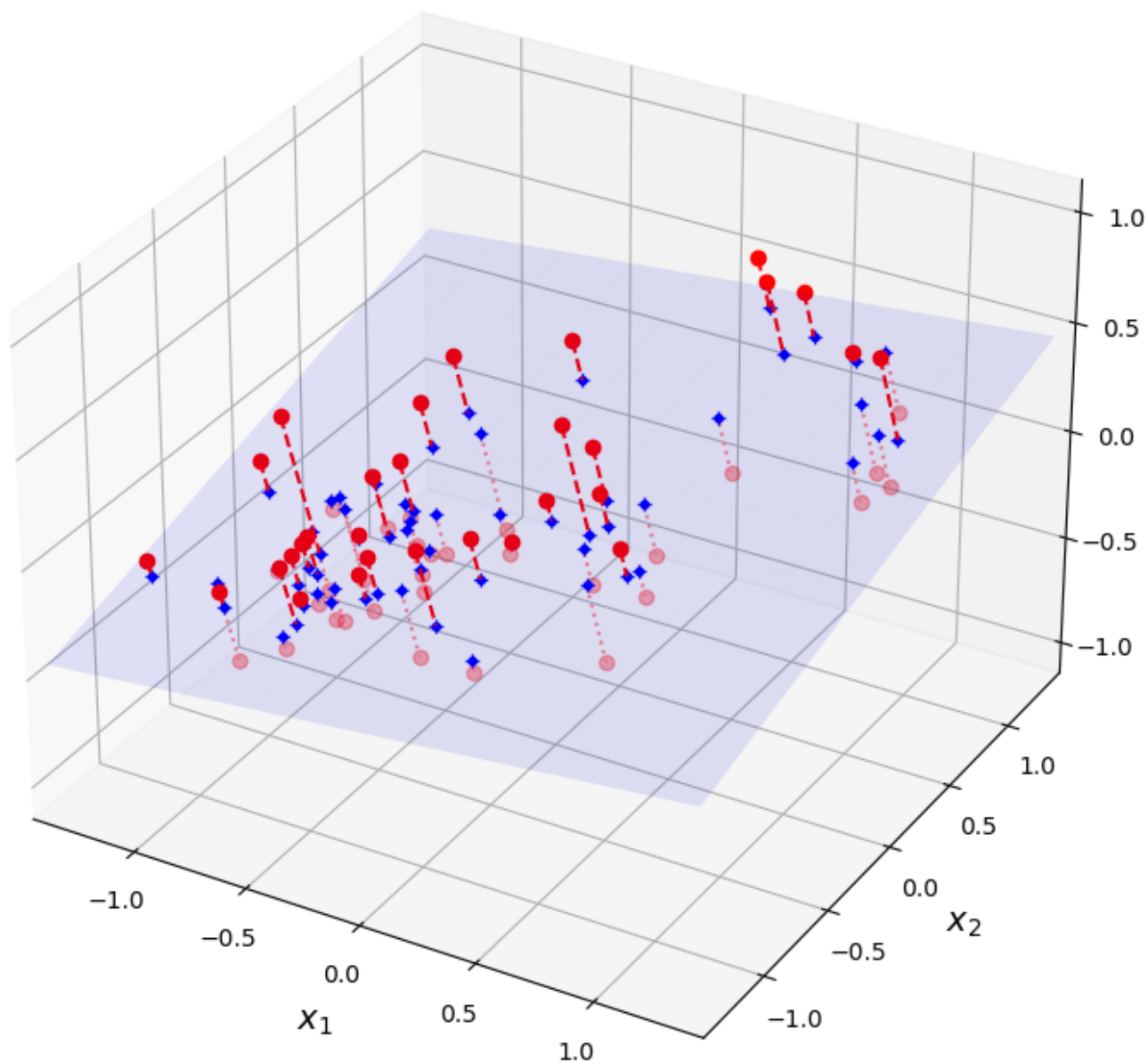
ax.plot(X3D_above[:, 0], X3D_above[:, 1], X3D_above[:, 2], "ro")

def set_xyz_axes(ax, axes):
    ax.xaxis.set_rotate_label(False)
    ax.yaxis.set_rotate_label(False)
    ax.zaxis.set_rotate_label(False)
    ax.set_xlabel("$x_1$", labelpad=8, rotation=0)
    ax.set_ylabel("$x_2$", labelpad=8, rotation=0)
    ax.set_zlabel("$x_3$", labelpad=8, rotation=0)
    ax.set_xlim(axes[0:2])
    ax.set_ylim(axes[2:4])
    ax.set_zlim(axes[4:6])

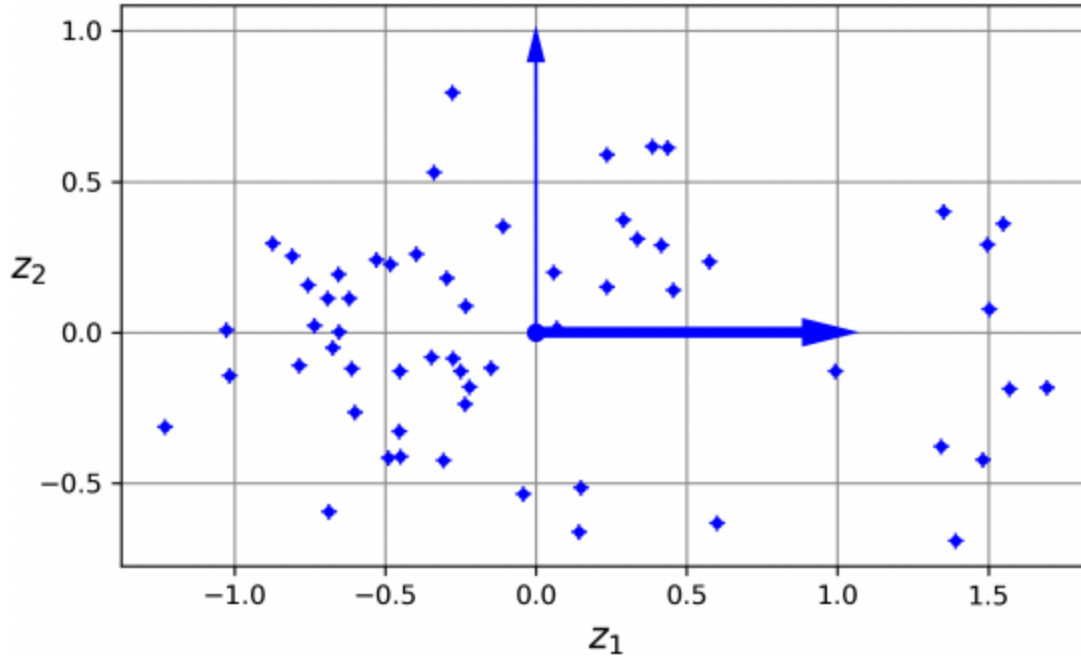
set_xyz_axes(ax, axes)
ax.set_zticks([-1, -0.5, 0, 0.5, 1])

save_fig("dataset_3d_plot", tight_layout=False)
plt.show()

```

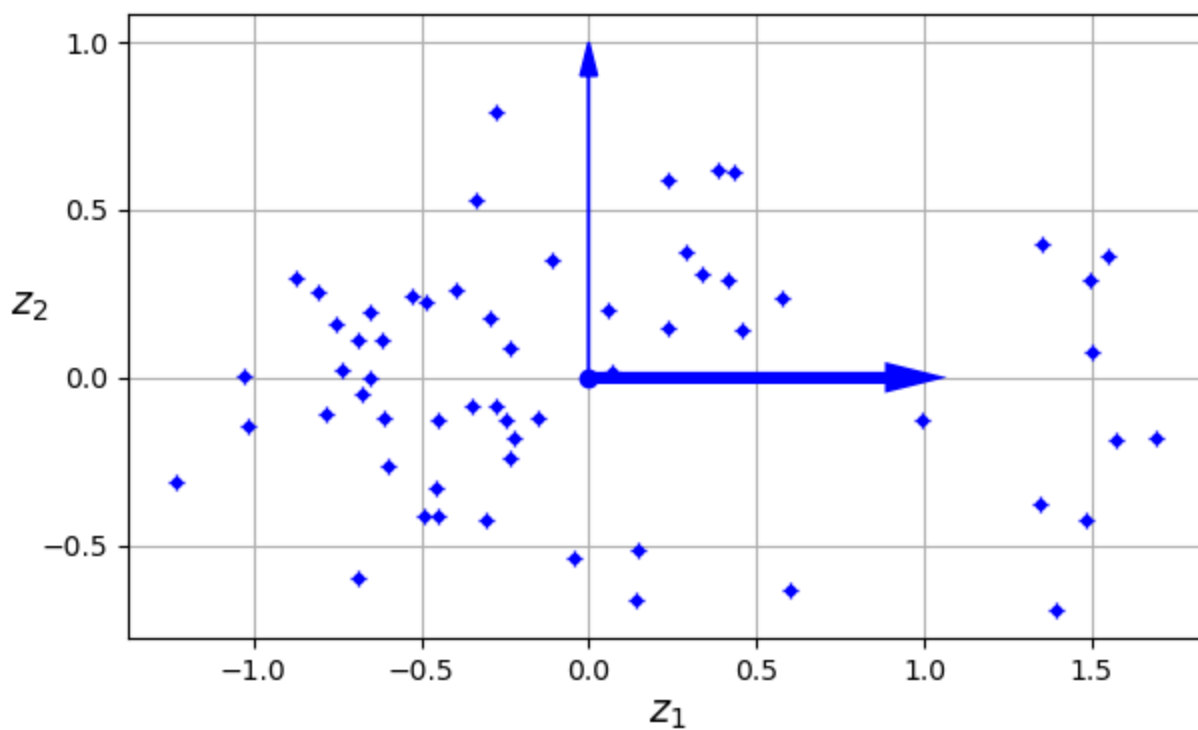


请注意，所有的训练实例都靠近一个平面：这是一个高维（3D）空间的一个低维（2D）子空间。如果我们将每个训练实例垂直投影到这个子空间上（用连接实例到平面的短虚线表示），我们将得到如下图所示的新的2D数据集。我们刚刚将数据集的维数从三维降为二维。请注意，这些轴对应于新的特征  $z_1$  和  $z_2$ ：它们是平面上投影的坐标。



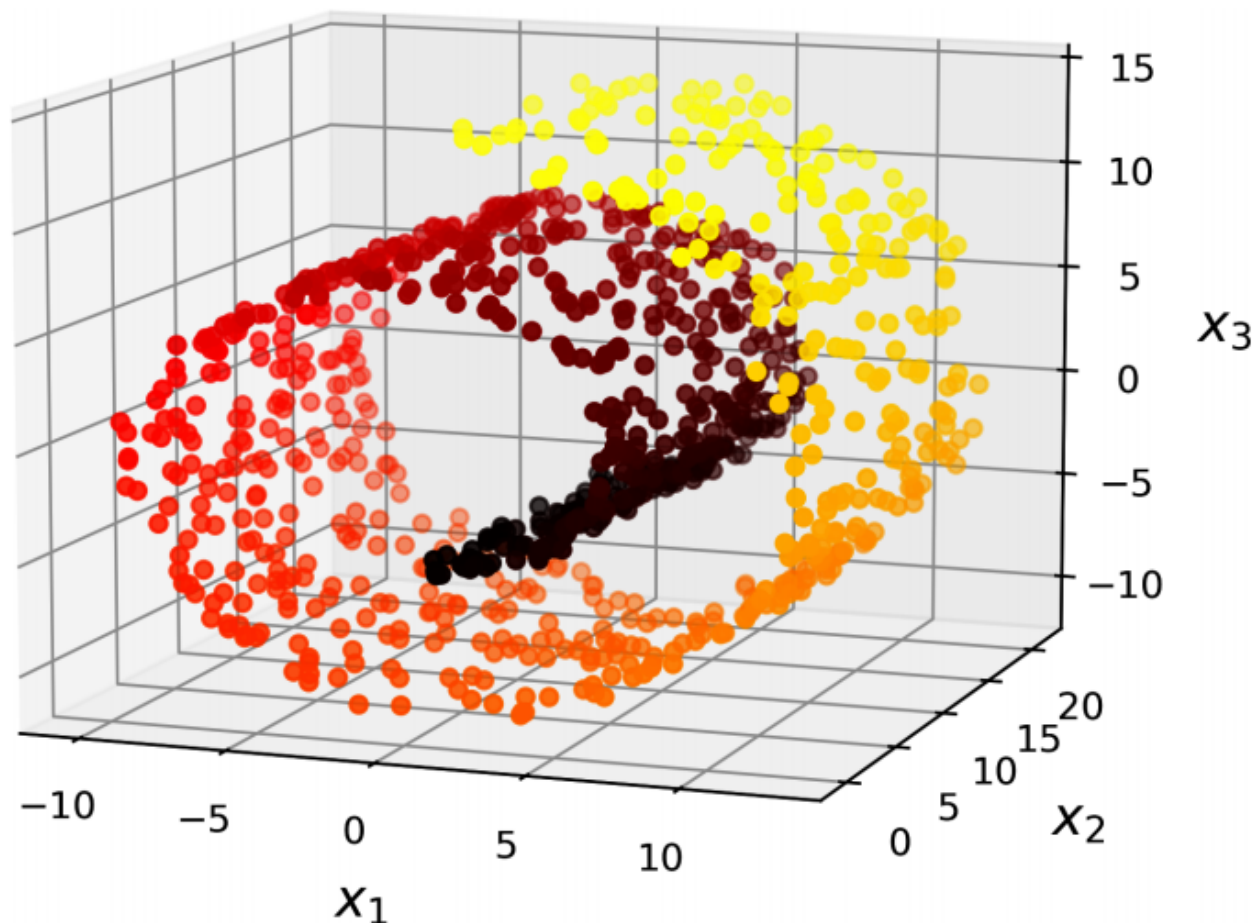
In [36]: *# extra code - this cell generates and saves Figure 8-3*

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, aspect='equal')
ax.plot(X2D[:, 0], X2D[:, 1], "b+")
ax.plot(X2D[:, 0], X2D[:, 1], "b.")
ax.plot([0], [0], "bo")
ax.arrow(0, 0, 1, 0, head_width=0.05, length_includes_head=True,
        head_length=0.1, fc='b', ec='b', linewidth=4)
ax.arrow(0, 0, 0, 1, head_width=0.05, length_includes_head=True,
        head_length=0.1, fc='b', ec='b', linewidth=1)
ax.set_xlabel("$z_1$")
ax.set_yticks([-0.5, 0, 0.5, 1])
ax.set_ylabel("$z_2$", rotation=0)
ax.set_axisbelow(True)
ax.grid(True)
save_fig("dataset_2d_plot")
```



