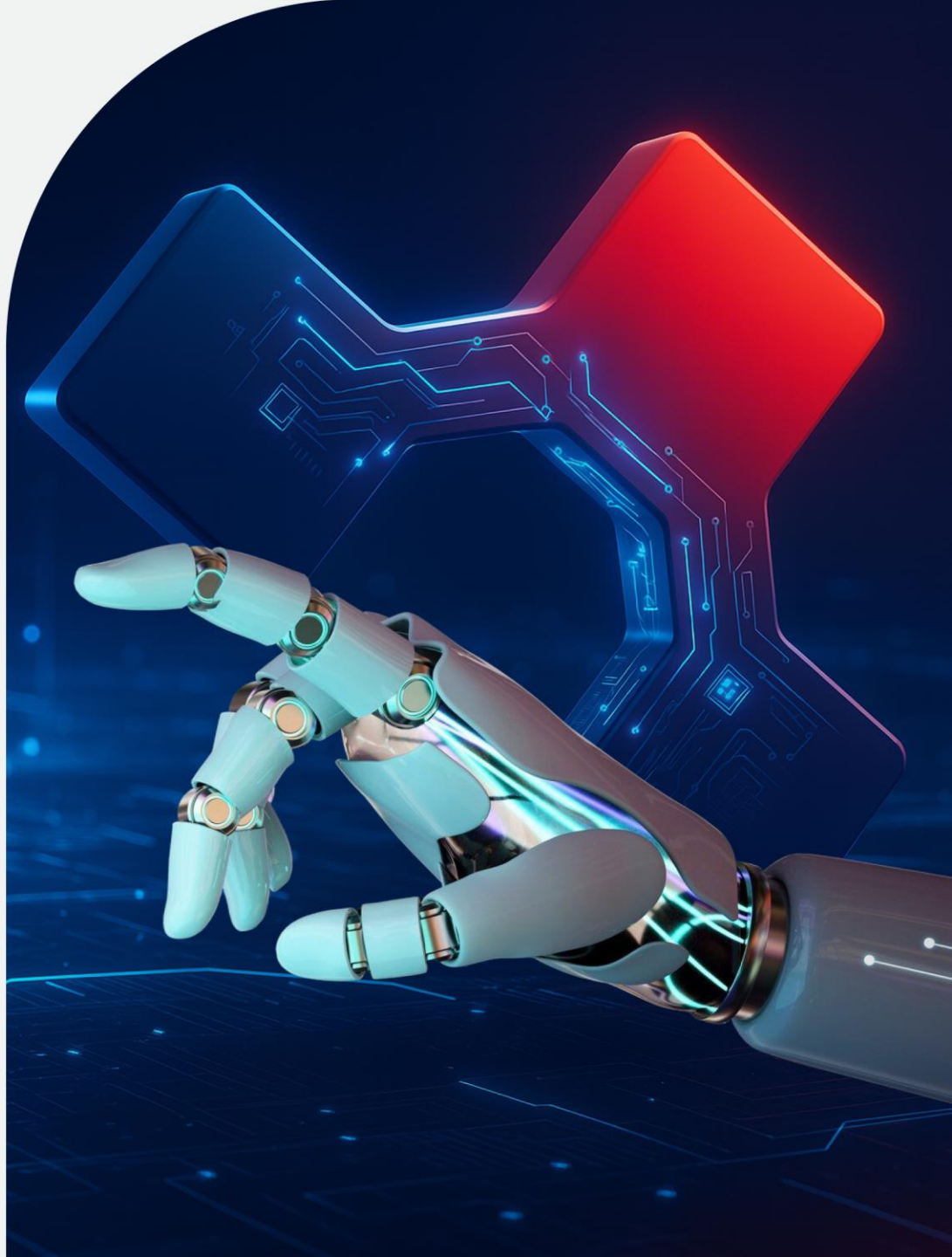
NCIA

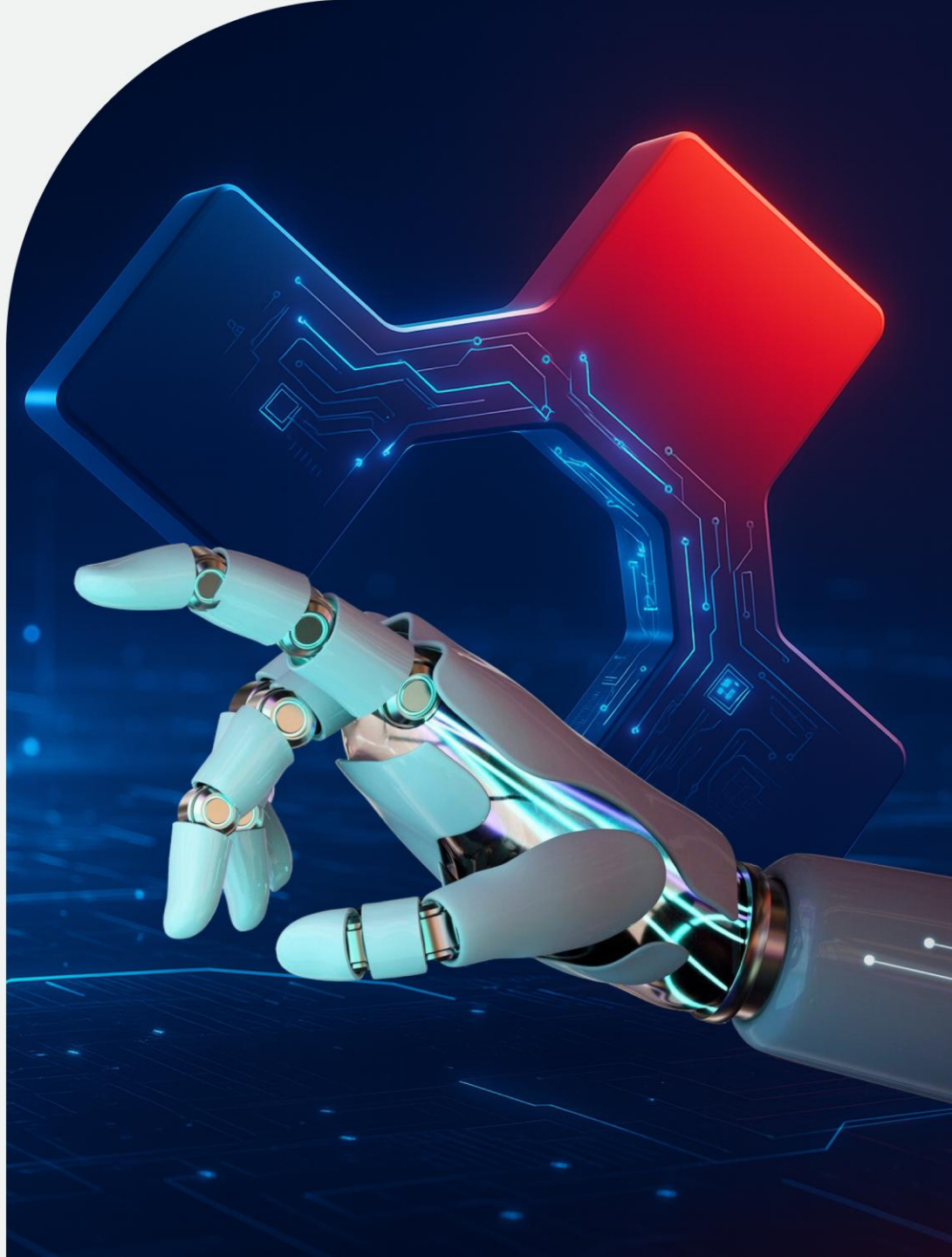
Núcleo de Capacitação em Inteligência Artificial



