Chapter 7:
Pengurangan Dimensi (*Dimensionality Reduction*)

November 29, 2020

Banyak masalah *machine learning* melibatkan ribuan bahkan jutaan *feature-feature* untuk setiap training *instances*. Hal ini bukan hanya membuat training berjalan sangat lambat, tetapi juga membuat sulit untuk menemukan solusi yang baik. Permasalahan ini biasanya disebut dengan ***curse of dimensionality***.

Untungnya, pada permasalahan di dunia nyata, sering terdapat kemungkinan untuk mengurangi jumlah *feature* dengan cukup banyak, sehingga permasalahan yang sangat sulit atau bahkan tidak ada solusi (*intractable*) menjadi lebih sederhana dan dapat diperoleh solusi (*tractable*). Sebagai contoh, dataset MNIST yang sudah kita lihat di Chapter 3, kalau kita lihat piksel-piksel pada ujung gambar-gambar di dataset tersebut kebanyakan berwarna putih. Sehingga untuk mengurangi jumlah *feature* kita bisa menanggalkan piksel-piksel tersebut dari training set tanpa kehilangan informasi yang signifikan. Selain itu, pada kasus-kasus yang berhubungan dengan gambar atau image, piksel-piksel yang berdekatan biasanya mempunyai korelasi yang cukup tinggi. Dan jika kita gabungkan piksel-piksel yang berdekatan tersebut menjadi satu piksel yang representatif (misalkan dengan merata-ratakan intensitasnya), kita tidak akan kehilangan informasi secara signifikan.

Catatan. Mengurangi dimensi (*dimensionality reduction*) tentu akan menyebabkan kehilangan informasi (misalkan untuk kasus kompresi gambar ke JPEG dapat mendegradasi kualitas gambar). Oleh sebab itu, meskipun akan mempercepat proses training tetapi membuat kinerja sistem sedikit lebih buruk. Selain itu *pipelines* menjadi lebih kompleks dan sulit untuk dipertahankan. Maka, jika training terlalu lambat, kita harus mencoba terlebih dahulu untuk melakukan training berdasarkan data original sebelum mempertimbangkan pengurangan dimensi. Pada kasus tertentu, mengurangi dimensi dari data training bisa menghasilkan *filtering* noise dan juga menghilangkan detail data yang tidak diperlukan sehingga akan menghasilkan kinerja yang lebih tinggi. Tetapi secara umum hal itu tidak terjadi. Pengurangan dimensi kebanyakan akan mempercepat proses training saja.

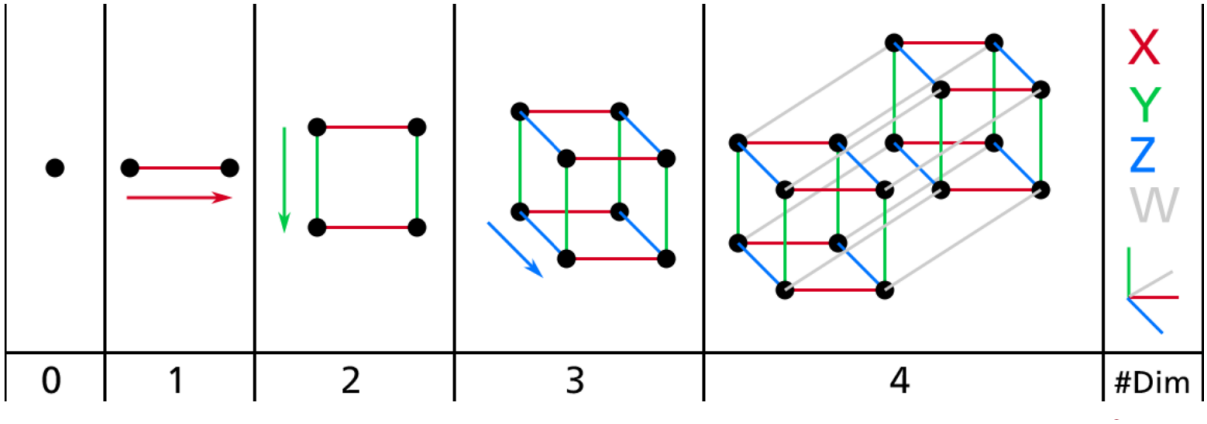
Selain untuk mempercepat proses training, pengurangan dimensi sangat dibutuhkan untuk visualisasi data (*DataViz*). Mengurangi jumlah dimensi sampai dengan dua atau tiga, akan memungkinkan untuk memvisualisasikan pandangan ringkas (*condensed view*) dari data traning yang dimensinya besar. Hal ini biasanya akan menghasilkan *insight* yang sangat penting dengan mendeteksi pola, seperti *cluster-cluster*. Terlebih lagi *DataViz* sangat esensial untuk mengkomunikasikan kesimpulan-kesimpulan tertentu kepada orang-orang yang bukan *data scientist*, dan khususnya pembuat keputusan yang akan menggunakan hasil yang kita peroleh.