## 2.2 流形学习 (Manifold Learning)

然而，投影并不总是降维的最佳方法。在许多情况下，子空间可能会扭曲和旋转，例如在下图中表示的著名的瑞士滚筒玩具数据集中。



```
In [37]: from sklearn.datasets import make_swiss_roll
```

```
X_swiss, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)
```

```
In [38]: # extra code - this cell generates and saves Figure
```

```
from matplotlib.colors import ListedColormap
```

```
darker_hot = ListedColormap(plt.cm.hot(np.linspace(0, 0.8, 256)))
```

```
axes = [-11.5, 14, -2, 23, -12, 15]
```

```
fig = plt.figure(figsize=(6, 5))
```

```
ax = fig.add_subplot(111, projection='3d')
```

```
ax.scatter(X_swiss[:, 0], X_swiss[:, 1], X_swiss[:, 2], c=t, cmap=darker_hot)
```

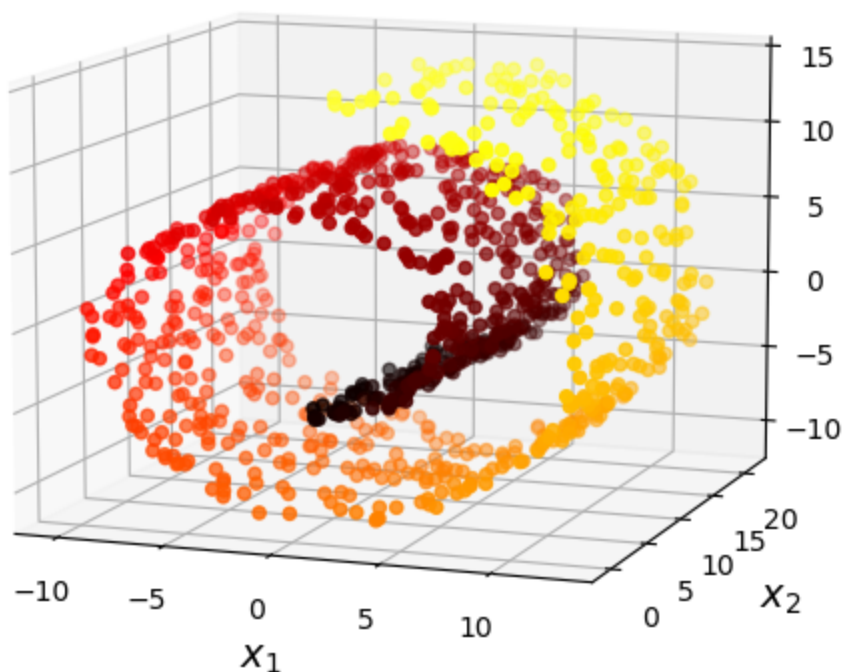
```
ax.view_init(10, -70)
```

```
set_xyz_axes(ax, axes)
```

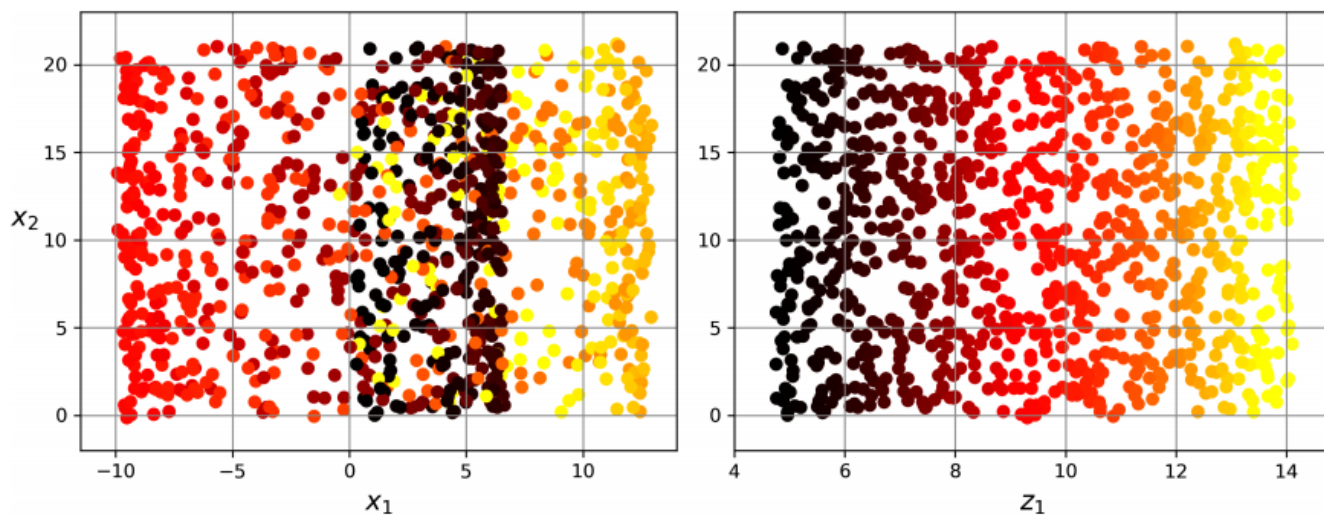
```
save_fig("swiss_roll_plot")
```

```
plt.show()
```





简单地投影到一个平面上（例如，通过下降  $x_3$ ）就会将不同的瑞士滚动层挤压在一起，如下图的左侧所示。相反，您可能想要的是展开瑞士卷，以获得下图右侧的2D数据集。



In [39]: `# extra code - this cell generates and saves plots for Figure 8-5`

```
plt.figure(figsize=(10, 4))

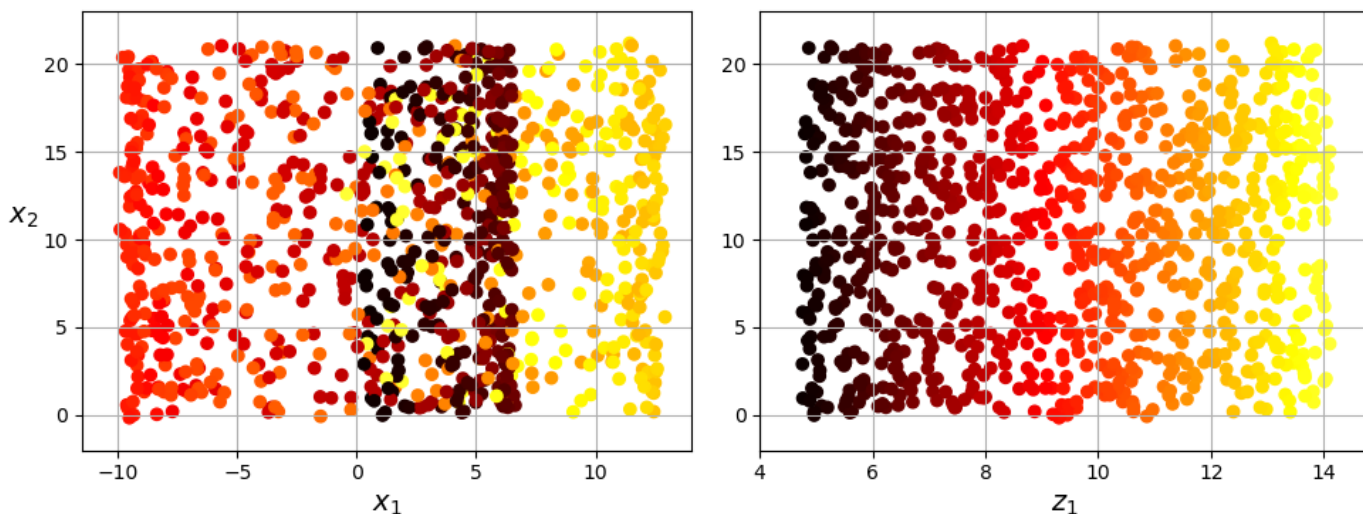
plt.subplot(121)
plt.scatter(X_swiss[:, 0], X_swiss[:, 1], c=t, cmap=darker_hot)
plt.axis(axes[:4])
plt.xlabel("$x_1$")
plt.ylabel("$x_2$", labelpad=10, rotation=0)
plt.grid(True)

plt.subplot(122)
plt.scatter(t, X_swiss[:, 1], c=t, cmap=darker_hot)
```



```
plt.axis([4, 14.8, axes[2], axes[3]])
plt.xlabel("$z_1$")
plt.grid(True)

save_fig("squished_swiss_roll_plot")
plt.show()
```

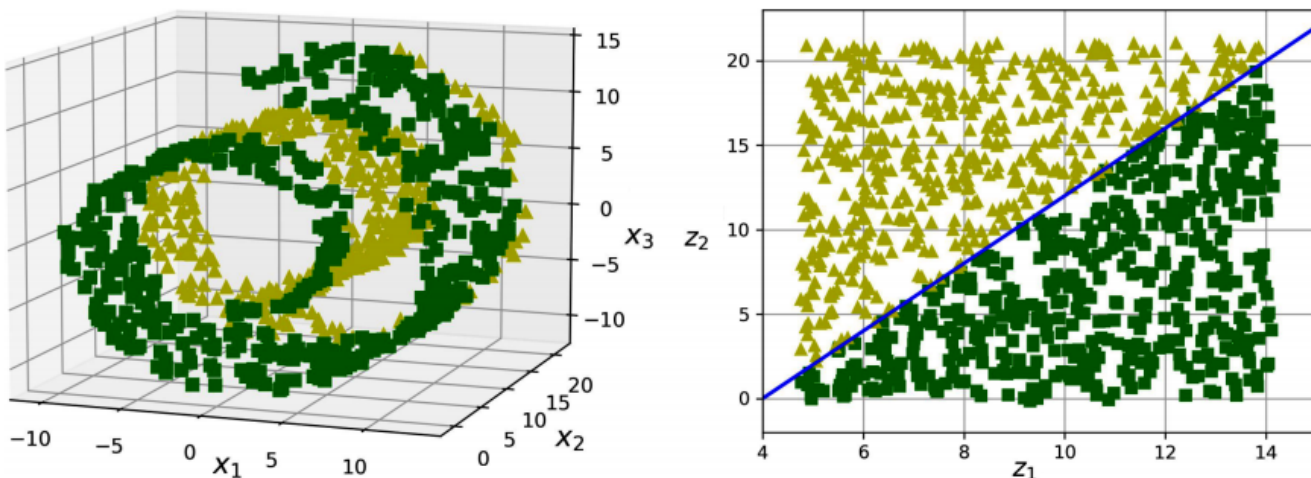


瑞士卷是一个二维流形的一个例子。简单地说，二维流形是一种可以在高维空间中弯曲和扭曲的二维形状。更一般地说， $d$  维流形是  $n$  维空间（其中  $d < n$ ）的一部分，它局部类似于  $d$  维超平面。在瑞士卷的情况下， $d = 2$  和  $n = 3$ ：它在局部类似于一个二维平面，但它是在三维空间中滚动的。

许多降维算法通过建模训练实例所在的流形来工作；这被称为 **流形学习 (manifold learning)**。它依赖于流形假设 (**manifold assumption**)，也称为 **流形假说 (manifold hypothesis)**，它认为大多数现实世界的高维数据集接近一个更低维的流形。这种假设经常被经验观察到。

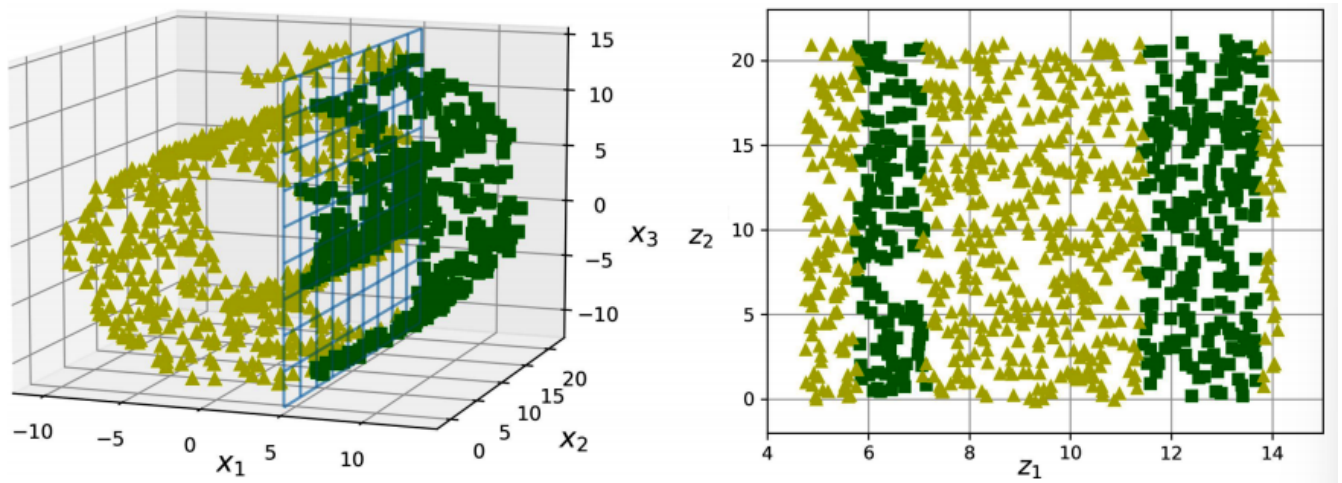
再一次，想想 MNIST 数据集：所有手写的数字图像都有一些相似之处。它们是由连接的线组成的，边界是白色的，它们或多或少是位于中心的。如果你随机生成图像，只有一小部分看起来像手写的数字。换句话说，如果你试图创建一个数字图像，你可以获得的自由度远远低于如果你可以生成任何你想要的图像的自由度。这些约束条件倾向于将数据集挤成一个较低维的流形中。

流形假设通常伴随着另一个隐含的假设：如果用流形的低维空间来表示，手头的任务（例如，分类或回归）将会更简单。例如，在下图中，瑞士卷被分为两类：在三维空间（左边）中，决策边界相当复杂，但在二维展开流形空间（右边）中，决策边界是一条直线。



然而，这种隐含的假设并不总是成立的。例如，在下图中，决策边界位于  $x_1 = 5$  处。这个决策边界在原始的三维空间（一个垂直平面）中看起来非常简单，但在展开的流形（一个由四个独立的线段组成的集合）中看