NCIA | FOXCONN FPF tech

Dimensionality Redution

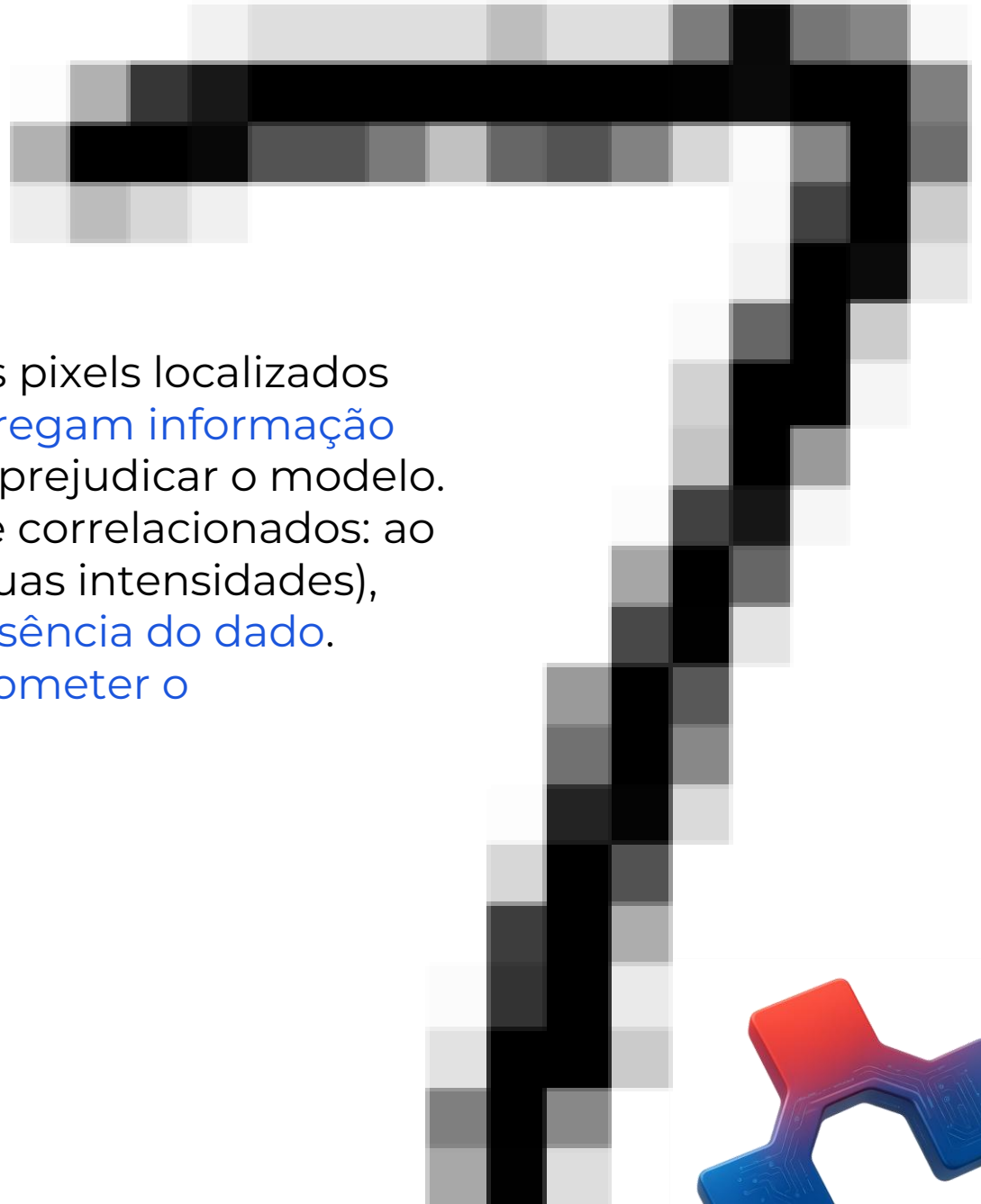
The Curse of Dimensionality; Main Approaches for Dimensionality Reduction; Projection; Manifold Learning; PCA; Preserving the Variance; Principal Components; Projecting Down to d Dimensions; Using Scikit-Learn; Explained Variance Ratio; Choosing the Right Number of Dimensions; PCA for Compression; Randomized PCA;



NCIA | FOXCONN FPF tech

1. Introdução à Redução de Dimensionalidade

Muitos problemas de aprendizado de máquina envolvem milhares ou até milhões de atributos (features). Esse excesso de dimensões torna o treinamento extremamente lento e aumenta a dificuldade de encontrar boas soluções. Esse fenômeno é conhecido como *maldição da dimensionalidade*. Nosso objetivo aqui é entender esse problema e aprender técnicas que permitem reduzir o número de dimensões sem perder informação relevante.

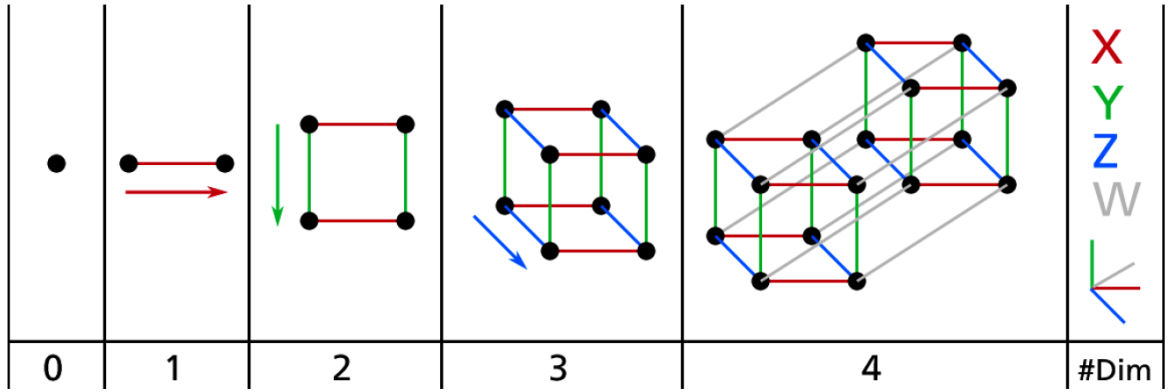


No conjunto de imagens MNIST, por exemplo, os pixels localizados nas bordas quase sempre são brancos e não carregam informação útil. Isso significa que podemos eliminá-los sem prejudicar o modelo. Além disso, muitos pixels vizinhos são altamente correlacionados: ao fundi-los (por exemplo, calculando a média de suas intensidades), conseguimos reduzir dimensões mantendo a essência do dado. Assim, simplificamos o treinamento sem comprometer o desempenho.

Existem duas abordagens principais para a redução de dimensionalidade. A primeira é a projeção, que consiste em representar os dados em um espaço de menor dimensão. A segunda é o aprendizado de variedades (*manifold learning*), que explora a estrutura geométrica dos dados em alta dimensão. Neste capítulo, vamos estudar três métodos populares: Análise de Componentes Principais (PCA), Projeção Aleatória e *Locally Linear Embedding* (LLE).

2. The Curse of Dimensionality

Nossa intuição funciona bem em três dimensões, mas falha quando pensamos em espaços de centenas ou milhares de dimensões. Na figura abaixo temos uma progressão: ponto (0D), segmento (1D), quadrado (2D), cubo (3D) e tesseract (4D). Esse último, impossível de visualizar corretamente, ilustra como é difícil imaginar objetos em dimensões superiores. Esse exemplo nos prepara para entender como os dados se comportam de forma muito diferente em espaços de alta dimensionalidade.