Pada *Chapter* ini kita akan mendiskusikan *curse of dimensionality* dan mendapatkan gambaran apa yang terjadi pada ruang dengan dimensi besar. Kemudian, kita akan mendiskusikan dua pen-

dekatan utama pada masalah pengurangan dimensi, yaitu *projection* dan *manifold learning*. Selain itu kita akan mempelajari tiga teknik yang paling populer pada masalah pengurangan dimensi, yaitu: *Principle Component Analysis* (PCA), Kernel PCA dan *Locally Linear Embedding* (LLE).

1 *Curse of Dimensionality*

Kita terbiasa hidup di tiga dimensi dan intuisi kita selalu gagal ketika mencoba membayangkan ruang dimensi yang lebih besar. Bahkan mungkin hanya untuk *hypercube* 4D, kita kesulitan untuk membayangkan di dalam pikiran kita seperti Gambar 7.1.

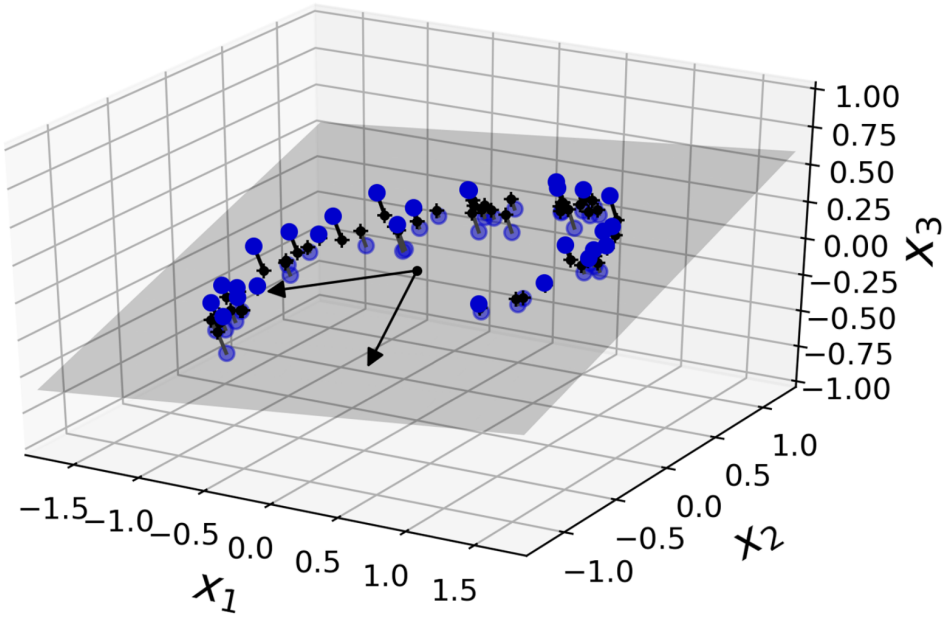


Gambar 7.1: Point, segment, square, cube, dan tesseract (0D ke 4D hypercubes)