起来更复杂。



In [40]: *# extra code - this cell generates and saves plots for Figure 8-6*

```
axes = [-11.5, 14, -2, 23, -12, 15]
x2s = np.linspace(axes[2], axes[3], 10)
x3s = np.linspace(axes[4], axes[5], 10)
x2, x3 = np.meshgrid(x2s, x3s)

positive_class = X_swiss[:, 0] > 5
X_pos = X_swiss[positive_class]
X_neg = X_swiss[~positive_class]

fig = plt.figure(figsize=(6, 5))
ax = plt.subplot(1, 1, 1, projection='3d')
ax.view_init(10, -70)
ax.plot(X_neg[:, 0], X_neg[:, 1], X_neg[:, 2], "y^")
ax.plot_wireframe(5, x2, x3, alpha=0.5)
ax.plot(X_pos[:, 0], X_pos[:, 1], X_pos[:, 2], "gs")
set_xyz_axes(ax, axes)
save_fig("manifold_decision_boundary_plot1")
plt.show()

fig = plt.figure(figsize=(5, 4))
ax = plt.subplot(1, 1, 1)
ax.plot(t[positive_class], X_swiss[positive_class, 1], "gs")
ax.plot(t[~positive_class], X_swiss[~positive_class, 1], "y^")
ax.axis([4, 15, axes[2], axes[3]])
ax.set_xlabel("$z_1$")
ax.set_ylabel("$z_2$", rotation=0, labelpad=8)
ax.grid(True)
save_fig("manifold_decision_boundary_plot2")
plt.show()

positive_class = 2 * (t[:, 1] - 4) > X_swiss[:, 1]
X_pos = X_swiss[positive_class]
X_neg = X_swiss[~positive_class]

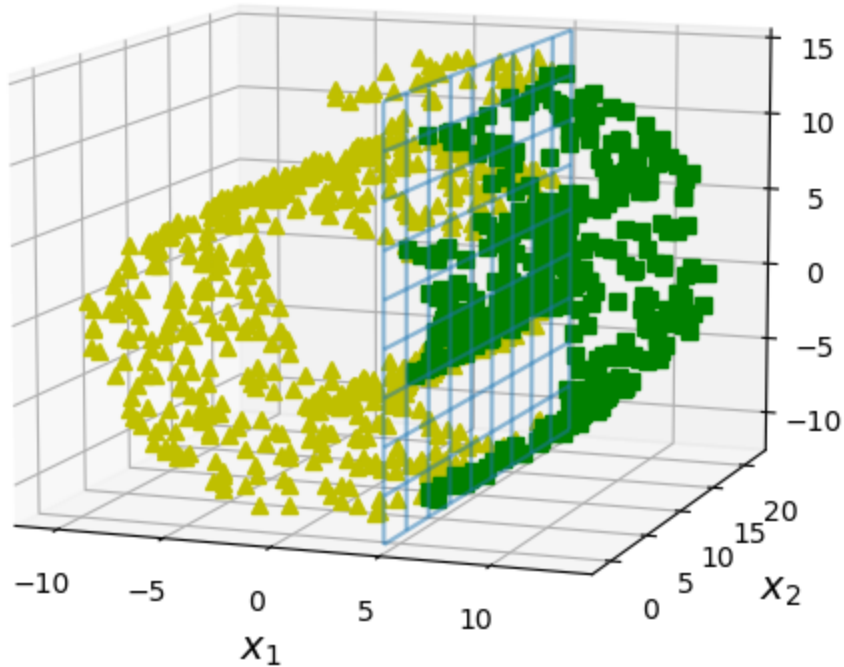
fig = plt.figure(figsize=(6, 5))
ax = plt.subplot(1, 1, 1, projection='3d')
ax.view_init(10, -70)
ax.plot(X_neg[:, 0], X_neg[:, 1], X_neg[:, 2], "y^")
ax.plot(X_pos[:, 0], X_pos[:, 1], X_pos[:, 2], "gs")
ax.xaxis.set_rotate_label(False)
ax.yaxis.set_rotate_label(False)
ax.zaxis.set_rotate_label(False)
ax.set_xlabel("$x_1$", rotation=0)
ax.set_ylabel("$x_2$", rotation=0)
```

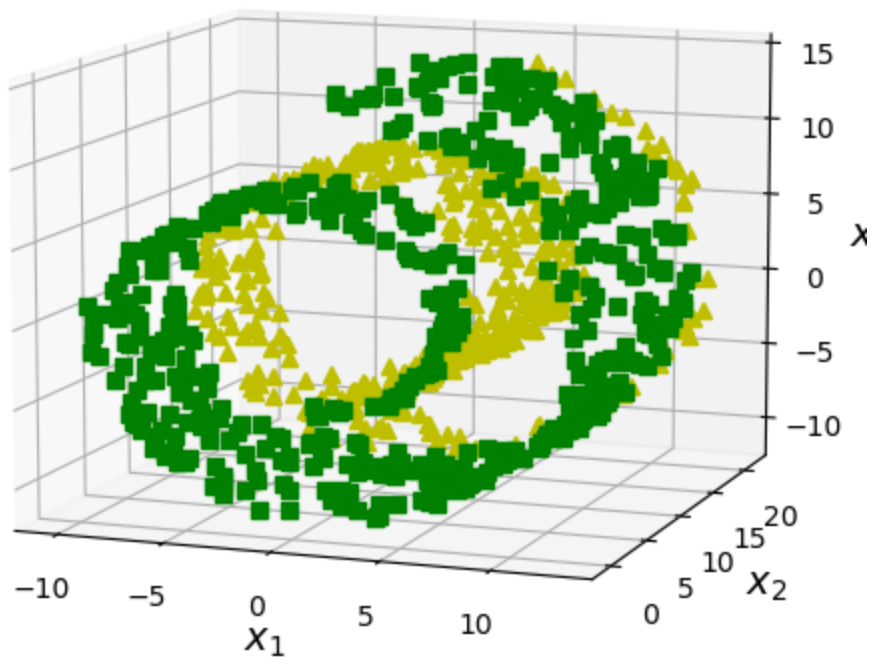
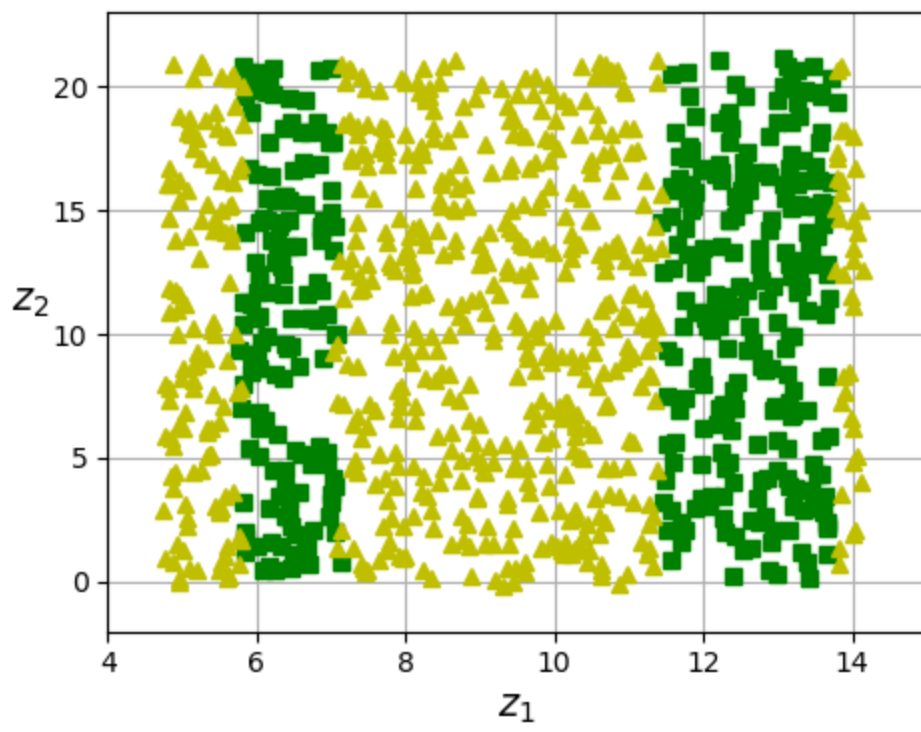
```

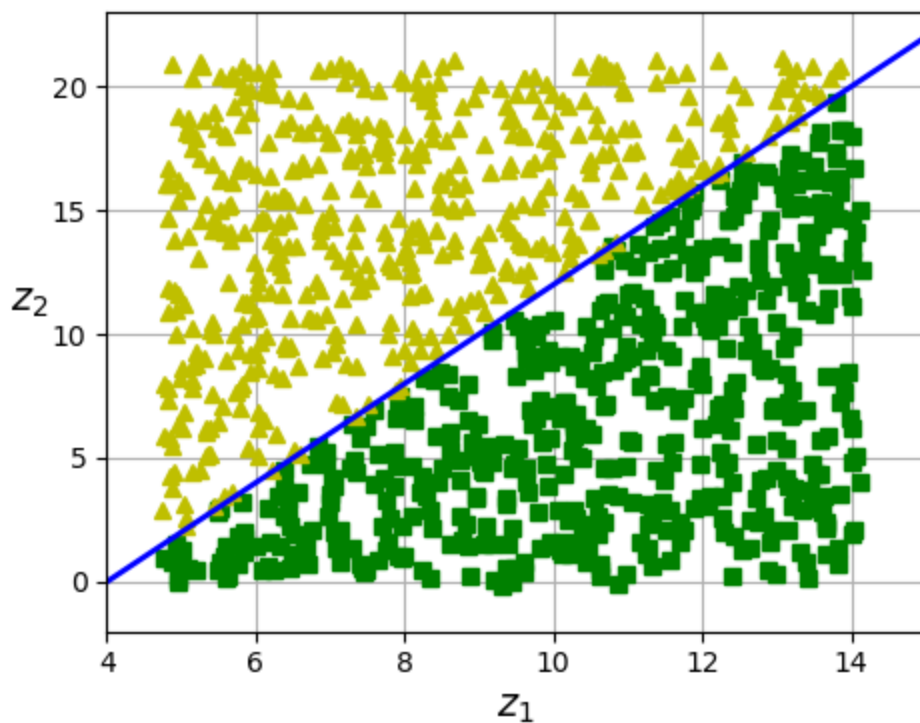
ax.set_zlabel("$x_3$", rotation=0)
ax.set_xlim(axes[0:2])
ax.set_ylim(axes[2:4])
ax.set_zlim(axes[4:6])
save_fig("manifold_decision_boundary_plot3")
plt.show()

fig = plt.figure(figsize=(5, 4))
ax = plt.subplot(1, 1, 1)
ax.plot(t[positive_class], X_swiss[positive_class, 1], "gs")
ax.plot(t[~positive_class], X_swiss[~positive_class, 1], "y^")
ax.plot([4, 15], [0, 22], "b-", linewidth=2)
ax.axis([4, 15, axes[2], axes[3]])
ax.set_xlabel("$z_1$")
ax.set_ylabel("$z_2$", rotation=0, labelpad=8)
ax.grid(True)
save_fig("manifold_decision_boundary_plot4")
plt.show()

```







简而言之，在训练模型之前降低训练集的维数通常会加快训练速度，但这可能并不总是能带来更好或更简单的解决方案；这完全取决于数据集。

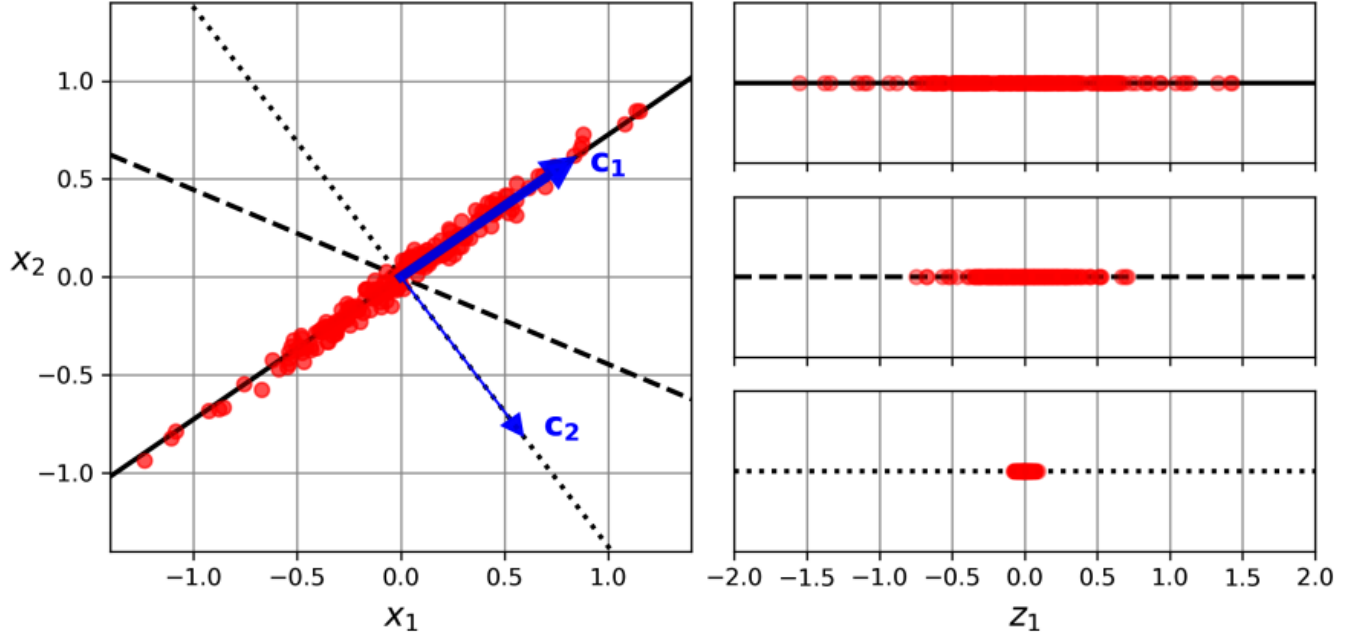
希望你现在能很好地了解维度的诅咒是什么，以及降维算法可以如何对抗它，特别是当流形假设成立时。本章的其余部分将介绍一些最流行的降维算法。

### 3. 主成分分析（Principal component analysis, PCA）

主成分分析（PCA）是目前为止最流行的降维算法。首先，它识别出最接近数据的超平面，然后它将数据投影到它上面。

#### 3.1 保留方差（Preserving the Variance）

在将训练集投影到低维超平面之前，首先需要选择正确的超平面。例如，在下图中的左侧重新显示有一个简单的二维数据集，以及三个不同的轴（即一维超平面）。右边是数据集在每个轴上的投影的结果。正如您所看到的，实线上的投影保留了最大的方差（顶部），而虚线上的投影保留了很少的方差（底部），而虚线上的投影保留了中等数量的方差（中间）。