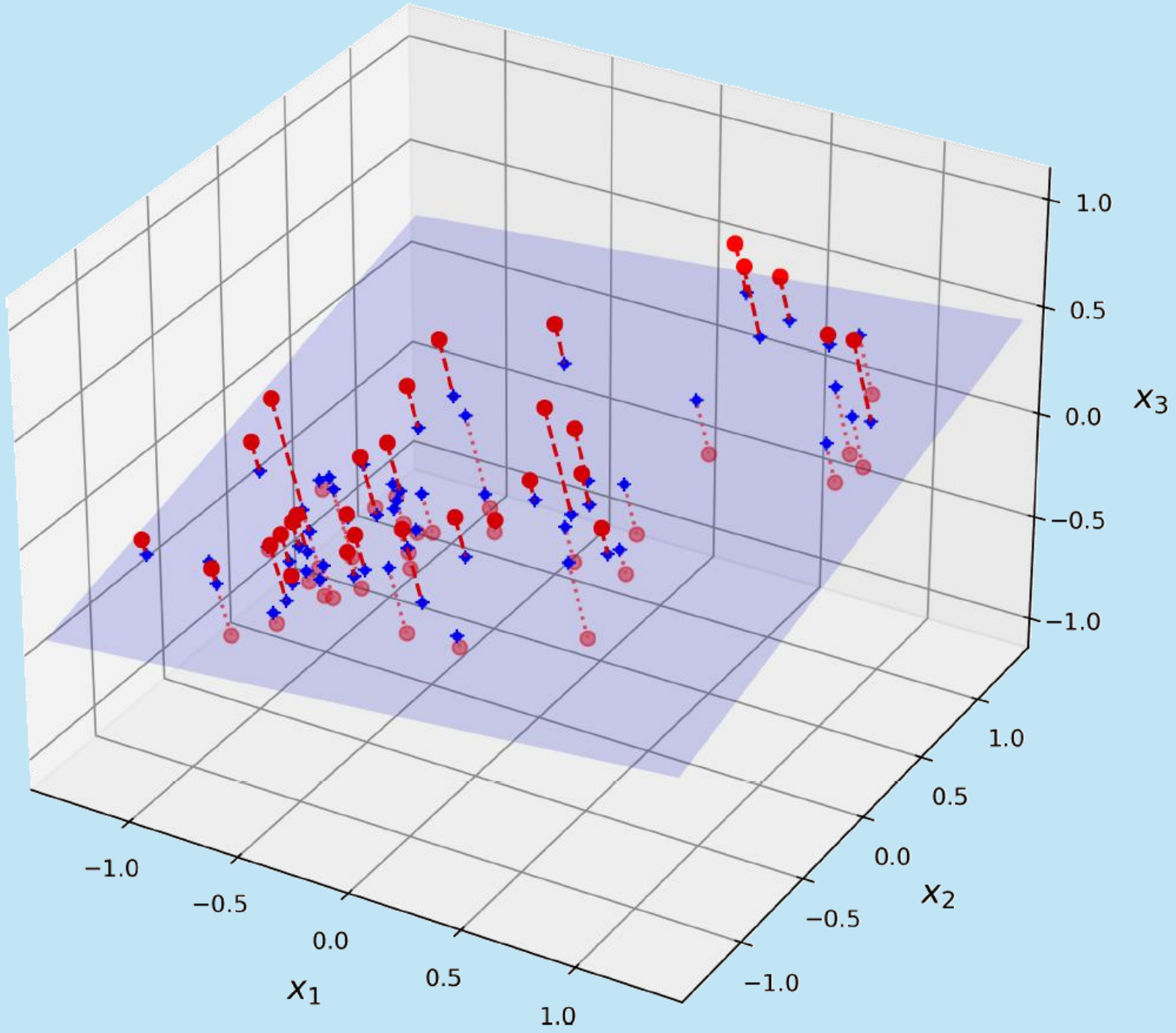


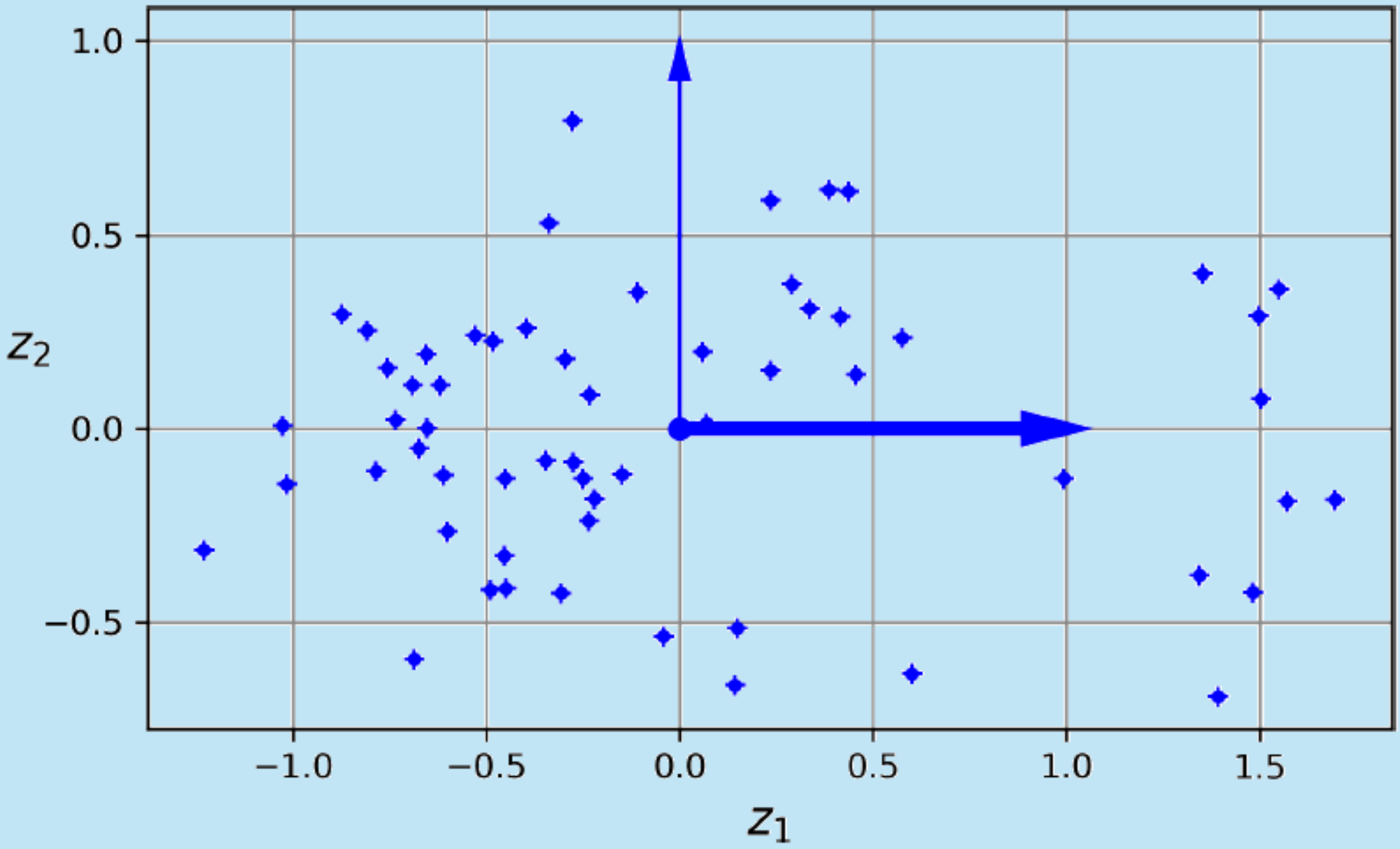
3. Main Approaches for Dimensionality Reduction

Antes de entrar nos algoritmos específicos, é importante entender que existem duas abordagens principais para reduzir a dimensionalidade: projeção e manifold learning. A projeção consiste em projetar os dados em um subespaço de menor dimensão. Já o *manifold learning* parte da ideia de que os dados reais geralmente estão próximos de uma variedade (manifold) de baixa dimensão que pode estar dobrada ou torcida dentro de um espaço de maior dimensão.

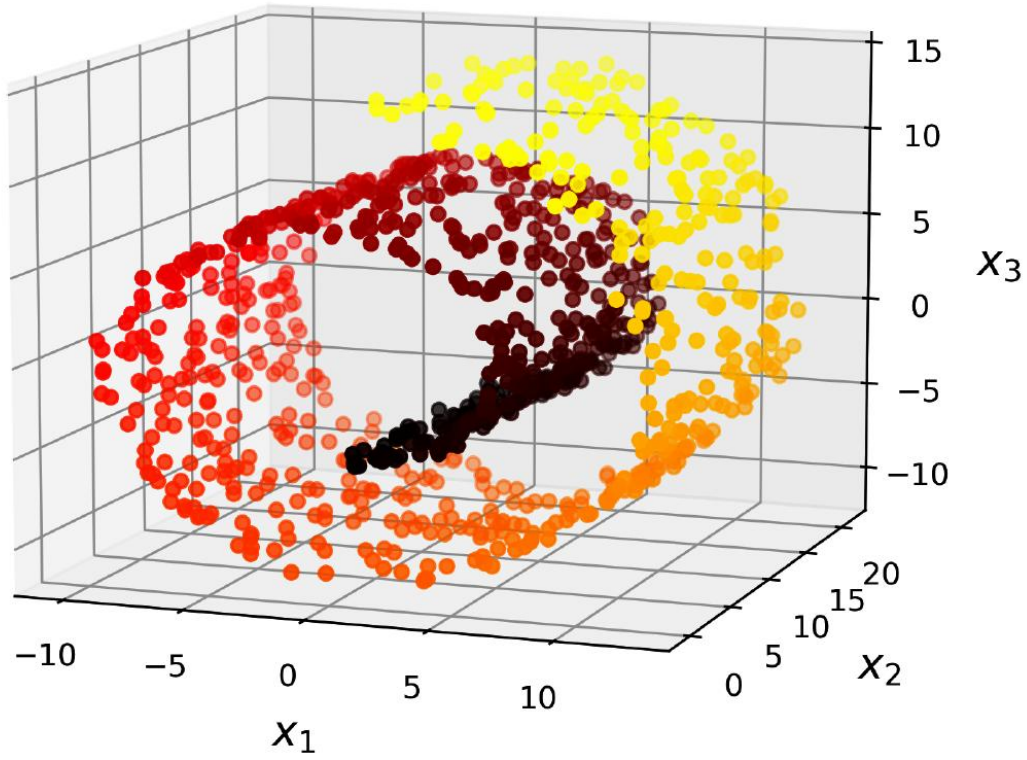
3.1 Projection

Em muitos problemas reais, os exemplos de treino não estão espalhados uniformemente em todas as direções. Eles se concentram próximos de um subespaço de menor dimensão. A próxima figura mostra um conjunto de dados em 3D cujos pontos estão próximos de um plano. Projetando-os perpendicularmente sobre esse plano, produzimos uma imagem correspondente de menor dimensionalidade (2D) sem perder muita informação.