Ternyata, banyak hal berprilaku sangat berbeda pada ruang dimensi tinggi. Misalkan, jika kita mengambil titik secara acak pada unit *square* (*square* 1×1), maka akan diperoleh hanya $0,4\%$ kemungkinan titik berlokasi lebih kecil dari $0,001$ dari batas (*border*). Dengan kata lain, adalah sangat kecil kemungkinan titik acak mempunyai nilai ekstrim pada dimensi sembarang. Tetapi pada dimensi 10000 unit *hypercube*, kemungkinan tersebut malah lebih besar dari $99,999999\%$. Kebanyakan titik pada *hypercube* dengan dimensi tinggi sangat dekat dengan perbatasan.

Beberapa perbedaan lain yang dapat menjadi masalah diantaranya adalah sebagai berikut. Jika kita mengambil dua titik secara random pada unit *square*, jarak antara dua titik ini secara rata-rata adalah kira-kira sama dengan 0.52. Tetapi jika kita lakukan dua titik random pada unit 3D *cube*, jarak rata-rata menjadi kira-kira 0.66. Tetapi, apa yang terjadi jika kita mengambil dua titik random tersebut pada *hypercube* berdimensi 10000? Jarak rata-rata, percaya atau tidak, akan mempunyai harga 408.25 (sekitar $\sqrt{1000000/6}$). Hal ini *counterintuitive*, bagaimana bisa dua titik menjadi sangat berjauhan padahal berada pada *hypercube* yang sama? Mungkin hal ini disebabkan banyaknya ruang pada dimensi-dimensi tinggi. Sehingga, dataset dengan dimensti tinggi mempunyai resiko bersifat *very sparse* (sangat jarang/renggang). Kebanyakan *instance-instance* training akan berjauhan satu sama lain. Ini juga berarti bahwa *instance* baru kemungkinan besar jauh dari training *instance* dan membuat prediksi sangat tidak *reliable* dibandingkan dengan dimensi lebih kecil. Ringkasnya, semakin banyak dimensi yang dimiliki oleh dataset training, semakin besar kemungkinan terjadi *overfitting*.

Secara teori, salah satu solusi dari *the curse of dimensionality* bisa jadi dengan menambah ukuran training set untuk meraih kepadatan training *instance* yang cukup. Tetapi pada kenyataan praktis, jumlah training *instances* yang dibutuhkan untuk mencapai kepadatan tertentu naik se-



Gambar 7.2: Dataset 3D yang berlokasi dekat dengan sebuah sub-ruang 2D

cara eksponensial dengan bertambahnya jumlah dimensi. Oleh sebab itu jumlah training *instances* yang dibutuhkan menjadi tidak rasional untuk dimensi sangat tinggi.

2 Pendekatan utama dalam *Dimensionality Reduction*

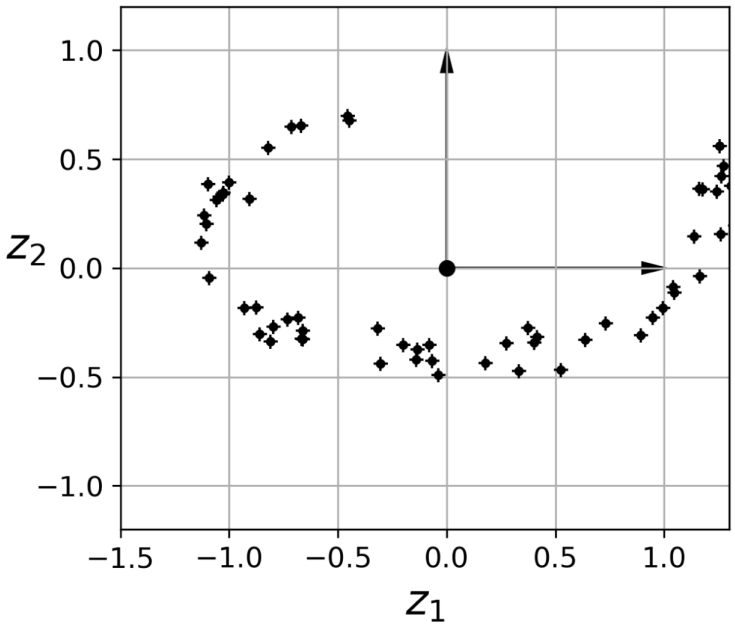
Sebelum kita masuk pada algoritma-algoritma spesifik untuk pengurangan dimensi (*dimensionality reduction*), terlebih dahulu akan kita tinjau dua pendekatan utama yaitu ***Projection*** dan ***Manifold Learning***.