In [41]: *# extra code - this cell generates and saves Figure 8-7*

```
angle = np.pi / 5
stretch = 5
m = 200

np.random.seed(3)
X_line = np.random.randn(m, 2) / 10
X_line = X_line @ np.array([[stretch, 0], [0, 1]]) # stretch
X_line = X_line @ [[np.cos(angle), np.sin(angle)],
                    [np.sin(angle), np.cos(angle)]] # rotate

u1 = np.array([np.cos(angle), np.sin(angle)])
u2 = np.array([np.cos(angle - 2 * np.pi / 6), np.sin(angle - 2 * np.pi / 6)])
u3 = np.array([np.cos(angle - np.pi / 2), np.sin(angle - np.pi / 2)])

X_proj1 = X_line @ u1.reshape(-1, 1)
X_proj2 = X_line @ u2.reshape(-1, 1)
X_proj3 = X_line @ u3.reshape(-1, 1)

plt.figure(figsize=(8, 4))
plt.subplot2grid((3, 2), (0, 0), rowspan=3)
plt.plot([-1.4, 1.4], [-1.4 * u1[1] / u1[0], 1.4 * u1[1] / u1[0]], "k-",
         linewidth=2)
plt.plot([-1.4, 1.4], [-1.4 * u2[1] / u2[0], 1.4 * u2[1] / u2[0]], "k--",
         linewidth=2)
plt.plot([-1.4, 1.4], [-1.4 * u3[1] / u3[0], 1.4 * u3[1] / u3[0]], "k:",
         linewidth=2)
plt.plot(X_line[:, 0], X_line[:, 1], "ro", alpha=0.5)
plt.arrow(0, 0, u1[0], u1[1], head_width=0.1, linewidth=4, alpha=0.9,
         length_includes_head=True, head_length=0.1, fc="b", ec="b", zorder=10)
plt.arrow(0, 0, u3[0], u3[1], head_width=0.1, linewidth=1, alpha=0.9,
         length_includes_head=True, head_length=0.1, fc="b", ec="b", zorder=10)
plt.text(u1[0] + 0.1, u1[1] - 0.05, r"$\mathbf{c_1}$", color="blue")
plt.text(u3[0] + 0.1, u3[1], r"$\mathbf{c_2}$", color="blue")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$", rotation=0)
plt.axis([-1.4, 1.4, -1.4, 1.4])
plt.grid()

plt.subplot2grid((3, 2), (0, 1))
plt.plot([-2, 2], [0, 0], "k-", linewidth=2)
plt.plot(X_proj1[:, 0], np.zeros(m), "ro", alpha=0.3)
plt.gca().get_yaxis().set_ticks([])
```

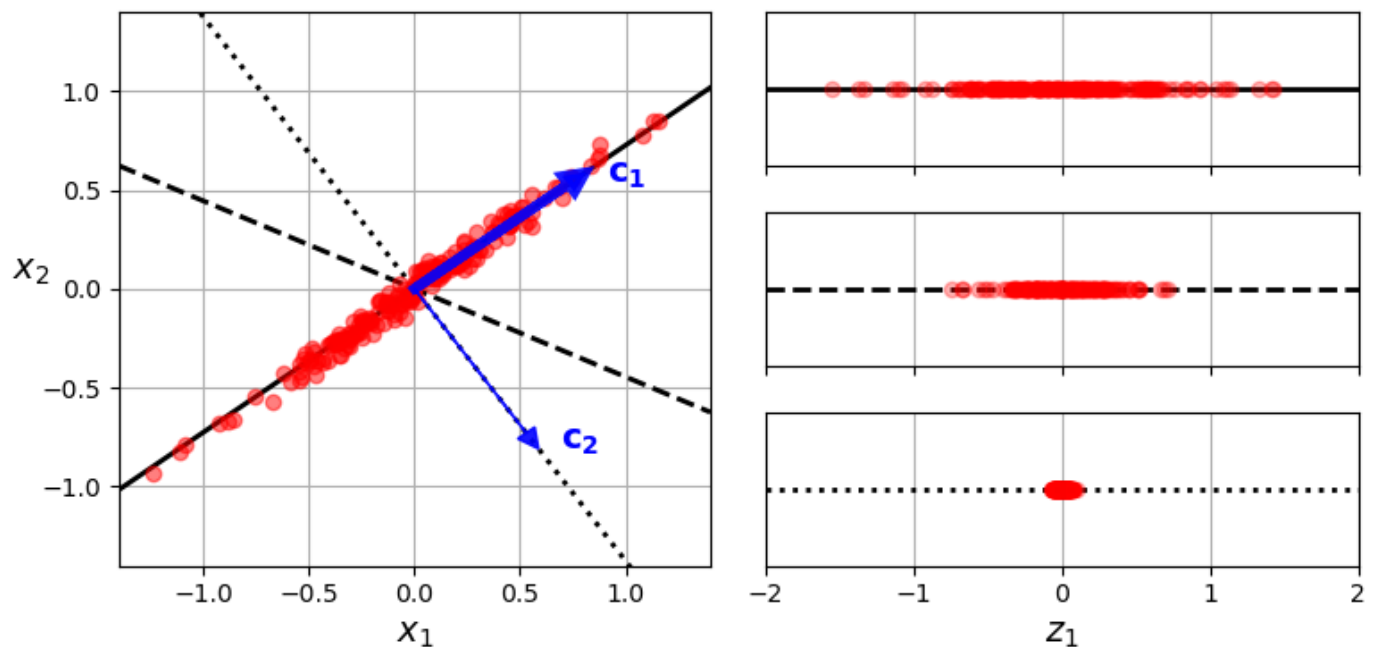


```
plt.gca().get_xaxis().set_ticklabels([])
plt.axis([-2, 2, -1, 1])
plt.grid()

plt.subplot2grid((3, 2), (1, 1))
plt.plot([-2, 2], [0, 0], "k--", linewidth=2)
plt.plot(X_proj2[:, 0], np.zeros(m), "ro", alpha=0.3)
plt.gca().get_yaxis().set_ticks([])
plt.gca().get_xaxis().set_ticklabels([])
plt.axis([-2, 2, -1, 1])
plt.grid()

plt.subplot2grid((3, 2), (2, 1))
plt.plot([-2, 2], [0, 0], "k:", linewidth=2)
plt.plot(X_proj3[:, 0], np.zeros(m), "ro", alpha=0.3)
plt.gca().get_yaxis().set_ticks([])
plt.axis([-2, 2, -1, 1])
plt.xlabel("$z_1$")
plt.grid()

save_fig("pca_best_projection_plot")
plt.show()
```



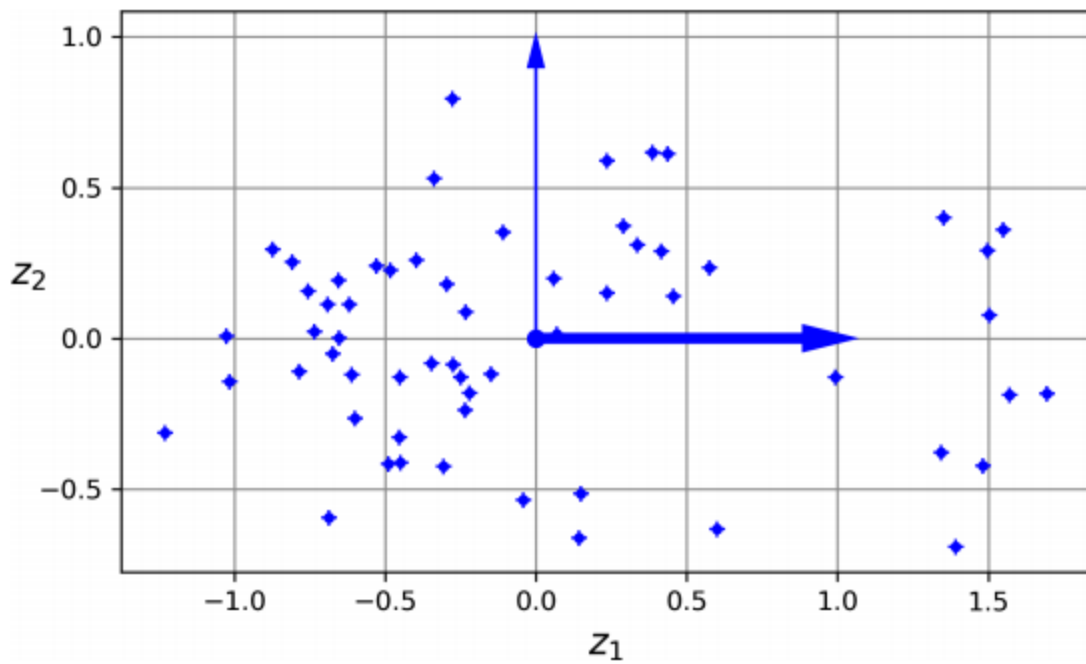
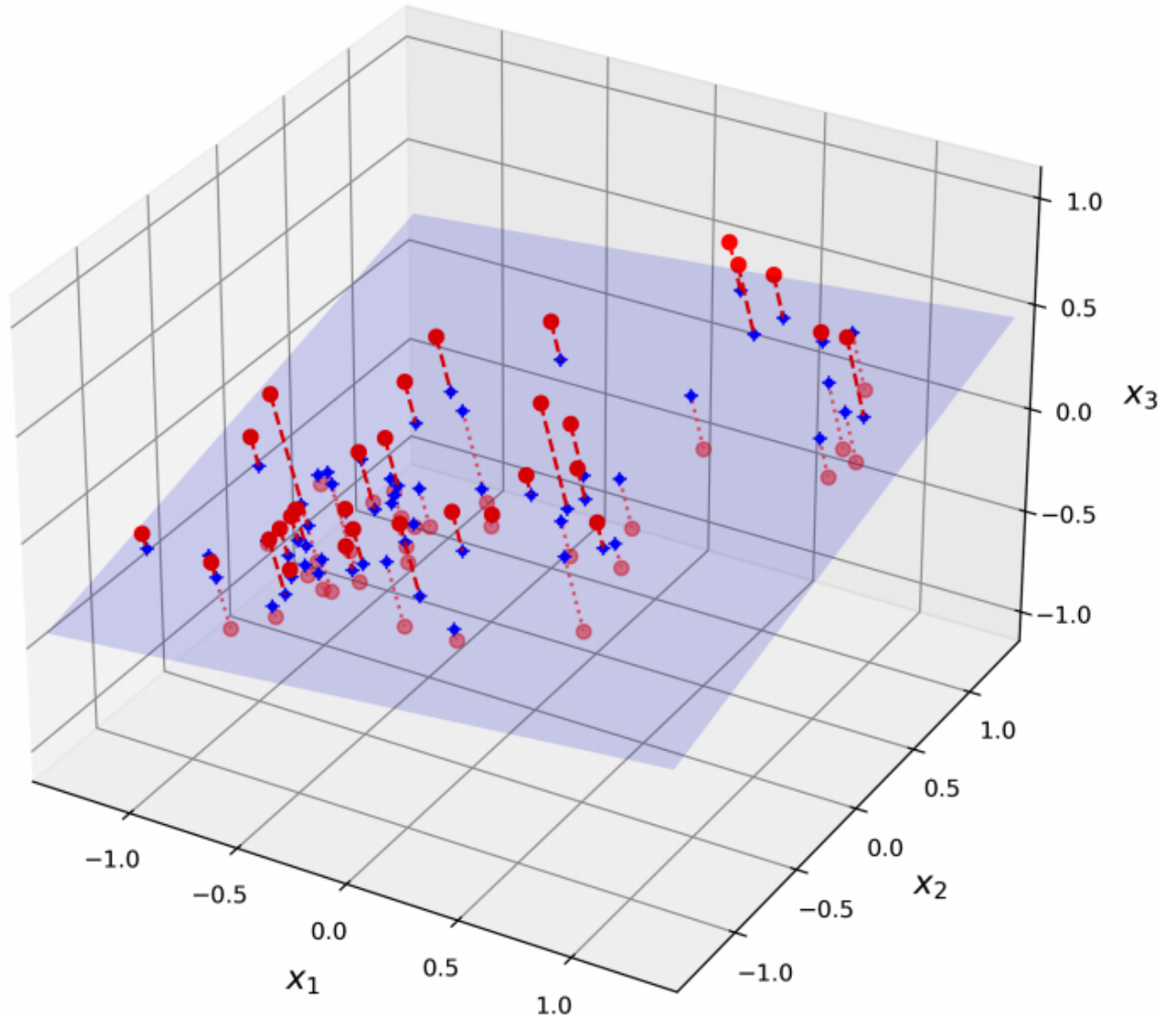
选择保持最大方差的轴似乎是合理的，因为它很可能比其他投影损失更少的信息。另一种证明这种选择的方法是，该轴使原始数据集与在该轴上的投影之间的均方距离最小化。这是PCA背后的一个相当简单的想法。

## 3.2 主成分（Principal Components）

PCA 确定在训练集中占最大方差的轴。在上图中，它是实线所在的轴。它还找到与第一个轴正交的第二个轴，该轴占剩余方差的最大量。在此二维示例中，别无选择：它是虚线。如果它是一个更高维的数据集，PCA 还会找到第三个轴，与之前的两个轴正交，然后是第四个、第五个等等——与数据集中的维数一样多的轴。

第  $i$  个轴称为数据的第  $i$  个主成分（**Principle Component, PC**）。在上图中，第一个 PC 是向量  $c_1$  所在的轴，第二个 PC 是向量  $c_2$  所在的轴。

在下图中，前两个 PC 是在投影平面上的，而第三个 PC 是与该平面正交的轴。投影结束后，在下图中，第一个 PC 对应  $z_1$  轴，第二个 PC 对应  $z_2$  轴。