1.0
0.5
0.0
−0.5
−1.0
x_3
1.0
0.5
0.0
−0.5
−1.0
x_2
−1.0
−0.5
0.0
0.5
1.0
x_1

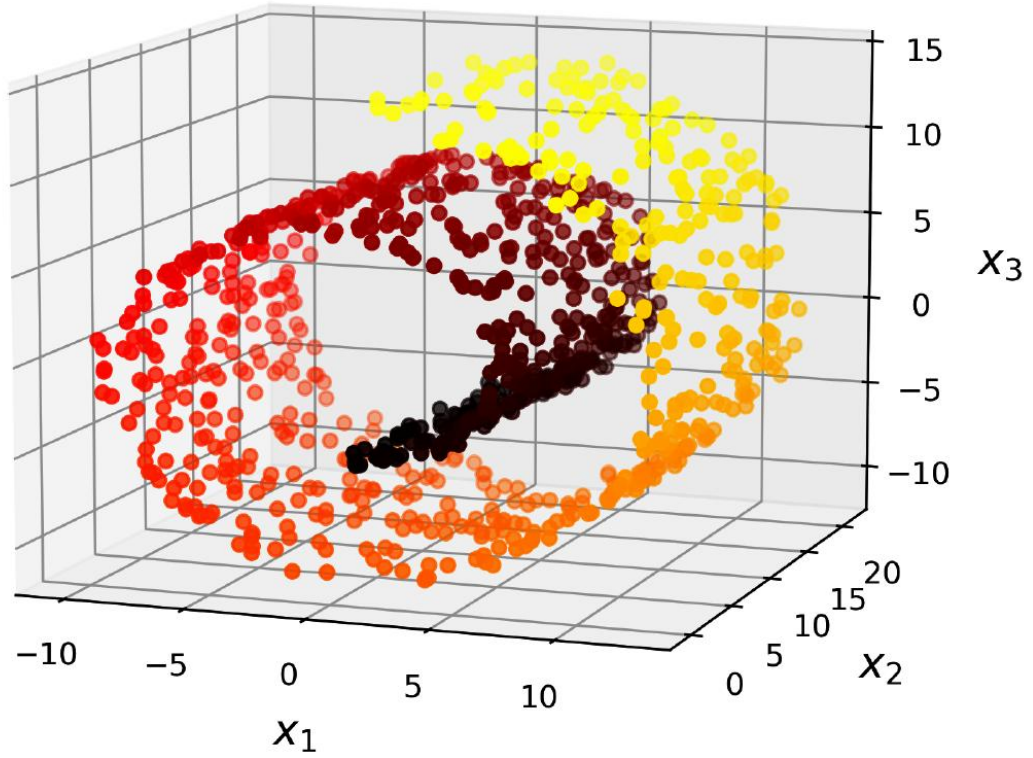

1.0
0.5
0.0
−0.5
z_2
−1.0
−0.5
0.0
0.5
1.0
1.5
z_1

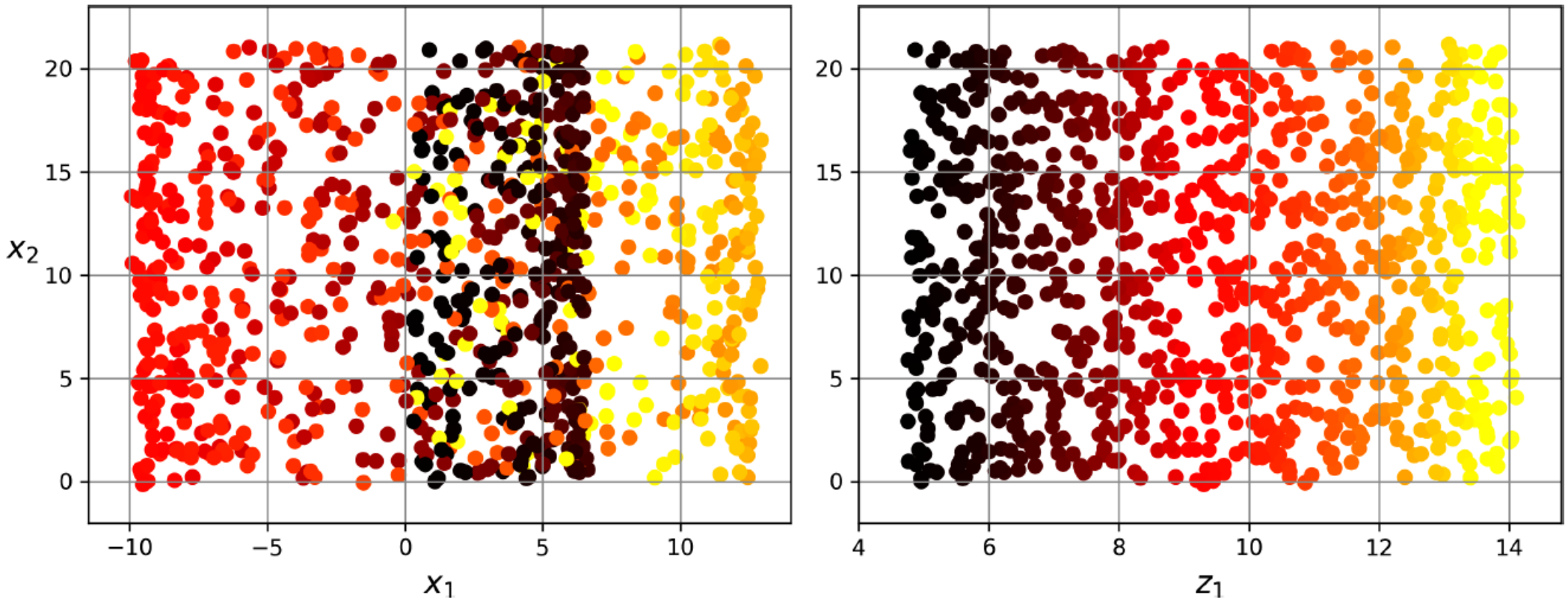
Nem sempre projetar em um plano resolve. O exemplo clássico é o *Swiss roll*: um conjunto de dados 2D enrolado dentro de um espaço 3D. Se apenas projetarmos para 2D descartando uma dimensão, diferentes camadas se sobrepõem. O ideal é “desenrolar” o Swiss roll, obtendo uma representação 2D plana.



3.2 Manifold Learning

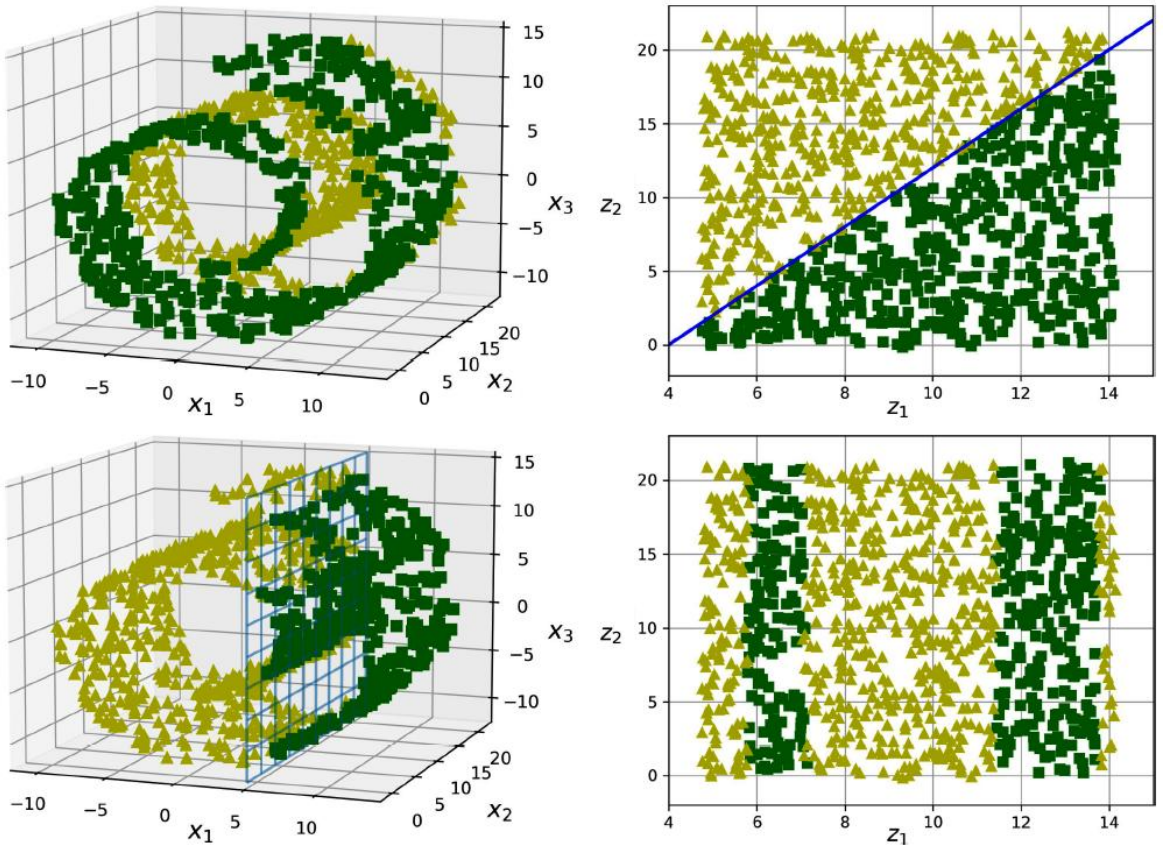
O Swiss roll é um exemplo de variedade (manifold): uma forma 2D torcida em um espaço 3D. De forma geral, um *d*-dimensional manifold é uma parte de um espaço *n*-dimensional (com $d < n$) que localmente se parece com um hiperplano de dimensão *d*. Muitos algoritmos de redução de dimensionalidade buscam justamente aprender esse manifold subjacente — por isso chamamos essa abordagem de manifold learning.