2.1 Proyeksi (*Projection*)

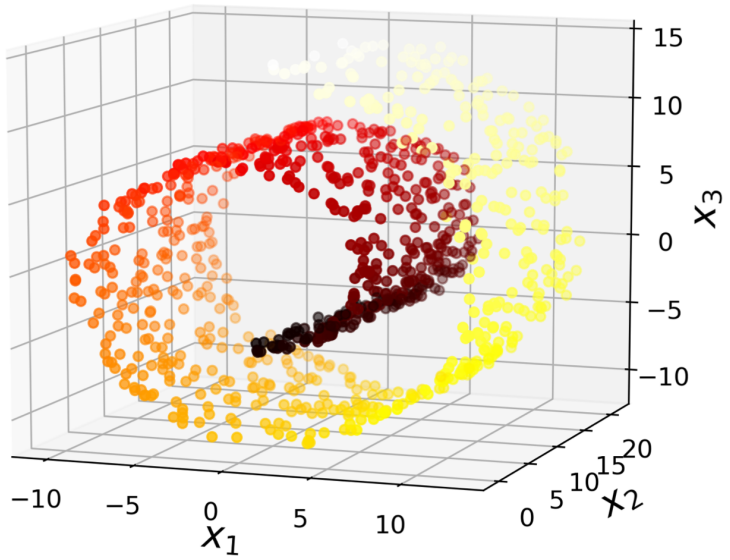
Pada permasalahan-permasalah real, biasanya *training instance* tidak tersebar secara uniform di seluruh dimensi. Banyak *feature-feature* yang hampir konstan, sementara yang lain saling berkorelasi sangat kuat. Hasilnya, semua training *instance* terletak di dalam (dekat dengan) sub-ruang (*subspace*) yang berdimensi jauh lebih rendah yang merupakan bagian dari ruang berdimensi tinggi tadi. Konsep ini terdengar cukup abstrak, oleh sebab ini kita akan melihat contoh berikut.

Pada Gambar 7.2 kita bisa lihat dataset 3D yang direpresentasikan dengan bulatan. Pada gambar tersebut jika kita perhatikan, semua training *instances* terletak dekat dengan sebuah bidang (*plane*) yang merupakan sub-ruang dari ruang 3D, dengan dimensi sub-ruang yang lebih rendah (2D). Jika kita melakukan proyeksi tegak lurus (*perpendicular*) terhadap pada sub-ruang ini, kita akan mendapatkan dataset 2D baru seperti yang ditunjukan pada Gambar 7.3 (Pada Gambar 7.2 Proyeksi tersebut direpresentasikan dengan garis hitam pendek, yang menghubungkan tiap *instance* dengan bidang 2D). Sehingga kita sudah melakukan pengurangan dimensi dari 3D menjadi 2D. Pada Gambar 7.3, sumbu-sumbu tersebut sudah menyatakan *feature* baru yaitu Z_1 dan Z_2 (koordinat hasil proyeksi pada bidang).