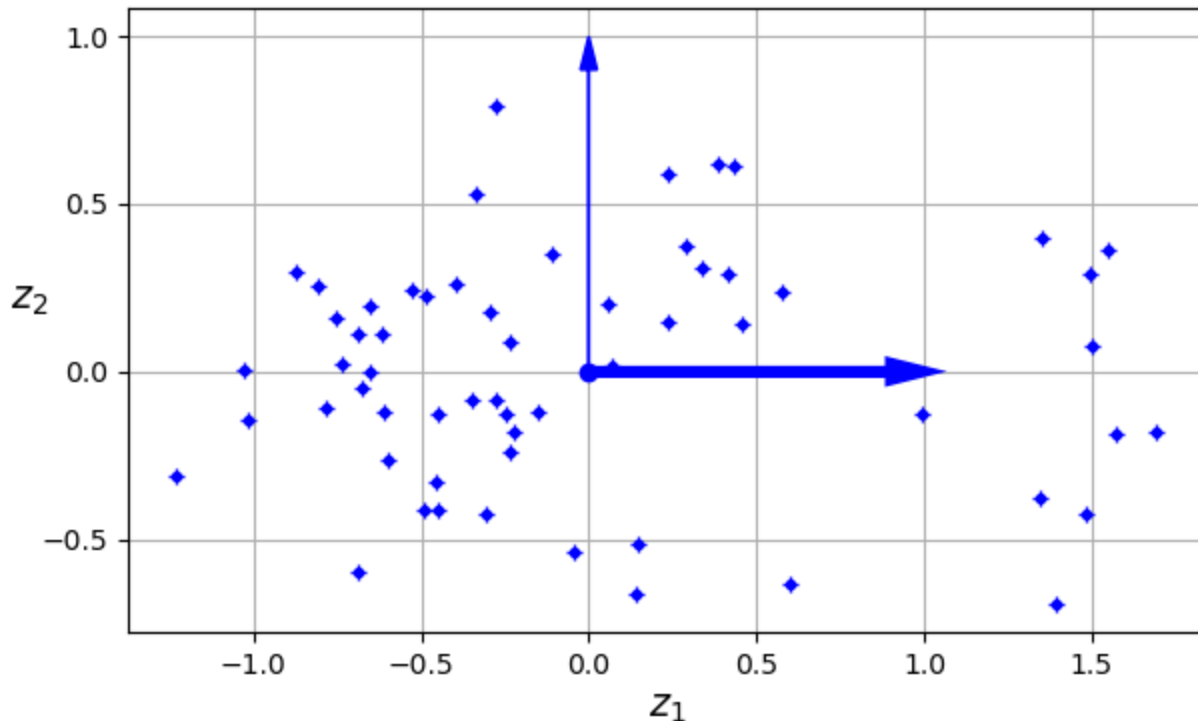
In [42]: *# extra code - this cell generates and saves Figure*

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, aspect='equal')
ax.plot(X2D[:, 0], X2D[:, 1], "b+")
ax.plot(X2D[:, 0], X2D[:, 1], "b.")
ax.plot([0], [0], "bo")
```

```

ax.arrow(0, 0, 1, 0, head_width=0.05, length_includes_head=True,
        head_length=0.1, fc='b', ec='b', linewidth=4)
ax.arrow(0, 0, 0, 1, head_width=0.05, length_includes_head=True,
        head_length=0.1, fc='b', ec='b', linewidth=1)
ax.set_xlabel("$z_1$")
ax.set_yticks([-0.5, 0, 0.5, 1])
ax.set_ylabel("$z_2$", rotation=0)
ax.set_axisbelow(True)
ax.grid(True)
save_fig("dataset_2d_plot")

```



注意：对于每个主成分，PCA 找到一个指向 PC 方向的以零为中心的单位向量。由于两个相反的单位向量位于同一轴上，PCA 返回的单位向量的方向并不稳定：如果稍微扰动训练集并再次运行 PCA，单位向量可能指向与原始向量相反的方向。然而，它们通常仍位于相同的轴上。在某些情况下，一对单位向量甚至可能旋转或交换（如果沿这两个轴的方差非常接近），但它们定义的平面通常会保持不变。

那么，你如何找到一个训练集的主成分呢？幸运的是，有一个标准的矩阵分解技术称为 **奇异值分解**

（**singular value decomposition, SVD**），可以分解训练集矩阵  $\mathbf{X}$  到三个矩阵的矩阵乘法  $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ ，其中  $\mathbf{V}$  包含单位向量，定义所有的主成分，如式所示。

主成分矩阵（**Principal components matrix**）：

$$\mathbf{V} = \begin{bmatrix} | & | & \cdots & | \\ \mathbf{c}_1 & \mathbf{c}_2 & & \mathbf{c}_n \\ | & | & & | \end{bmatrix}$$

下面的 Python 代码使用 NumPy 的 `svd()` 函数来获得下图中所示的 3D 训练集的所有主成分，然后提取定义前两个 PC 的两个单位向量：

```

In [43]: import numpy as np

# X = [...] # the small 3D dataset was created earlier in this notebook
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)

```

```
c1 = Vt[0]
c2 = Vt[1]
```

```
In [44]: c1
```

```
Out[44]: array([0.67857588, 0.70073508, 0.22023881])
```

```
In [46]: c2
```

```
Out[46]: array([-0.72817329, 0.6811147 , 0.07646185])
```

```
In [47]: # extra code - shows how to construct Σ from s
m, n = X.shape
Σ = np.zeros_like(X_centered)
Σ[:,n, :n] = np.diag(s)
assert np.allclose(X_centered, U @ Σ @ Vt)
```

注意：PCA 假设数据集以原点为中心。正如您将看到的，Scikit-Learn 的 PCA 类负责为您居中数据。如果您自己实现 PCA（如前面的示例），或者如果您使用其他库，请不要忘记先将数据居中。

### 3.3 向下投影到 $d$ 维（Projecting Down to $d$ Dimensions）

一旦确定了所有主成分，就可以通过将数据集投影到由前  $d$  个主成分定义的超平面上，将数据集的维数降低到  $d$  维。选择此超平面可确保投影保留尽可能多的方差。

为了将训练集投影到超平面上并获得维度为  $d$  的缩减数据集  $\mathbf{X}_{d-proj}$ ，计算训练集矩阵  $\mathbf{X}$  与矩阵  $\mathbf{W}_d$  的矩阵乘法， $\mathbf{W}_d$  定义为包含  $\mathbf{V}$  的前  $d$  列的矩阵，如公式所示。

将训练集向下投影到  $d$  维：

$$\mathbf{X}_{d-proj} = \mathbf{X}\mathbf{W}_d$$

下面的 Python 代码将训练集投射到由前两个主成分定义的平面上：

```
In [48]: W2 = Vt[:2].T
X2D = X_centered @ W2
```

你知道了！您现在知道如何通过将数据集投影到任意数量的维度来降低它的维数，同时保持尽可能多的方差。

### 3.4 使用 Scikit-Learn（Using Scikit-Learn）

Scikit-Learn 的 PCA 类使用 SVD 来实现 PCA，就像我们在本章前面所做的那样。下面的代码应用 PCA 将数据集的维度降为二个维度（注意，它会自动处理数据中心）：

```
In [67]: from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X2D = pca.fit_transform(X)
```

```
In [68]: X2D
```

```
Out[68]: array([[ -8.73231190e-01,  2.94598030e-01],
 [ 1.48885182e-01, -5.14935573e-01],
 [ 1.35121872e+00,  3.99501548e-01],
```

```

[ 4.54366763e-01,  1.39984497e-01],
[-7.34389086e-01,  2.28934648e-02],
[-2.33347464e-01,  8.67844755e-02],
[-8.08435321e-01,  2.52457557e-01],
[ 1.48281454e+00, -4.22796305e-01],
[ 3.85679006e-01,  6.16229365e-01],
[ 1.54972180e+00,  3.60381563e-01],
[-1.22938760e+00, -3.12504780e-01],
[-4.54653275e-01, -3.28839370e-01],
[ 1.34315899e+00, -3.79446240e-01],
[-6.55233341e-01,  1.92367174e-01],
[-2.49510114e-01, -1.28486810e-01],
[-3.46562831e-01, -8.32312189e-02],
[-6.90221113e-01,  1.13712645e-01],
[-5.29757591e-01,  2.40403321e-01],
[-3.96344855e-01,  2.60334107e-01],
[-6.19519220e-01,  1.13588889e-01],
[ 3.34910399e-01,  3.09476565e-01],
[-4.52441114e-01, -1.28501562e-01],
[-1.02718730e+00,  7.20555799e-03],
[ 2.34128174e-01,  1.50077825e-01],
[-3.37764152e-01,  5.30112382e-01],
[ 1.69474397e+00, -1.82984269e-01],
[-1.09314174e-01,  3.51175914e-01],
[ 4.35623436e-01,  6.12839802e-01],
[ 2.35325731e-01,  5.89871786e-01],
[-4.82837025e-01,  2.25347605e-01],
[ 6.93794782e-02,  1.35642761e-02],
[-6.01101009e-01, -2.66037502e-01],
[-4.49216775e-01, -4.11240529e-01],
[-4.29034522e-02, -5.35785454e-01],
[-6.54088401e-01,  5.63983620e-04],
[ 1.57041965e+00, -1.87726583e-01],
[-6.73463903e-01, -5.05721035e-02],
[-6.86742215e-01, -5.95418050e-01],
[ 5.75586422e-01,  2.34557794e-01],
[-1.49418128e-01, -1.18936809e-01],
[-2.21078895e-01, -1.80920559e-01],
[ 2.89720421e-01,  3.72114568e-01],
[-7.55668778e-01,  1.57349664e-01],
[ 6.00705477e-01, -6.31653062e-01],
[-4.88641270e-01, -4.15788917e-01],
[ 1.49634682e+00,  2.92009077e-01],
[-7.85027564e-01, -1.10347990e-01],
[ 5.79066350e-02,  1.99192616e-01],
[-2.77406262e-01,  7.94401224e-01],
[-2.74485955e-01, -8.72995357e-02],
[-3.05746776e-01, -4.23762463e-01],
[ 1.50190663e+00,  7.78167424e-02],
[ 1.43169044e-01, -6.62805258e-01],
[ 9.92291389e-01, -1.28561421e-01],
[ 4.15074632e-01,  2.88365478e-01],
[ 1.39109949e+00, -6.90805423e-01],
[-2.35063043e-01, -2.37751947e-01],
[-2.95398348e-01,  1.80021529e-01],
[-6.11472315e-01, -1.19651404e-01],
[-1.01712295e+00, -1.42509885e-01]])

```

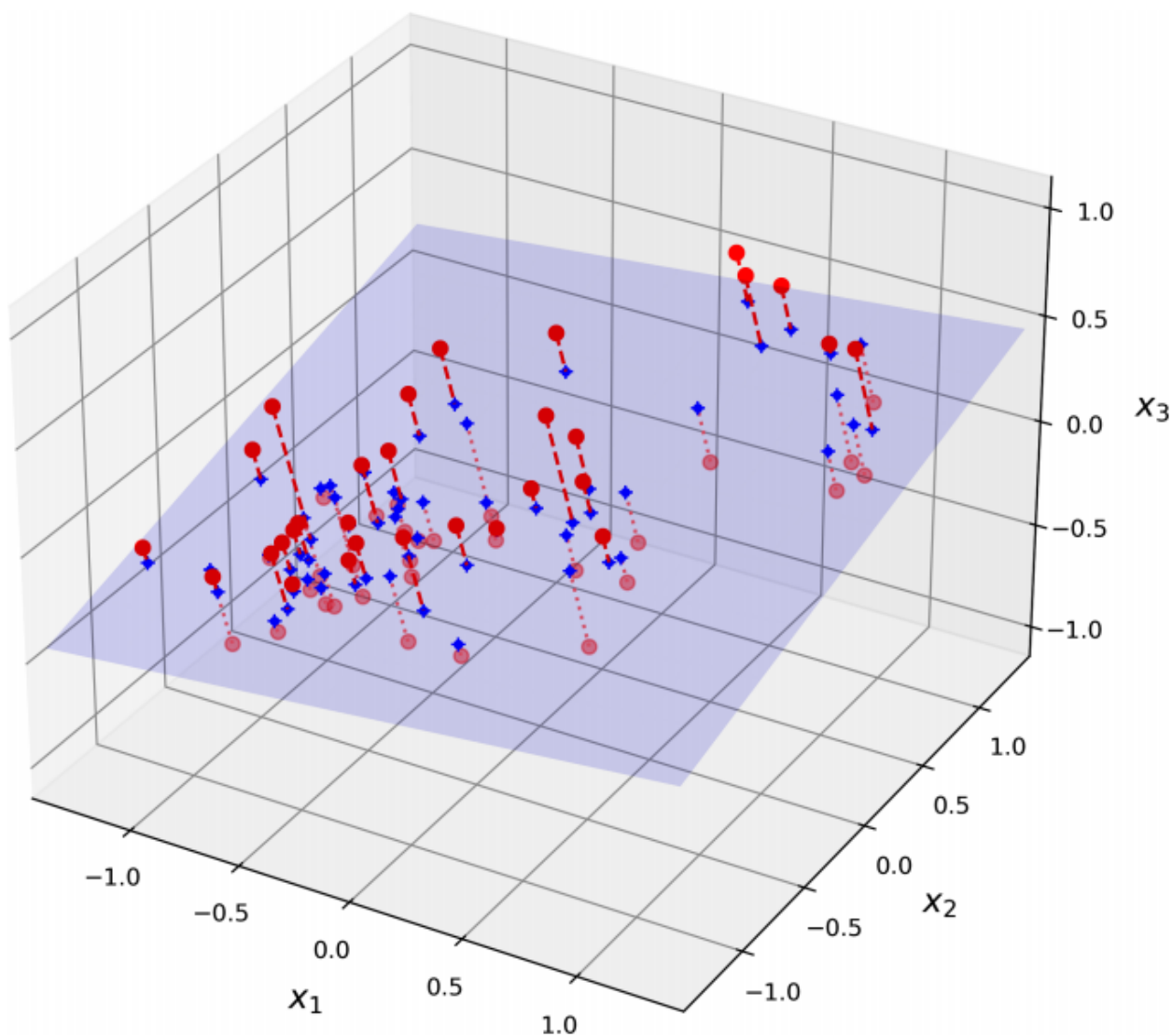
在将 **PCA** 转换器拟合到数据集之后，它的 **components\_** 属性包含  $W_d$  的转置：它为前  $d$  个主成分中的每一个包含一行。

```
In [69]: pca.components_
```

```
Out[69]: array([[ 0.67857588,  0.70073508,  0.22023881],
 [ 0.72817329, -0.6811147 , -0.07646185]])
```

### 3.5 解释方差比（Explained Variance Ratio）

另一个有用的信息是每个主成分的 **解释方差比（explained variance ratio）**，可通过 `explained_varianceratio` 变量获得。该比率表示沿每个主成分的数据集方差的比例。



例如，让我们看一下上图表示的 3D 数据集的前两个分量的解释方差比：

```
In [70]: pca.explained_variance_ratio_  
Out[70]: array([0.7578477 , 0.15186921])
```

这个输出告诉我们，大约 76% 的数据集的方差位于第一个 PC，约 15% 位于第二个 PC。这给第三个 PC 剩下的大约 9%，所以我们有理由假设第三个 PC 可能携带很少的信息。

### 3.6 选择正确的维数（Choosing the Right Number of Dimensions）

与其任意选择要减少到的维数，不如选择加起来足够大的方差部分的维数更简单——比如 95%（当然，这个规则的一个例外是，如果您正在降低数据可视化的维度，在这种情况下，您将希望将维度降低到 2 或 3）。

下面的代码加载并分割 MNIST 数据集（在第 3 章中介绍），并在不降维的情况下执行 PCA，然后计算保留训练集 95% 方差所需的最小维数：