20
15
10
5
0
x_2
−10
−5
0
5
10
x_1
20
15
10
5
0
4
6
8
10
12
14
z_1

A hipótese do manifold afirma que a maioria dos conjuntos de dados reais de alta dimensão está, na verdade, próxima de um manifold de baixa dimensão. Pense no MNIST: as imagens de dígitos têm restrições claras (linhas conectadas, fundo branco, centralização), o que limita os graus de liberdade. Assim, embora as imagens tenham milhares de pixels, elas se distribuem em uma estrutura de dimensão muito menor.

Um benefício da redução de dimensionalidade é que ela pode simplificar tarefas de aprendizado. Na figura ao lado (linha superior), o Swiss roll está dividido em duas classes: em 3D, a fronteira de decisão é complexa, mas no espaço 2D desenrolado ela se torna uma linha reta. Porém, isso não é sempre verdade: na linha inferior da mesma figura, a fronteira é simples em 3D (um plano vertical), mas fica mais complicada no 2D desenrolado. Ou seja, reduzir dimensões acelera o treino, mas nem sempre simplifica o problema.

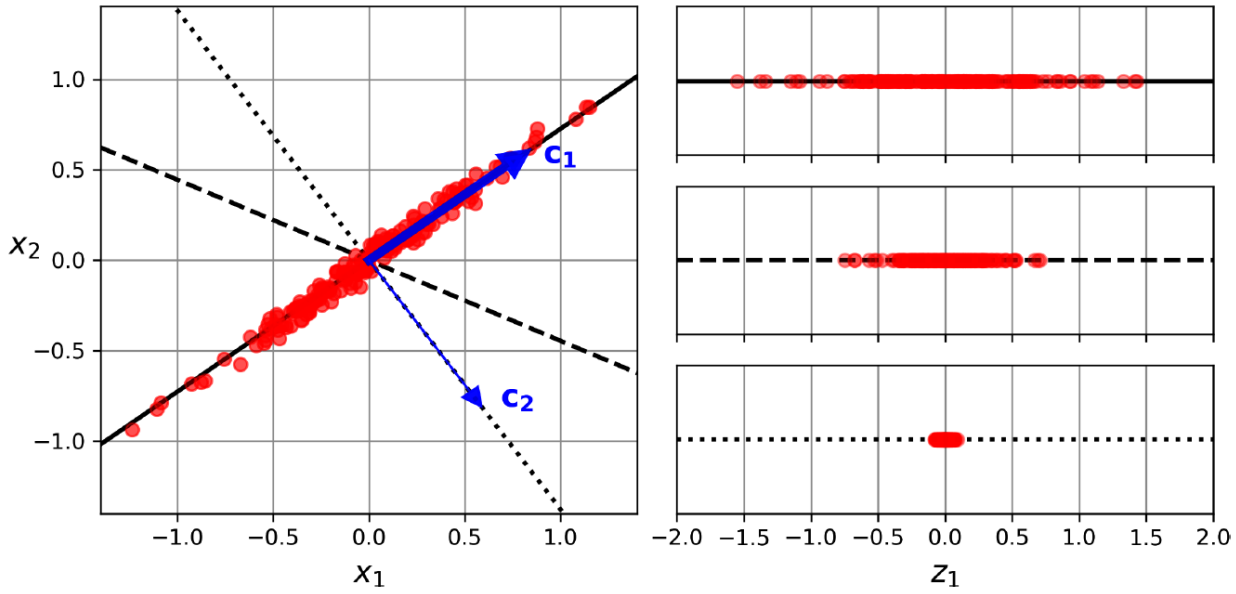


4. PCA

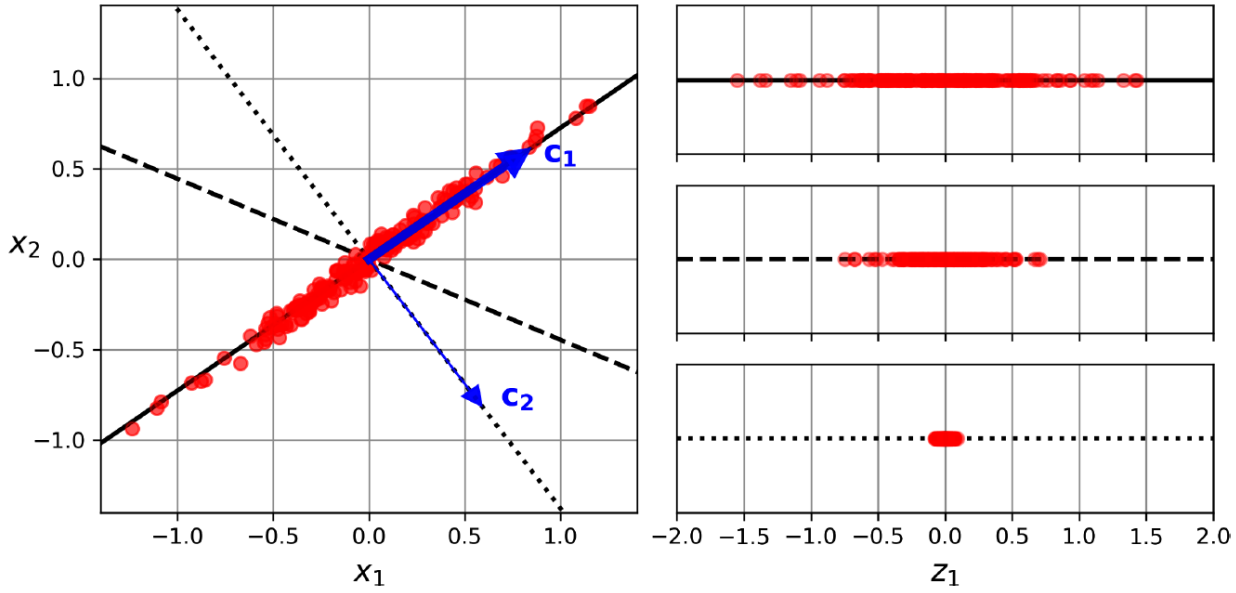
A Análise de Componentes Principais (PCA) é o algoritmo mais popular de redução de dimensionalidade. Ele busca identificar o hiperplano que fica mais próximo dos dados e projeta o conjunto de treino nesse subespaço. A ideia é simples: encontrar os eixos que preservam a maior parte da variância dos dados, reduzindo a perda de informação.

4.1 Preserving the Variance

Antes de projetar os dados em um hiperplano de menor dimensão, precisamos escolher o melhor eixo. A Figura 8-7 mostra um exemplo em 2D com três possíveis eixos. Ao projetar os pontos em cada eixo, percebemos que a projeção sobre a linha sólida preserva mais variância, enquanto a projeção na linha pontilhada perde quase toda a informação. O PCA sempre escolhe o eixo que preserva a máxima variância, o que equivale a m s e suas projeções.



O PCA encontra primeiro o eixo que explica a maior parte da variância, chamado primeiro componente principal (PC1). Em seguida, busca um segundo eixo, ortogonal ao primeiro, que explica a maior parte da variância restante (PC2), e assim por diante. Em um espaço com *n* dimensões, podemos obter até *n* componentes principais. Na abaixo, PC1 está alinhado com o vetor c1 e PC2 com o vetor c2.