Tetapi, proyeksi tidak selalu menjadi pendekatan yang terbaik untuk pengurangan dimensi. Pada



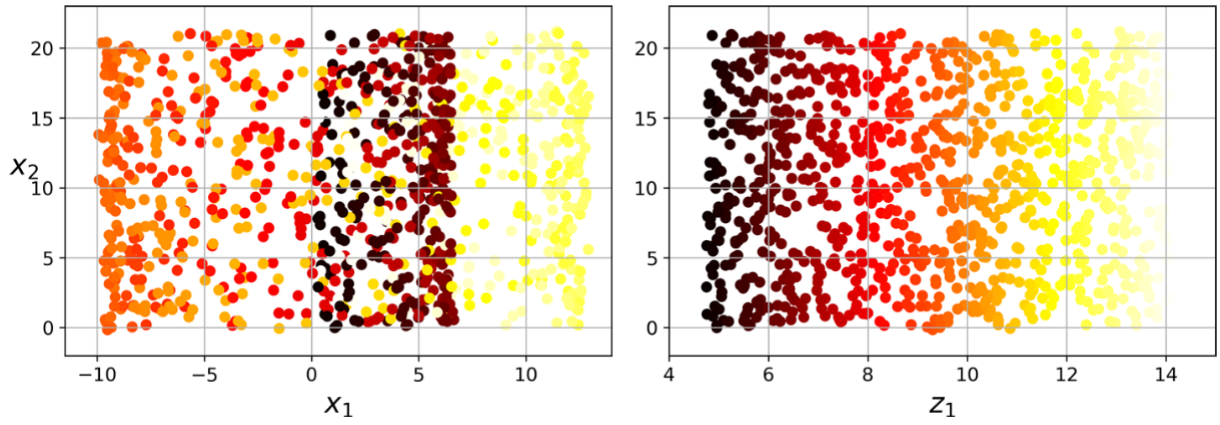
Gambar 7.3: Dataset 2D baru hasil proyeksi dari dataset 3D



Gambar 7.4: Dataset sintetik Swiss Roll

banyak kasus memungkin sub-ruang hasil proyeksi menjadi berliku, seperti pada dataset sintetik *Swiss roll* yang ditunjukan pada Gambar 7.4.

Dengan memproyeksikannya secara langsung training *instances* di dataset *Swiss Roll* pada sebuah bidang (misalkan dengan menghilangkan variable x_3) maka akan menumpukan beberapa layer dari *Swiss Roll* secara bersamaan, seperti yang ditunjukan pada Gambar 7.5 sebelah kiri (terlihat data sulit untuk dipisahkan). Yang kita inginkan sebetulnya untuk *unroll* dataset *Swiss roll* tadi sehingga didapatkan dataset 2D seperti pada Gambar 7.5 sebelah kanan (terlihat mudah untuk dipisahkan).



Gambar 7.5: Hasil proyeksi pada sebuah bidang (kiri) dan hasil unroll (kanan) dataset Swiss Roll

2.2 *Manifold Learning*

Dataset *Swiss Roll* merupakan sebuah contoh 2D *manifold*. *Manifold* 2D adalah sebuah bentuk 2D yang dapat dibengkokan dan dibelitkan pada dimensi yang lebih tinggi. Secara umum, *manifold* dimensi d merupakan bagian dari sebuah ruang dimensi n dimana $d < n$ yang secara lokal menyerupai *hyperplane* dengan dimensi d. Pada kasus *Swiss Roll*, $d = 2$ dan $n = 3$, secara lokal menyerupai bidang 2D, tetapi digulung pada dimensi ketiga.