```
In [71]: from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)
X_train, y_train = mnist.data[:60_000], mnist.target[:60_000]
X_test, y_test = mnist.data[60_000:], mnist.target[60_000:]

pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1 # d equals 154

d
```

Out[71]: 154

然后您可以设置 **n\_components=d** 并再次运行 **PCA**，但还有更好的选择。您可以将 **n\_components** 设置为介于 0.0 和 1.0 之间的浮点数，而不是指定要保留的主成分的数量，指示您希望保留的方差比：

```
In [72]: pca = PCA(n_components=0.95)

X_reduced = pca.fit_transform(X_train)
```

成分的实际数量将在训练过程中确定，并存储在 **ncomponents** 属性中：

```
In [74]: pca.n_components_
```

Out[74]: 154

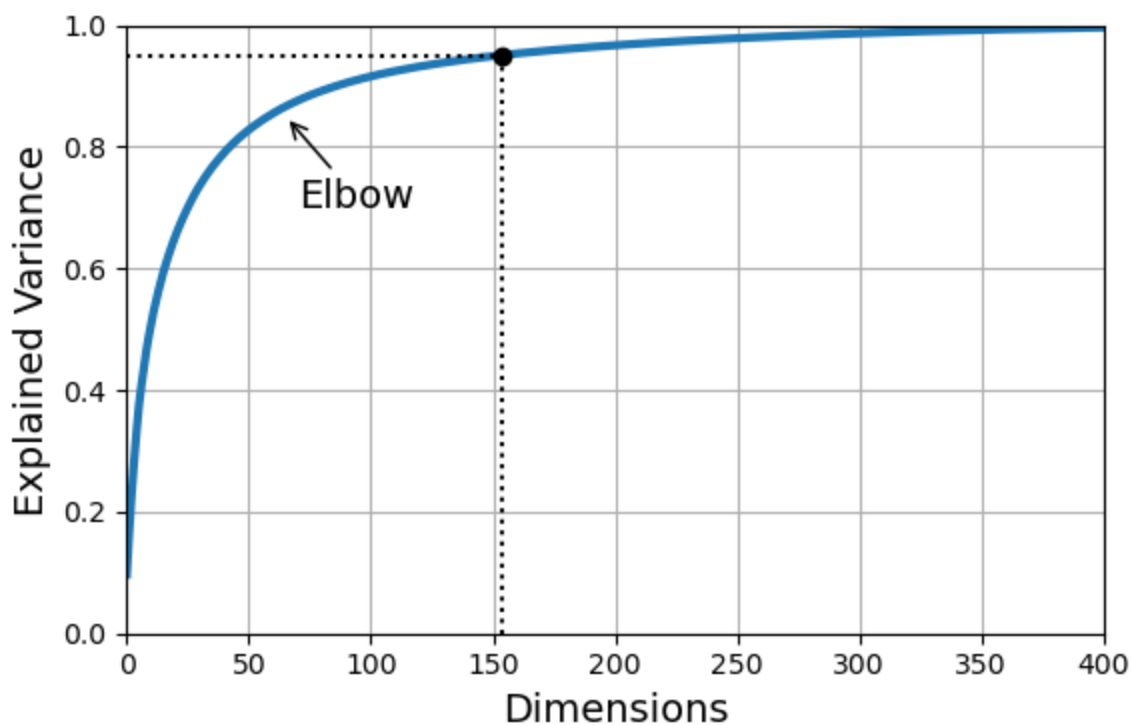
```
In [75]: pca.explained_variance_ratio_.sum() # extra code
```

Out[75]: 0.9501960192613033

另一种选择是将解释的方差绘制为维数的函数（简单地绘制 **cumsum**）。曲线中通常会有一个瓶颈，是解释方差停止快速增长的地方。在这种情况下，您可以看到将维度减少到大约 100 维也不会损失太多解释方差。

```
In [76]: # extra code - this cell generates and saves Figure
```

```
plt.figure(figsize=(6, 4))
plt.plot(cumsum, linewidth=3)
plt.axis([0, 400, 0, 1])
plt.xlabel("Dimensions")
plt.ylabel("Explained Variance")
plt.plot([d, d], [0, 0.95], "k:")
plt.plot([0, d], [0.95, 0.95], "k:")
plt.plot(d, 0.95, "ko")
plt.annotate("Elbow", xy=(65, 0.85), xytext=(70, 0.7),
            arrowprops=dict(arrowstyle="->"))
plt.grid(True)
save_fig("explained_variance_plot")
plt.show()
```



最后，如果您使用降维作为监督学习任务（例如，分类）的预处理步骤，那么您可以像调整任何其他超参数一样调整维数（参见第2章）。例如，下面的代码示例创建了一个两步 **pipeline**，首先使用 **PCA** 进行降维，然后使用随机森林进行分类。接下来，它使用 **RandomizedSearchCV** 来找到 **PCA** 和随机森林分类器超参数的良好组合。这个示例进行了快速搜索，只调优2个超参数，只对1000个实例进行训练，只运行10次迭代，但是如果您有时间，请随意进行更彻底的搜索：

```
In [77]: from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import make_pipeline

clf = make_pipeline(PCA(random_state=42),
                    RandomForestClassifier(random_state=42))

param_distrib = {
    "pca__n_components": np.arange(10, 80),
    "randomforestclassifier__n_estimators": np.arange(50, 500)
}

rnd_search = RandomizedSearchCV(clf, param_distrib, n_iter=10, cv=3,
                                random_state=42)

rnd_search.fit(X_train[:1000], y_train[:1000])
```

```
Out[77]: RandomizedSearchCV
          estimator: Pipeline
              └─ PCA
                  └─ RandomForestClassifier
```

让我们来看看所找到的最佳超参数：

```
In [79]: rnd_search.best_params_

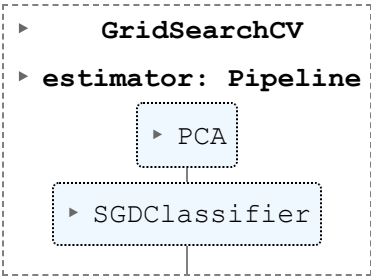
Out[79]: {'randomforestclassifier__n_estimators': 465, 'pca__n_components': 23}
```

值得注意的是，最佳成分数量有多低：我们将 784 维数据集减少到仅 23 维！这与我们使用随机森林这一事实有关，这是一个非常强大的模型。如果我们改用线性模型，例如 `SGDClassifier`，搜索会发现我们需要保留更多维度（大约70）。

```
In [81]: from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import GridSearchCV

clf = make_pipeline(PCA(random_state=42), SGDClassifier())
param_grid = {"pca__n_components": np.arange(10, 80)}
grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X_train[:1000], y_train[:1000])
```

```
Out[81]:
```



```
In [82]: grid_search.best_params_
```

```
Out[82]: {'pca__n_components': 67}
```

### 3.7 PCA 压缩（PCA for Compression）

经过降维后，训练集所占用的空间要少得多。例如，在将 PCA 应用于 MNIST 数据集后，同时保留了其 95% 的方差后，我们只剩下154个特征，而不是原来的784个特征。所以这个数据集现在还不到它原来大小的 20%，我们只失去了 5% 的方差！这是一个合理的压缩比，并且很容易看到这样的大小减少将如何极大地加速分类算法。

通过应用 PCA 投影的逆变换，也可以将简化的数据集解压缩为 784 维。这不会给你原始数据，因为投影丢失了一些信息（在删除的5%方差内），但它可能接近原始数据。原始数据与重构数据（压缩后解压缩）之间的均方距离称为 重构误差(reconstruction error)。

`inverse_transform()` 方法允许我们将简化的 MNIST 数据集解压缩回784维：

```
In [83]: pca = PCA(0.95)

X_reduced = pca.fit_transform(X_train, y_train)
```

```
In [84]: X_recovered = pca.inverse_transform(X_reduced)
```

下图显示了来自原始训练集（左侧）的一些数字，以及压缩和解压后的相应数字。你可以看到，有一些轻微的图像质量损失，但数字仍然基本完好无损。

```
In [85]: # extra code - this cell generates and saves Figure 8-9

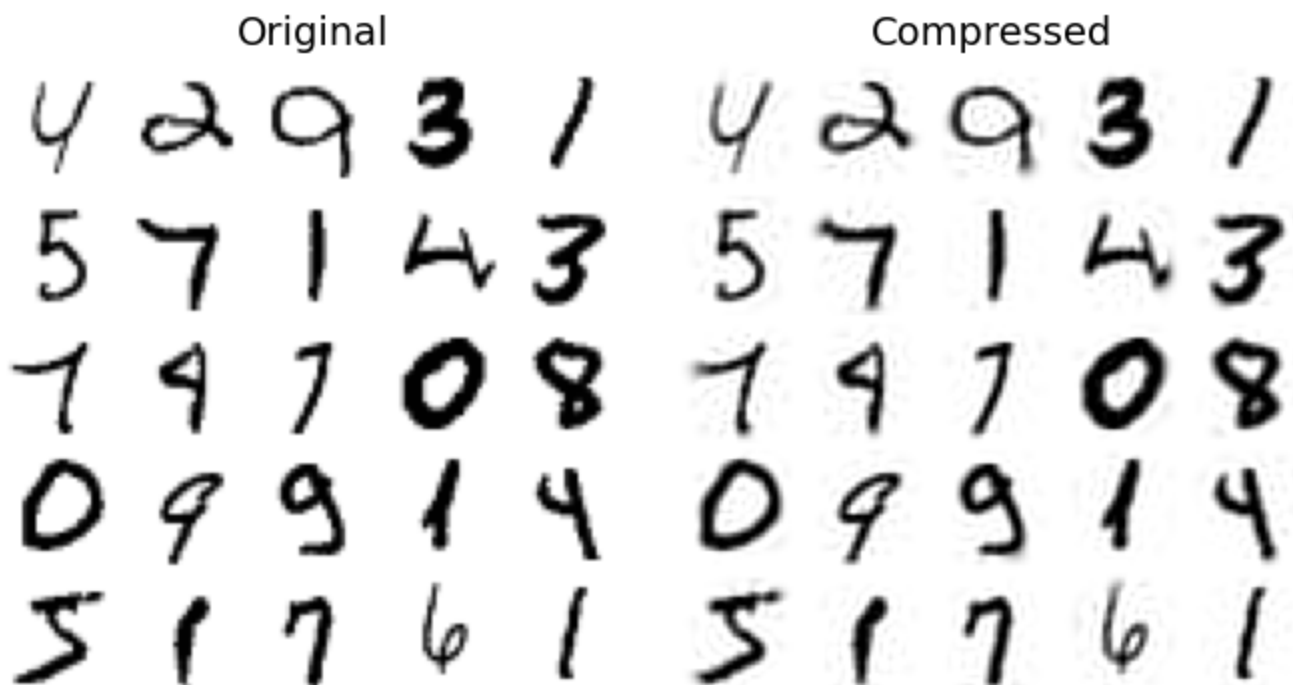
plt.figure(figsize=(7, 4))
for idx, X in enumerate((X_train[:2100], X_recovered[:2100])):
    plt.subplot(1, 2, idx + 1)
    plt.title(["Original", "Compressed"][idx])
    for row in range(5):
        for col in range(5):
            plt.imshow(X[row * 5 + col].reshape(28, 28), cmap="binary",
```

```

        vmin=0, vmax=255, extent=(row, row + 1, col, col + 1))
plt.axis([0, 5, 0, 5])
plt.axis("off")

save_fig("mnist_compression_plot")

```



**PCA** 逆变换回到原来的维数:

$$\mathbf{X}_{recovered} = \mathbf{X}_{d-proj} \mathbf{W}_d^T$$

### 3.8 随机 PCA（Randomized PCA）

如果你将 **svd\_solver** 超参数设置为“**randomized**”，Scikit-Learn 会使用一种称为 随机主成分分析（**randomized PCA**）的随机算法，该算法可以快速找到前  $d$  个主成分的近似值。它的计算复杂度是  $O(m \times d^2) + O(d^3)$ ，而不是  $O(m \times n^2) + O(n^3)$ ，所以当  $d$  远小于  $n$  时，它比完全 SVD 快得多：

```

In [86]: rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)

X_reduced = rnd_pca.fit_transform(X_train)

```

注意：默认情况下，**svd\_solver** 实际上被设置为“**auto**”：如果  $\max(m,n) > 500$  并且 **n\_components** 是一个小于  $\min(m,n)$  的80%的整数，Scikit-Learn 自动使用随机PCA算法，否则它使用完整的SVD方法。因此，前面的代码将使用随机化的 PCA 算法，即使你删除了 **svd\_solver="randomized"** 参数，因为  $154 < 0.8 \times 784$ 。如果您想强制Scikit-Learn 使用全 SVD 以获得稍微更精确的结果，您可以将 **svd\_solver** 超参数设置为“**full**”。

### 3.9 增量 PCA（Incremental PCA）

前面的 PCA 实现的一个问题是，它们需要整个训练集适合内存，以便使算法运行。幸运的是，增量 **PCA（IPCA）** 算法已经开发出来，允许您将训练集分成小批，并一次在一个小批中提供这些数据。这对于大型训练集和在线应用 **PCA** 都很有用。

以下代码将 MNIST 训练集分成 100 个小批次（使用 NumPy 的 **array\_split()** 函数）并将它们提供给 Scikit-Learn 的 **IncrementalPCA** 类，以将 MNIST 数据集的维数减少到 154 维，就像以前一样。请注意，您必须对

每个小批量调用 **partial\_fit()** 方法，而不是对整个训练集调用 **fit()** 方法：

```
In [87]: from sklearn.decomposition import IncrementalPCA

n_batches = 100

inc_pca = IncrementalPCA(n_components=154)

for X_batch in np.array_split(X_train, n_batches):

    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

或者，您可以使用 NumPy 的 **memmap** 类，它允许您操作存储在磁盘上二进制文件中的大型数组，就好像它完全在内存中一样；该类仅在需要时将其需要的数据加载到内存中。为了演示这一点，让我们首先创建一个内存映射 (memmap) 文件并将 MNIST 训练集复制到其中，然后调用 **flush()** 以确保仍在缓存中的所有数据都保存 to 磁盘。在现实生活中，**X\_train** 通常不适合内存，因此您将逐块加载它并将每个块保存到 **memmap** 数组的正确部分：

```
In [88]: filename = "my_mnist.mmap"

X_mmap = np.memmap(filename, dtype='float32', mode='write', shape=X_train.shape)

X_mmap[:] = X_train  # could be a loop instead, saving the data chunk by chunk

X_mmap.flush()
```