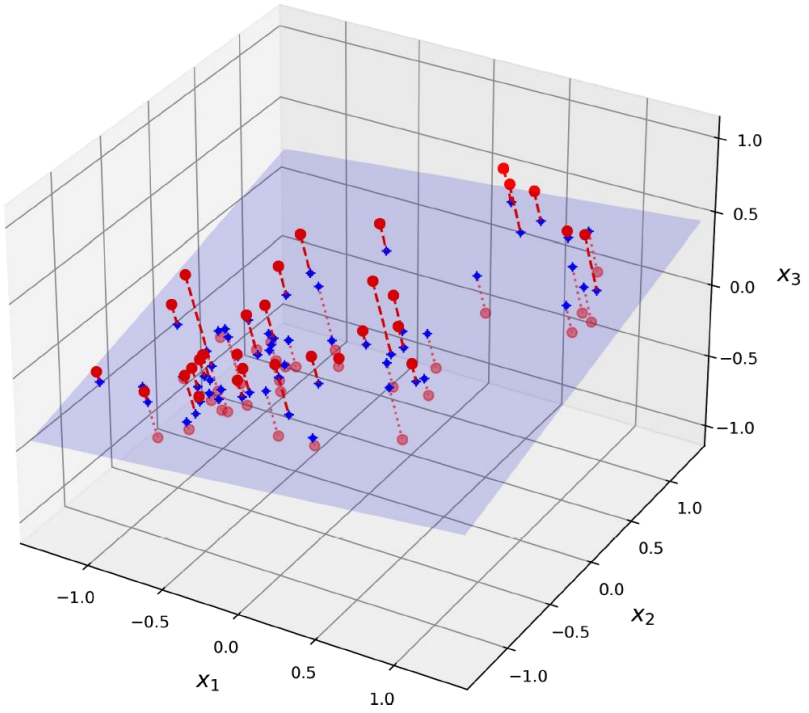
4.2 Principal Components

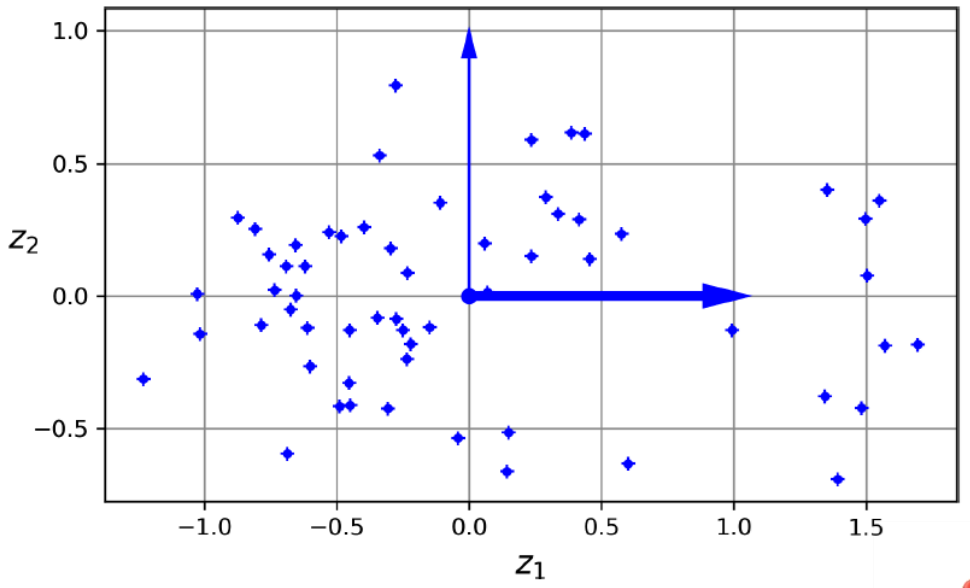
Para calcular os componentes principais usamos a Decomposição em Valores Singulares (SVD). Ela decompõe a matriz de dados X centralizada em três matrizes: $U\Sigma V^T$ .As colunas de Vsão exatamente os vetores unitários que definem os componentes principais. O código em Python abaixo mostra como obter os PCs de um dataset 3D e extrair os dois primeiros:

```python
import numpy as np

X = [...] # pequeno dataset 3D
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt[0]
c2 = Vt[1]
```

Depois de identificar os componentes principais, podemos projetar o dataset em um hiperplano definido pelos primeiros *d* PCs. Isso garante que a projeção preserve a maior variância possível. Por exemplo, no dataset 3D da esquerda, ao projetar sobre os dois primeiros PCs obtemos uma versão 2D que ainda mantém a maior parte da informação.





4.3 Projecting Down to d Dimensions

Seja W_d a matriz formada pelas prir $\mathbf{X}_{d\text{-proj}} = \mathbf{X}\mathbf{W}_d$; V. O dataset projetado é dado por:

No código abaixo projetamos c

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

pelos dois primeiros PCs:

Assim, reduzimos a dimensionalidade mantendo o máximo de variância dos dados.

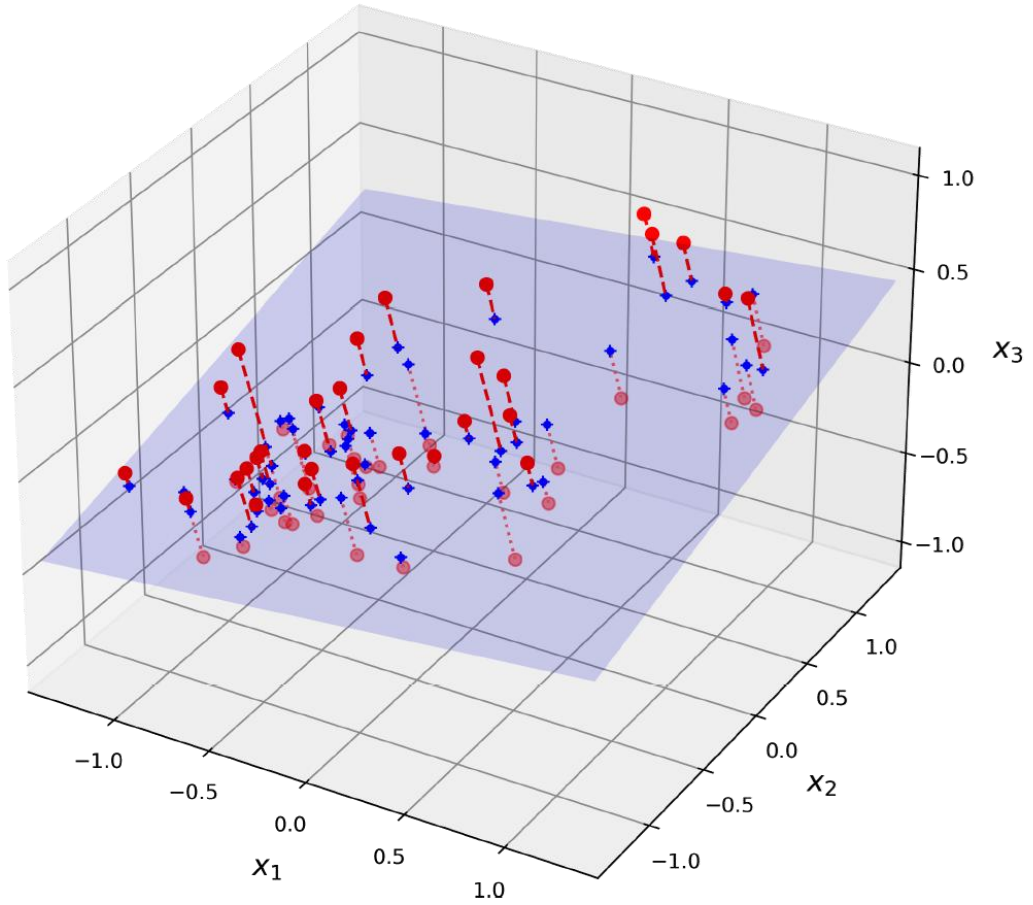
4.4 PCA no Scikit-Learn

A classe PCA do Scikit-Learn implementa PCA usando SVD internamente. Para reduzir os dados a duas dimensões, basta criar o transformador com n_components=2, ajustar com fit e aplicar transform. O método já centraliza os dados automaticamente.

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X2D = pca.fit_transform(X)
```

4.5 Explained Variance Ratio

O atributo explained_variance_ratio_ mostra a fração da variância explicada por cada componente principal. No exemplo do dataset 3D da figura ao lado, o primeiro componente explica cerca de 76% da variância, o segundo 15%, e o terceiro apenas 9%. Isso indica que os dois primeiros componentes já capturam a maior parte da informação.