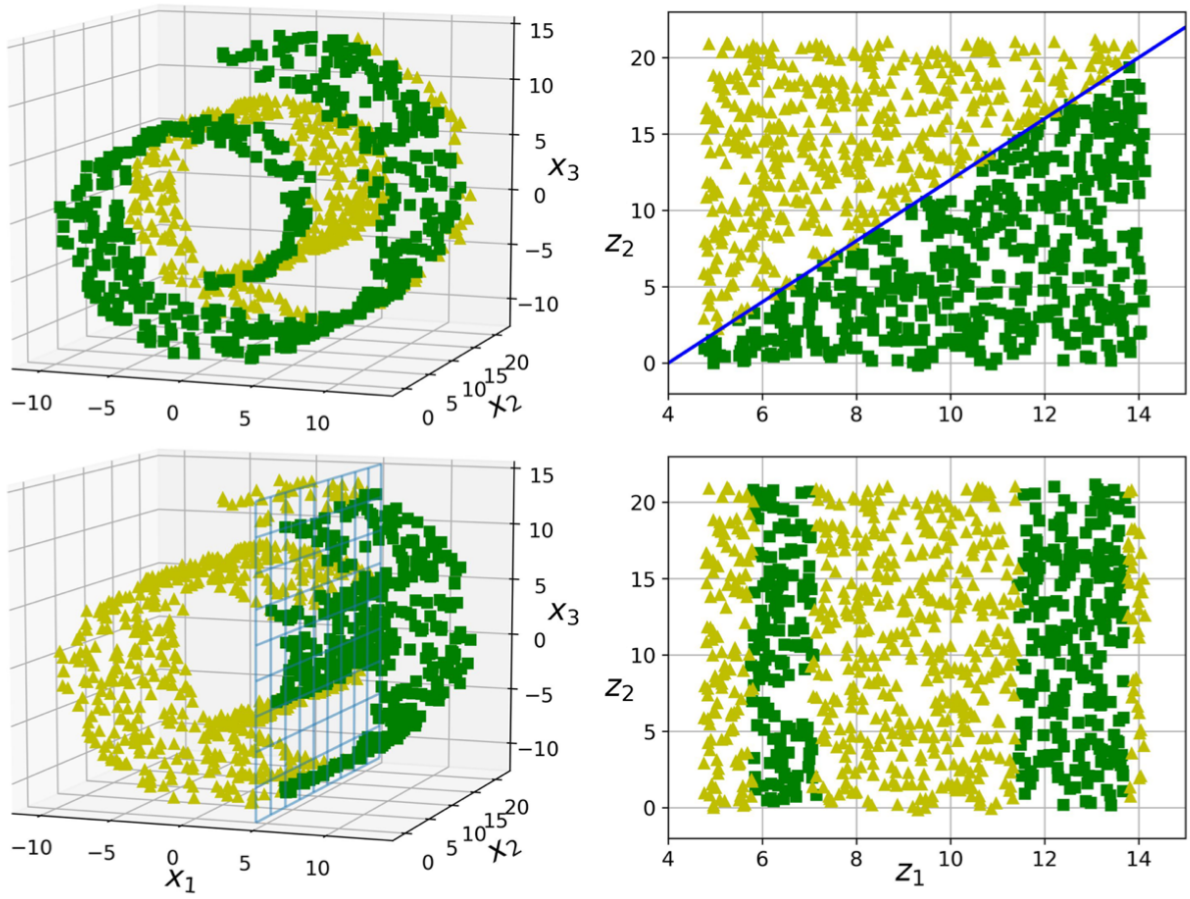
Banyak algoritma pengurangan dimensi bekerja dengan memodelkan *manifold* dimana terletak *instance-instance*, dan disebut dengan ***Manifold Learning***. Metode ini bergantung pada asumsi *manifold* (disebut juga *manifold hypothesis*) yang secara prinsip kebanyakan dataset real dengan dimensi tinggi terletak pada *manifold* dengan dimensi yang jauh lebih rendah. Asumsi ini sering tampak secara empiris.

Jika kita lihat lagi dataset MNIST berisi gambar-gambar tulisan tangan dijit angka yang mempunyai keserupaan (*similarity*). Mereka terbuat dari garis-garis yang terhubung, dengan batas warna putih, dan cenderung berada di tengah gambar. Jika kita membangkitkan *image* secara random, maka sangat kecil kemungkinan untuk mendapatkan gambar yang mirip dengan dijit-dijit tulisan tangan. Dengan kata lain, derajat kebebasan yang tersedia untuk kita membuat gambar dijit jauh lebih kecil dibandingkan dengan derajat kebebasan untuk membuat gambar apapun yang diinginkan. Keterbatasan ini cenderung akan memeras (*squeeze*) dataset menjadi manifold dengan dimensi yang lebih rendah.