接下来，我们可以加载 **memmap** 文件，并像使用常规的 NumPy 数组一样使用它。让我们使用 **IncrementalPCA** 类来降低其维数。由于该算法在任何给定时间只使用数组的一小部分，内存使用仍在控制中。这使得可以调用通常的 **fit()** 方法，而不是 **partial\_fit()**，这非常方便：

```
In [89]: X_mmap = np.memmap(filename, dtype="float32", mode="readonly").reshape(-1, 784)

batch_size = X_mmap.shape[0] // n_batches

inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)

inc_pca.fit(X_mmap)
```

```
Out[89]: ▼ IncrementalPCA
IncrementalPCA(batch_size=600, n_components=154)
```

注意：只有原始的二进制数据被保存到磁盘上，因此在加载它时需要指定数组的数据类型和形状。如果您省略了该形状，则 **np.memmap()** 将返回一个一维数组。

对于非常高维的数据集，PCA 可能速度太慢了。正如您前面看到的，即使您使用随机 PCA，它的计算复杂度仍然是  $O(m \times d^2) + O(d^3)$ ，所以维数  $d$  的目标数不能太大。如果您处理的数据集具有数万或更多特征（例如图像）的数据集，那么训练可能会变得太慢：在这种情况下，您应该考虑使用随机投影。

## 4. 随机投影（Random Projection）

顾名思义，随机投影算法使用随机线性投影将数据投影到一个低维空间。这听起来可能很疯狂，但事实证明，这样的随机投影实际上很可能很好地保持距离，正如 William B. Johnson 和 Joram Lindenstrauss 在一个

著名的引理中在数学上所证明的那样。因此，两个相似的实例在投影后将保持相似，而两个非常不同的实例将保持非常不同。

显然，你下降的维度越多，丢失的信息就越多，距离扭曲的也就越多。那么如何选择最佳维数呢？Johnson 和 Lindenstrauss 想出了一个方程式来确定要保留的最小维数，以确保（很有可能）距离的变化不会超过给定的公差。例如，如果您的数据集包含  $m = 5000$  个实例，每个实例具有  $n = 20000$  个特征，并且您不希望任何两个实例之间的平方距离变化超过  $\epsilon = 10\%$ ，那么您应该向下投影数据到  $d$  维，其中  $d \geq 4\log(m)/(\frac{1}{2}\epsilon^2 - \frac{1}{3}\epsilon^3)$ ，即 7300 维。这是一个相当显著的降维！请注意，方程式不使用  $n$ ，它仅依赖于  $m$  和  $\epsilon$ 。该等式由 `johnson_lindenstrauss_min_dim()` 函数实现：

```
In [93]: from sklearn.random_projection import johnson_lindenstrauss_min_dim

m, ε = 5_000, 0.1

d = johnson_lindenstrauss_min_dim(m, eps=ε)

d
```

Out[93]: 7300

```
In [94]: # extra code - show the equation computed by johnson_lindenstrauss_min_dim

d = int(4 * np.log(m) / (ε ** 2 / 2 - ε ** 3 / 3))

d
```

Out[94]: 7300

现在我们可以生成一个形为  $[d, n]$  的随机矩阵  $\mathbf{P}$ ，其中每个项都从一个均值为 0、方差为  $1/d$  的高斯分布中随机抽样，并使用它从  $n$  维投影到  $d$  维：

```
In [95]: n = 20_000
np.random.seed(42)
P = np.random.randn(d, n) / np.sqrt(d) # std dev = square root of variance

X = np.random.randn(m, n) # generate a fake dataset
X_reduced = X @ P.T
```

仅此而已！它简单而有效，而且不需要训练：算法创建随机矩阵唯一需要的就是数据集的形状。数据本身根本不被使用。

Scikit-Learn 提供了一个 **GaussianRandomProjection** 类来做我们刚刚做的事情：当你调用它的 `fit()` 方法时，它使用 `johnson_lindenstrauss_min_dim()` 来确定输出维度，然后它生成一个随机矩阵，它存储在 `components_` 属性中。然后当你调用 `transform()` 时，它使用这个矩阵来执行投影。创建转换器时，如果要调整  $\epsilon$ （默认为 0.1），则可以设置 `eps`，如果要强制使用特定的目标维度  $d$ ，则可以设置 `n_components`。下面的代码示例给出了与前面代码相同的结果（您还可以验证 `gaussian_rndproj.components` 是否等于  $\mathbf{P}$ ）：

```
In [96]: from sklearn.random_projection import GaussianRandomProjection

gaussian_rnd_proj = GaussianRandomProjection(eps=ε, random_state=42)

X_reduced = gaussian_rnd_proj.fit_transform(X) # same result as above
```

Scikit-Learn 还提供了第二个随机投影转换器，称为 **SparseRandomProjection**。它以相同的方式确定目标维数，生成相同形状的随机矩阵，并逐步进行方向投影。主要的区别在于随机矩阵是稀疏的。这意味着它使



用的内存少得多：大约 25 MB，而不是之前的测试中几乎 1.2GB！它也要快得多，生成随机矩阵和降维数都要快得多：在这种情况下大约快 50%。此外，如果输入是稀疏的，则转换将使其保持稀疏（除非您设置 **dense\_output=True**）。最后，它具有与以往的方法相同的保持距离特性，且降维的质量具有可比性。简而言之，通常最好使用这个转换器而不是第一个转换器，特别是对于大型或稀疏的数据集。

稀疏随机矩阵中非零项的比例  $r$  称为其 **密度（density）**。默认情况下，它等于  $1/\sqrt{n}$ 。有 20000 个特征，这意味着随机矩阵中 141 单元中只有 1 个是非零的：这是相当稀疏的！如果您愿意，您可以将 **density** 超参数设置为另一个值。稀疏随机矩阵中的每个单元格都有一个概率  $r$  是非零的，每个非零值是  $-v$  或  $+v$ （两者都等可能），其中  $v = 1/\sqrt{dr}$ 。

如果您想执行逆变换，您首先需要使用 SciPy 的 **pinv()** 函数计算分量矩阵的伪逆，然后将简化的数据乘以伪逆的转置：

```
In [97]: components_pinv = np.linalg.pinv(gaussian_rnd_proj.components_)

X_recovered = X_reduced @ components_pinv.T
```

```
In [98]: # extra code - performance comparison between Gaussian and Sparse RP

from sklearn.random_projection import SparseRandomProjection

print("GaussianRandomProjection fit")
%timeit GaussianRandomProjection(random_state=42).fit(X)
print("SparseRandomProjection fit")
%timeit SparseRandomProjection(random_state=42).fit(X)

gaussian_rnd_proj = GaussianRandomProjection(random_state=42).fit(X)
sparse_rnd_proj = SparseRandomProjection(random_state=42).fit(X)
print("GaussianRandomProjection transform")
%timeit gaussian_rnd_proj.transform(X)
print("SparseRandomProjection transform")
%timeit sparse_rnd_proj.transform(X)
```

```
GaussianRandomProjection fit
3.25 s ± 37.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
SparseRandomProjection fit
2.3 s ± 43.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
GaussianRandomProjection transform
6.78 s ± 60.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
SparseRandomProjection transform
4.44 s ± 75.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

注意：如果分量矩阵较大，计算伪逆可能需要很长时间，如果  $d < n$ ，**pinv()** 的计算复杂度为  $O(dn^2)$ ，否则为  $O(nd^2)$ 。

总之，随机投影是一种简单、快速、内存高效、功能异常强大的降维算法，您应该记住它，特别是在处理高维数据集时。

注意：随机投影并不总是用于降低大型数据集的维度。例如，Sanjoy Dasgupta 等人在 2017 年发表的一篇文章。表明果蝇的大脑实现了随机投影的模拟，将密集的低维嗅觉输入映射到稀疏的高维二进制输出：对于每种气味，只有一小部分输出神经元被激活，但相似的气味会激活许多相同的神经元。这类似于一种称为 **局部敏感哈希（locality sensitive hashing, LSH）** 的著名算法，该算法通常用于搜索引擎中以对相似文档进行分组。

## 5. LLE

局部线性嵌入（**Locally linear embedding, LLE**）是一种 非线性降维（**nonlinear dimensionality reduction, NLDR**）技术。它是一种不依赖于投影的流形学习技术，不同于 PCA 和 随机投影。简而言之，**LLE** 的工作原理是首先测量每个训练实例与其最近邻居的线性关联，然后寻找训练集的低维表示，其中这些局部关系得到最佳保留（更多细节）。这种方法使它特别擅长展开扭曲的流形，特别是当没有太多的噪声时。

下面的代码制作了一个瑞士卷，然后使用 Scikit-Learn 的 **LocallyLinearEmbedding** 类来展开它：

```
In [99]: from sklearn.datasets import make_swiss_roll
from sklearn.manifold import LocallyLinearEmbedding

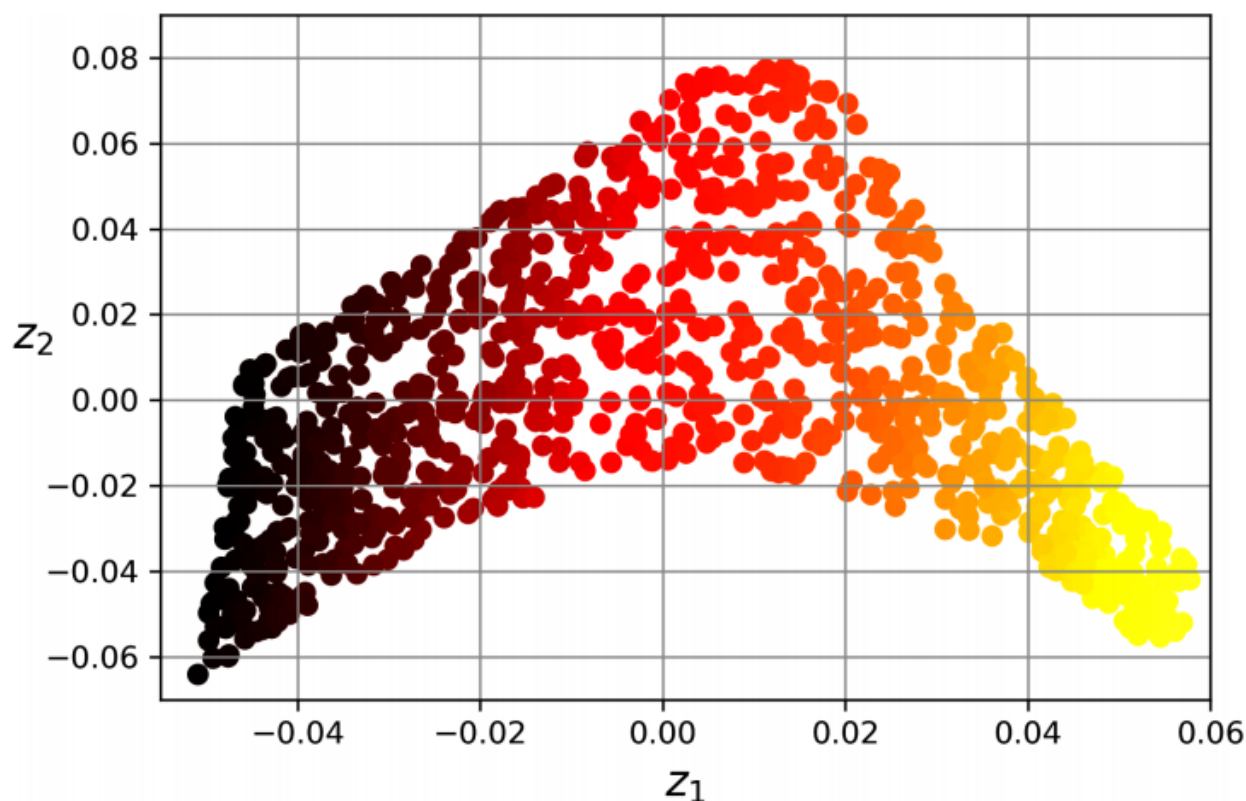
X_swiss, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10, random_state=42)

X_unrolled = lle.fit_transform(X_swiss)
```

变量  $t$  是一个 1D NumPy 数组，包含每个实例沿瑞士卷的滚动轴的位置。我们在这个例子中不使用它，但它可以被用作非线性回归任务的目标。

所得到的 2D 数据集如下图所示。正如您所看到的，瑞士卷已经完全展开，实例之间的距离在本地得到了很好的保存。

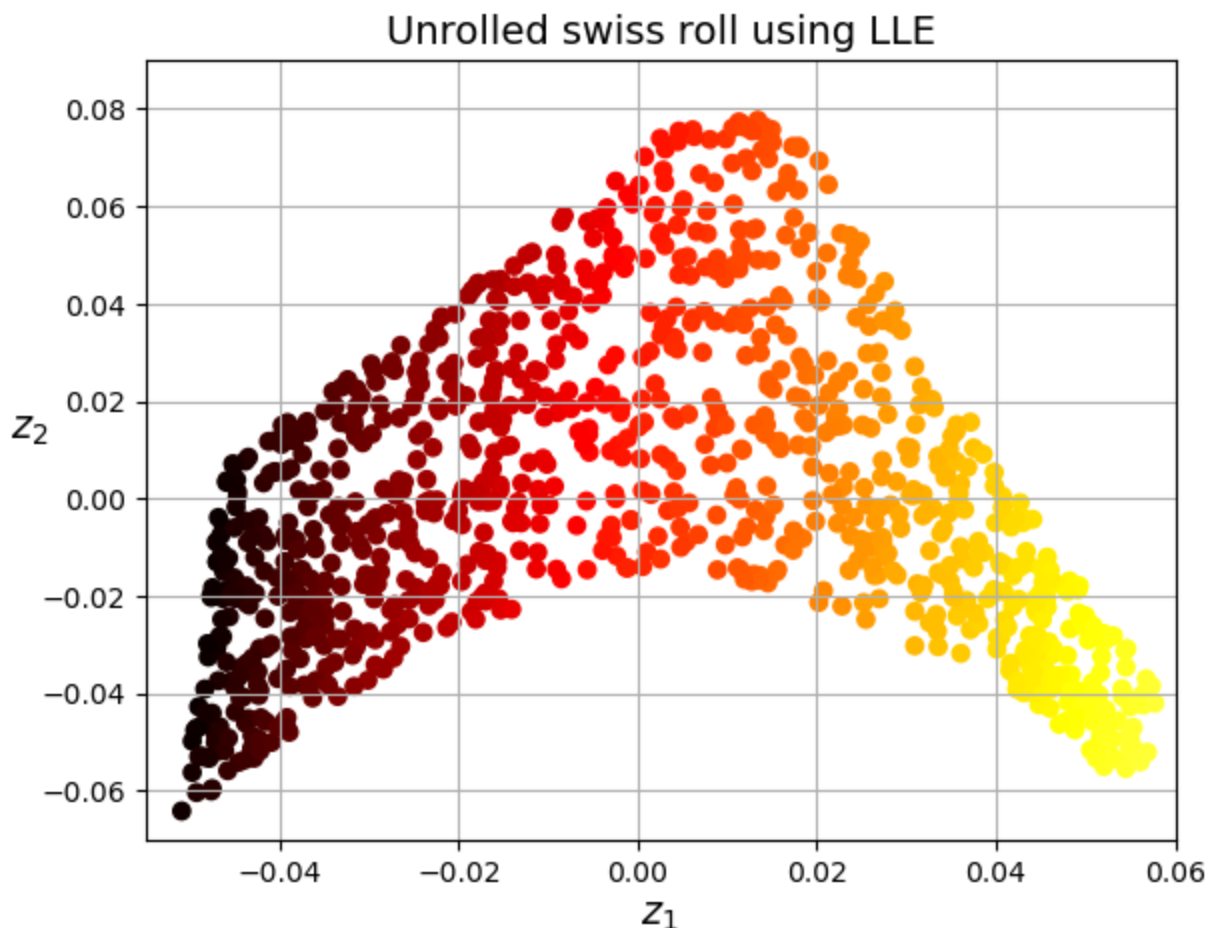