```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X2D = pca.fit_transform(X)

pca.explained_variance_ratio_
```

Output: array([0.7578477 , 0.15186921])

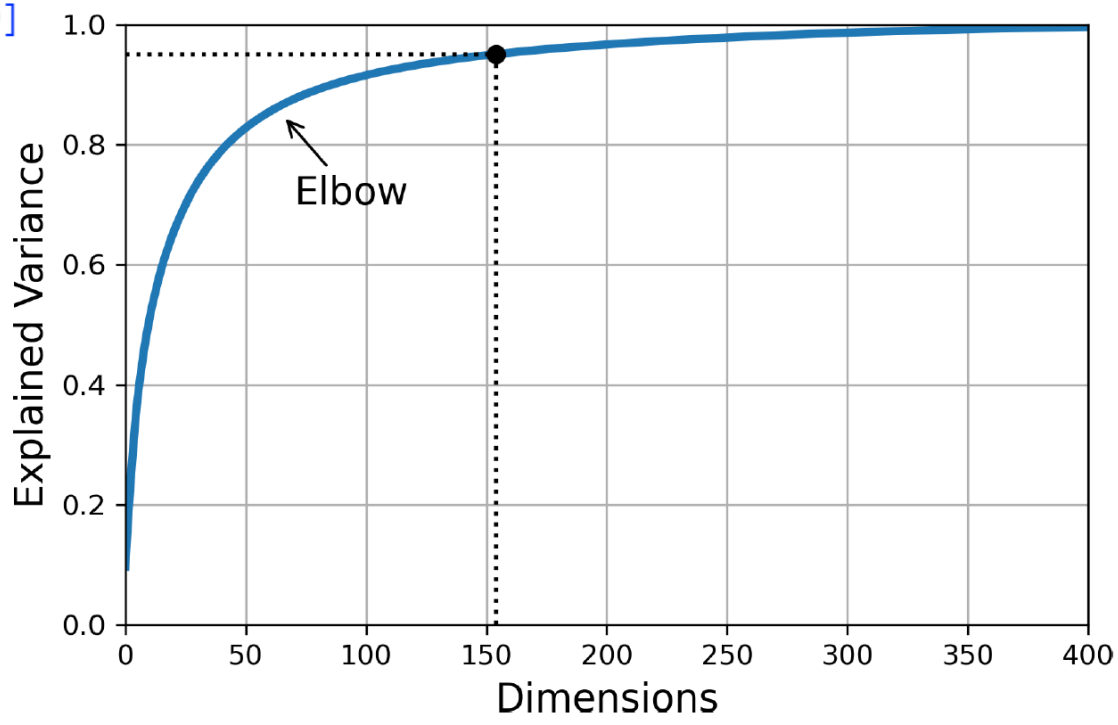
4.6 Choosing the Right Number of Dimensions

Em vez de escolher arbitrariamente o número de componentes, podemos selecionar a quantidade necessária para preservar uma fração da variância, por exemplo 95%. No MNIST, isso corresponde a 154 componentes, reduzindo de 784 atributos originais para apenas 154, mantendo praticamente toda a informação. Também é comum usar o gráfico de variância acumulada para identificar o “cotovelo” da curva, onde a variância deixa de crescer rapidamente.

```
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
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)

pca.n_components_
```

Output: 154



Quando a redução de dimensionalidade é parte de uma tarefa supervisionada, o número de componentes pode ser tratado como hiperparâmetro. O exemplo mostra um *pipeline* com PCA seguido de Random Forest. Usando RandomizedSearchCV, o melhor resultado para MNIST com 1.000 instâncias foi obtido com apenas 23 componentes, mostrando que modelos mais poderosos podem trabalhar bem com menos dimensões.

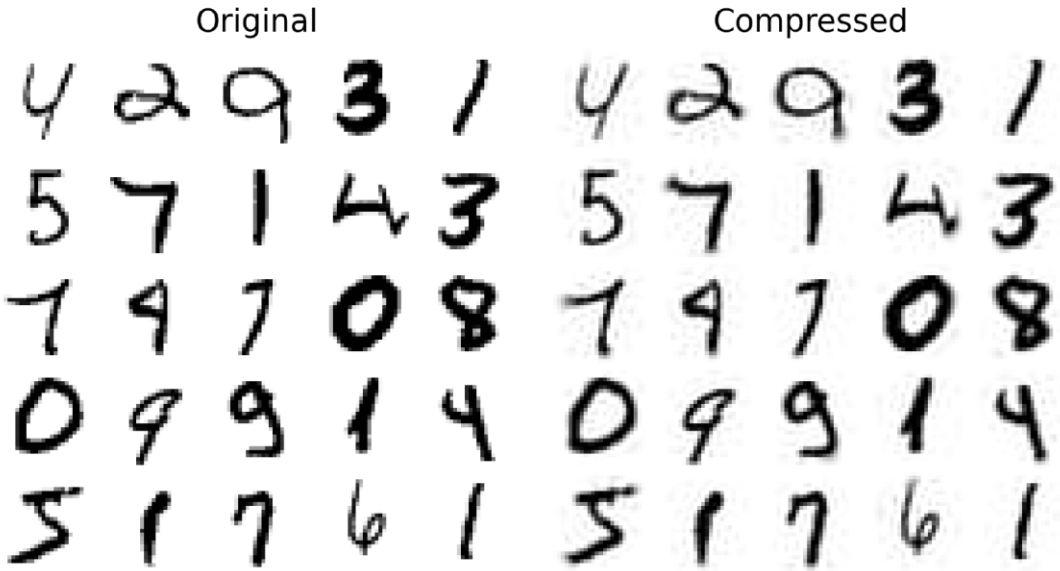
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import make_pipeline
clf = make_pipeline(PCA(random_state=42),
RandomForestClassifier(random_state=42))
param_distrib = {
"pca__n_components": np.arange(10, 80),
"randomforestclassifier__n_estimators": 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])

print(rnd_search.best_params_)
```

Output: {'randomforestclassifier__n_estimators': 465, 'pca__n_components': 23}

4.7 PCA for Compression

Uma aplicação prática do PCA é a compressão de dados. No MNIST, ao reduzir de 784 para 154 componentes (preservando 95% da variância), o dataset ocupa menos de 20% do espaço original. Embora parte da informação se perca, os dígitos comprimidos e depois reconstruídos ainda são reconhecíveis, com perda visual pequena. O erro médio dessa reconstrução é chamado de *reconstruction error*.



4.8 Randomized PCA

Quando precisamos apenas dos primeiros d componentes, o PCA com svd_solver="randomized" usa um algoritmo estocástico que aproxima bem os PCs com custo $O(m \cdot d^2)+O(d^3)$, em vez de $O(m \cdot n^2)+O(n^3)$ do SVD completo. Isso é dramaticamente mais rápido quando $d \ll n$ (m: instâncias, n: atributos).

```
from sklearn.decomposition import PCA
rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)
X_reduced = rnd_pca.fit_transform(X_train)
```

4.9. Incremental PCA (IPCA): processando em mini-batches)

O PCA “normal” precisa de todo o conjunto de treino em memória. O IncrementalPCA permite treinar em mini-batches via partial_fit, útil para bases grandes e para cenários online (dados chegando em fluxo).

```
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)
```

IPCA com memmap (dados maiores que a RAM)

Com NumPy memmap, manipulamos um array enorme salvo em disco como se estivesse na memória: o sistema carrega apenas os pedaços necessários. Primeiro, gravamos os dados em arquivo binário mapeado e garantimos flush(); na prática, você salvaria em blocos.

```
filename = "my_mnist.mmap"
X_mmap = np.memmap(filename, dtype='float32', mode='write', shape=X_train.shape)
X_mmap[:] = X_train  # ou gravar em blocos
X_mmap.flush()
```

Treinando IPCA direto do memmap

Depois, reabrimos o memmap em modo leitura e ajustamos o IncrementalPCA definindo batch_size. Como o algoritmo usa apenas partes do array por vez, o uso de memória fica sob controle, e podemos até chamar fit() em vez de partial_fit().

```
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)
```

4.10 Quando usar Random Projection (RP)

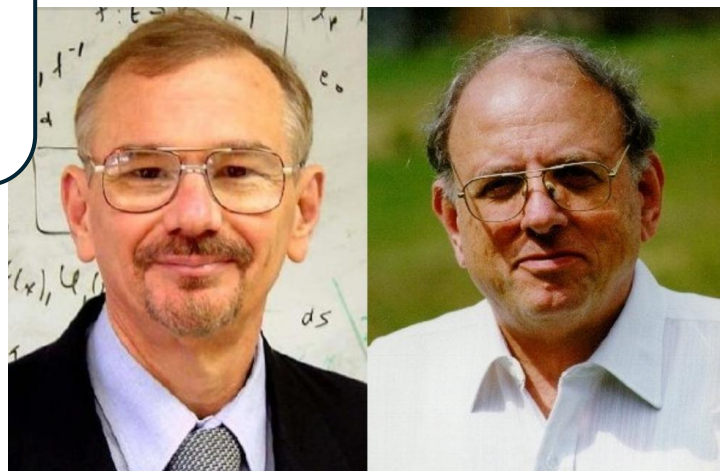
Mesmo com Randomized PCA, o custo $O(m \cdot d^2)+O(d^3)$ pode ser alto quando o alvo d ainda é grande (ex.: imagens com dezenas de milhares de features). Nesses casos, Random Projection é uma alternativa prática: projeta os dados para um espaço menor usando matrizes aleatórias com preservação aproximada de distâncias (ideia do lema de Johnson–Lindenstrauss), tornando o treino bem mais rápido.