```
In [100... # extra code - this cell generates and saves Figure 8-10

plt.scatter(X_unrolled[:, 0], X_unrolled[:, 1],
            c=t, cmap=darker_hot)
plt.xlabel("$z_1$")
plt.ylabel("$z_2$", rotation=0)
plt.axis([-0.055, 0.060, -0.070, 0.090])
plt.grid(True)

save_fig("lle_unrolling_plot")
```

```
plt.title("Unrolled swiss roll using LLE")
plt.show()
```



然而，距离并没有保留在更大的范围内：展开的瑞士卷应该是一个矩形，而不是这种拉伸和扭曲的条带。尽管如此，LLE 在为流形建模方面做得很好。

LLE 的工作原理如下：对于每个训练实例  $\mathbf{x}^{(i)}$ ，该算法识别其  $k$  最近邻（在上面的代码中  $k = 10$ ），然后尝试将  $\mathbf{x}^{(i)}$  重建为这些邻居的线性函数。更具体地说，它试图找到权重  $w_{i,j}$  使得  $\mathbf{x}^{(i)}$  和  $\sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)}$  之间的平方距离尽可能小，假设  $w_{i,j} = 0$  如果  $\mathbf{x}^{(j)}$  不是  $\mathbf{x}^{(i)}$  的  $k$  最近邻。因此，LLE 的第一步是如下方程中描述的约束优化问题，其中  $\mathbf{W}$  是包含所有权重  $w_{i,j}$  的权重矩阵。第二个约束简单地归一化每个训练实例  $\mathbf{x}^{(i)}$  的权重。

**LLE 步骤1：线性建模局部关系**

$$\begin{aligned} \hat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \quad & \sum_{i=1}^m \left( \mathbf{x}^{(i)} - \sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)} \right)^2 \\ \text{subject to} \quad & \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ n.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^m w_{i,j} = 1 & \text{for } i = 1, 2, \dots, m \end{cases} \end{aligned}$$

在此步骤之后，权重矩阵  $\hat{\mathbf{W}}\mathbf{W}$ （包含权重  $w_{i,j}$ ）对训练实例之间的局部线性关系进行编码。第二步是将训练实例映射到  $d$  维空间（其中  $d < n$ ），同时尽可能保留这些局部关系。如果  $\mathbf{z}^{(i)}$  是  $\mathbf{x}^{(i)}$  在这个  $d$  维空间中的

映射，那么我们希望  $\mathbf{z}^{(i)}$  和  $\sum_{j=1}^m \hat{w}_{i,j} \mathbf{z}^{(j)}$  的平方距离越小越好。这个想法导致如下公式中描述的无约束优化问题。它看起来与第一步非常相似，但我们不是保持实例固定并找到最佳权重，而是相反：保持权重固定并找到实例图像在低维空间中的最佳位置。注意  $\mathbf{Z}$  是包含所有  $\mathbf{z}^{(i)}$  的矩阵。

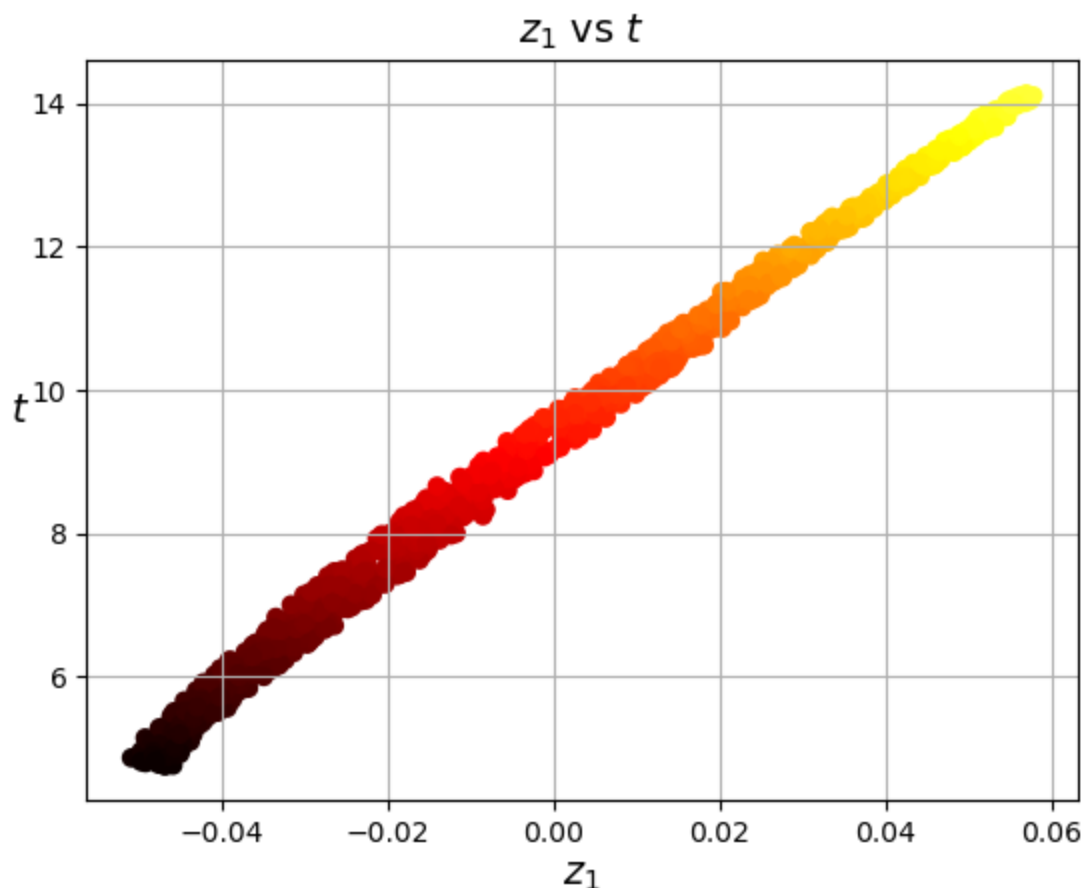
**LLE 步骤2：**降低维度，同时保持关系

$$\hat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^m \left( \mathbf{z}^{(i)} - \sum_{j=1}^m \hat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

Scikit-Learn 的 LLE 实现具有以下计算复杂度：  $O(m \log(m)n \log(k))$  用于寻找  $k$  个最近邻，  $O(mnk^3)$  用于优化权值，  $O(dm^2)$  用于构造低维表示。不幸的是，上一项的  $m^2$  使得该算法难以适应非常大的数据集。

正如您所看到的，LLE 与投影技术有很大的不同，而且它非常复杂，但它也可以构造更好的低维表示，特别是如果数据是非线性的。

```
In [101... # extra code - shows how well correlated z1 is to t: LLE worked fine
plt.title("$z_1$ vs $t$")
plt.scatter(X_unrolled[:, 0], t, c=t, cmap=darker_hot)
plt.xlabel("$z_1$")
plt.ylabel("$t$", rotation=0)
plt.grid(True)
plt.show()
```



## 6. 其他降维技术（Other Dimensionality Reduction Techniques）

在我们结束这一章之前，让我们快速看看Scikit-Learn中提供一些其他流行的降维技术：

- `sklearn.manifold.MDS`：多维缩放（**Multidimensional scaling, MDS**）降低了维数，同时试图保持实例之间的距离。随机投影对高维数据是这样的，但对低维数据效果不有效。
- `sklearn.manifold.Isomap`：**Isomap** 通过将每个实例与其最近的邻居连接起来来创建一个图，然后在试图保持实例之间的距离的同时降低维度。图中两个节点之间的距离是这些节点之间最短路径上的节点数。
- `sklearn.manifold.TSNE`：**t**分布随机邻域嵌入（**t-distributed stochastic neighbor embedding, t-SNE**）降低了维数，同时试图保持相似实例接近和不同实例分开。它主要用于可视化，特别是在高维空间中对实例簇进行可视化。例如，在本章结尾的练习中，您将使用t-SNE来可视化MNIST图像的2D图像。
- `sklearn.discriminant_analysis.LinearDiscriminantAnalysis`：线性判别分析（**Linear discriminant analysis, LDA**）是一种线性分类算法，它在训练期间学习类之间最具判别力的轴。然后可以使用这些轴定义一个超平面，将数据投影到该超平面上。这种方法的好处是投影将使类尽可能远离，因此 LDA 是一种很好的技术，可以在运行另一种分类算法之前降低维度（除非单独使用 LDA 就足够了）。

下图显示了瑞士卷上的 MDS、Isomap 和 t-SNE 的结果。MDS成功地使瑞士卷变平而不失去其全球曲率，而 Isomap 则完全下降。根据下游任务的不同，保留大规模的结构可能是好的或坏的。t-SNE 在压平瑞士卷方面做了一个合理的工作，保持了一点曲率，而且它还放大了集群，将卷分开。同样，这可能是好的或坏的，这取决于下游任务。

```
In [102... from sklearn.manifold import MDS

mds = MDS(n_components=2, random_state=42)
X_reduced_mds = mds.fit_transform(X_swiss)
```

```
In [103... from sklearn.manifold import Isomap

isomap = Isomap(n_components=2)
X_reduced_isomap = isomap.fit_transform(X_swiss)
```

```
In [104... from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, init="random", learning_rate="auto",
            random_state=42)
X_reduced_tsne = tsne.fit_transform(X_swiss)
```

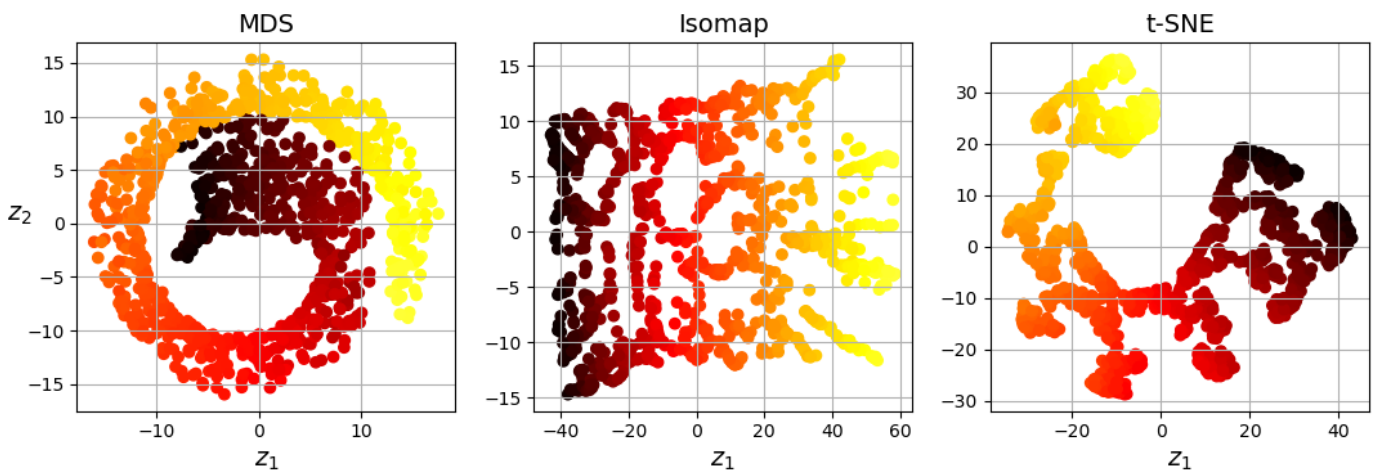
```
In [105... # extra code - this cell generates and saves Figure 8-11

titles = ["MDS", "Isomap", "t-SNE"]

plt.figure(figsize=(11, 4))

for subplot, title, X_reduced in zip((131, 132, 133), titles,
                                     (X_reduced_mds, X_reduced_isomap, X_reduced_tsne)):
    plt.subplot(subplot)
    plt.title(title)
    plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=darker_hot)
    plt.xlabel("$z_1$")
    if subplot == 131:
        plt.ylabel("$z_2$", rotation=0)
    plt.grid(True)

save_fig("other_dim_reduction_plot")
plt.show()
```



## 7. 额外的材料—Kernel PCA

In [106... `from sklearn.decomposition import KernelPCA`

```
rbf_pca = KernelPCA(n_components=2, kernel="rbf", gamma=0.04, random_state=42)
X_reduced = rbf_pca.fit_transform(X_swiss)
```

```
In [107... lin_pca = KernelPCA(kernel="linear")
rbf_pca = KernelPCA(kernel="rbf", gamma=0.002)
sig_pca = KernelPCA(kernel="sigmoid", gamma=0.002, coef0=1)

kernel_pcas = ((lin_pca, "Linear kernel"),
               (rbf_pca, rf"RBF kernel, $\gamma$={rbf_pca.gamma}$"),
               (sig_pca, rf"Sigmoid kernel, $\gamma$={sig_pca.gamma}, r={sig_pca.coef0}$"))

plt.figure(figsize=(11, 3.5))
for idx, (kpca, title) in enumerate(kernel_pcas):
    kpca.n_components = 2
    kpca.random_state = 42
    X_reduced = kpca.fit_transform(X_swiss)

    plt.subplot(1, 3, idx + 1)
    plt.title(title)
    plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=darker_hot)
    plt.xlabel("$Z_1$")
    if idx == 0:
        plt.ylabel("$Z_2$", rotation=0)
    plt.grid()

plt.show()
```