- PCA (SVD completo): base “didática” e robusta; caro quando n é grande.
- Randomized PCA: mesma qualidade prática para os primeiros d PCs, porém muito mais rápido quando $d \ll n$.
- IncrementalPCA: quando os dados não cabem em memória; funciona com mini-batches e memmap.
- Random Projection: para altíssima dimensionalidade (ex.: visão computacional), quando até Randomized PCA fica lento.

5. Random Projection

O algoritmo de Random Projection reduz dimensionalidade aplicando uma projeção linear aleatória. Surpreendentemente, isso tende a preservar bem as distâncias entre instâncias, como demonstrado pelo lema de Johnson–Lindenstrauss. Assim, pontos próximos continuam próximos após a projeção, e pontos distantes continuam distantes. Quanto mais dimensões descartamos, maior a distorção, mas o lema fornece um limite seguro.

Dado um conjunto de m pontos em um espaço de alta dimensão ($\mathbb{R}^n$) possível projetá-los aleatoriamente em um espaço de dimensão muito menor ($\mathbb{R}^d$)de forma que todas as distâncias entre os pontos sejam preservadas aproximadamente, com alta probabilidade.



O lema de Johnson–Lindenstrauss fornece uma fórmula para o número mínimo de dimensões necessárias para garantir que as distâncias mudem menos do que uma tolerância ε. Por exemplo, com m=5.000 instâncias, n=20.000 features e ε=10%, basta reduzir para 7.300 dimensões.

```
from sklearn.random_projection import johnson_lindenstrauss_min_dim
m, ε = 5_000, 0.1
d = johnson_lindenstrauss_min_dim(m, eps=ε)
print(d)  # 7300
```

Podemos criar uma matriz aleatória P de dimensões $[d, n]$,com elementos da distribuição Normal(0, 1/d). Multiplicando os dados por P^T ,obtemos a versão reduzida.

```
n = 20_000
np.random.seed(42)
P = np.random.randn(d, n) / np.sqrt(d)
X = np.random.randn(m, n)  # dataset sintético
X_reduced = X @ P.T
```

O Scikit-Learn implementa o mesmo processo na classe GaussianRandomProjection. Ao chamar fit(), ele calcula automaticamente d pelo lema de Johnson–Lindenstrauss (a menos que seja definido manualmente).

```
from sklearn.random_projection import GaussianRandomProjection

gaussian_rnd_proj = GaussianRandomProjection(eps=ε, random_state=42)
X_reduced = gaussian_rnd_proj.fit_transform(X)
```

O SparseRandomProjection é uma versão mais eficiente:

•A matriz de projeção é esparsa, usando muito menos memória (25 MB contra 1,2 GB no exemplo).
•Mais rápido (~50%).
•Mantém esparsidade da entrada (útil para textos e dados esparsos).
•Preserva as mesmas propriedades de distâncias.
A densidade padrão é $1/n$. Com $n = 20.000$, apenas 1 em cada ~141 células é diferente de zero.

Inverse Transform (reconstrução aproximada)
Texto corrido:
Para inverter a projeção, usamos a pseudoinversa da matriz de componentes. Essa reconstrução é aproximada, já que parte da informação foi perdida.

```
components_pinv = np.linalg.pinv(gaussian_rnd_proj.components_)
X_recovered = X_reduced @ components_pinv.T
```

O Random Projection é um método simples, rápido, leve em memória e eficiente. Ele é especialmente útil para datasets com altíssima dimensionalidade (ex.: texto, imagens), quando até PCA se torna inviável. Embora seja baseado em aleatoriedade, garante com alta probabilidade que as distâncias entre instâncias sejam preservadas.

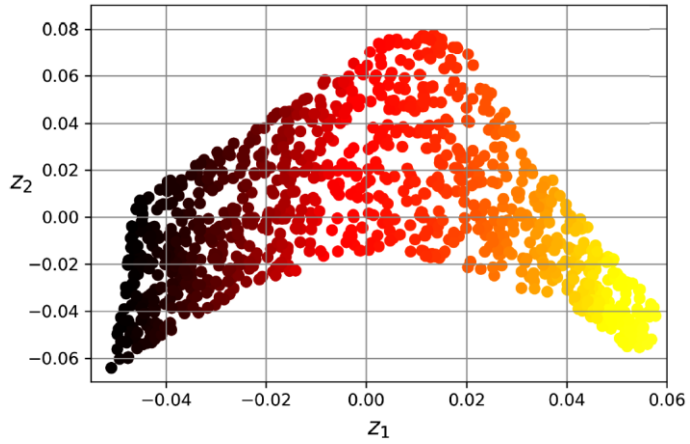
6. Locally Linear Embedding (LLE)

O *Locally Linear Embedding (LLE)* é uma técnica de redução de dimensionalidade não linear, baseada em *manifold learning*. Diferente do PCA e Random Projection, ele não projeta os dados em um subespaço, mas tenta desenrolar o manifold preservando as relações locais entre vizinhos. É especialmente eficaz quando os dados seguem uma estrutura curva (como o *Swiss roll*) e não há muito ruído.

No exemplo abaixo, o LLE aplica uma redução de 3D para 2D no dataset *Swiss roll*. Como resultado, o rolo é completamente desenrolado e as distâncias locais entre vizinhos são preservadas.

```
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)
```



Para cada instância $x^{(i)}$,o LLE encontra seus k vizinhos mais próximos. Em seguida, calcula pesos $w_{i,j}$ que melhor reconstroem $x^{(i)}$como combinação linear desses vizinhos. O problema é resolvido impondo que pesos somem 1 e sejam 0 para não vizinhos.

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

No segundo passo, mapeamos as instâncias em um espaço de menor dimensão Z. Agora os pesos ficam fixos, e o objetivo é encontrar representações $z^{(i)}$ que preservem as mesmas relações locais.

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

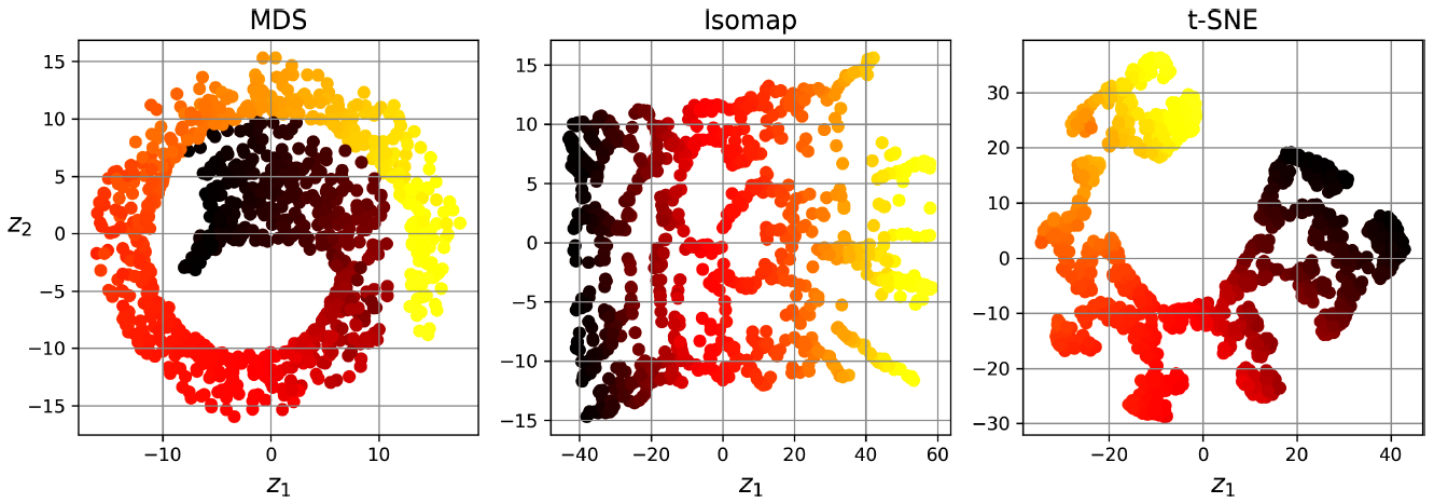
O LLE é poderoso, mas tem custo elevado:

- para encontrar vizinhos,
- $O(mnk^3)$ para calcular os pesos,
- $O(dm^2)$ para gerar representações em baixa dimensão.

O termo m^2 faz com que ele **não escale bem** para datasets muito grandes. Ainda assim, produz representações melhores em dados **não lineares**.

Além do PCA, Random Projection e LLE, o Scikit-Learn implementa várias outras técnicas:

- MDS (Multidimensional Scaling): preserva distâncias entre instâncias.
- Isomap: preserva distâncias geodésicas (caminho mais curto no grafo de vizinhanças).
- t-SNE: preserva vizinhos próximos e separa instâncias diferentes (muito usado para visualização e clusters).
- LDA (Linear Discriminant Analysis): supervisionado, encontra eixos que maximizam separação entre classes.



FPFtech