Asumsi *manifold* sering diikuti dengan asumsi implisit lain, bahwa tugas yang akan dilakukan (mis. regresi atau klasifikasi) akan lebih sederhana jika diekspresikan pada ruang *manifold* dengan dimensi yang lebih kecil. Misalkan baris atas pada Gambar 7.6, *Swiss roll* dibagi ke dalam 2 kelas, yaitu pada 3D (gambar sebelah kiri), batas keputusan (*decision boundary*) akan cukup kompleks, tetapi pada ruang *unrolled manifold* 2D (gambar sebelah kanan) batas keputusan merupakan garis lurus.

Kendatipun demikian, asumsi implisit ini tidak selalu benar. Misalkan pada baris bawah di Gambar 7.6, batas keputusan berada pada $x_1 = 5$. Batas keputusan terlihat sederhana pada ruang 3D yang ada (berupa bidang vertikal), tetapi malah lebih kompleks pada *unrolled manifold* (teridiri dari 4 garis yang independen). Secara singkat, mengurangi dimensi training set sebelum men-

training model akan mempercepat training, tetapi tidak selalu membawa pada solusi yang lebih sederhana. Semuanya akan tergantung pada dataset yang dipunyai.



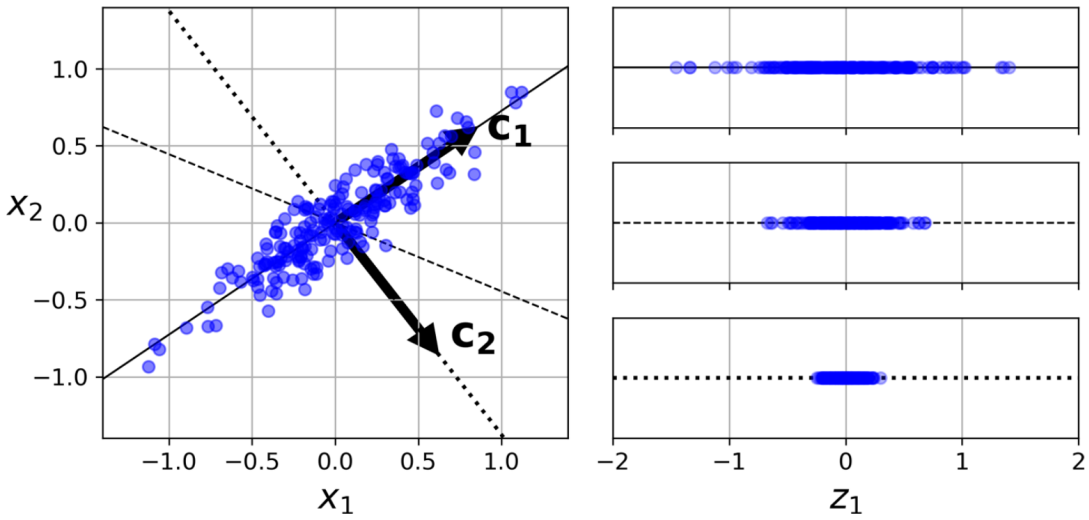
Gambar 7.6: Batas keputusan (decision boundary) tidak selalu lebih sederhana pada dimensi lebih rendah

3 *Principle Component Analysis* (PCA)

PCA sejauh ini merupakan algoritma pengurangan dimensi yang paling populer. Pertama-tama akan mengidentifikasi *hyperplane* yang terletak sangat dekat dengan data, dan kemudian melakukan proyeksi data *hyperplane* tersebut, seperti yang ditunjukan pada Gambar 7.2 sebelumnya.

3.1 Mempertahankan Variansi

Sebelum melakukan proyeksi training set pada *hyperplane* dengan dimensi yang lebih rendah, maka pertama kali yang harus dilakukan adalah memilih *hyperplane* yang tepat. Sebagai contoh, dataset 2D sederhana yang direpresentasikan pada Gambar 7.7, bersama 3 sumbu berbeda berupa *hyperplane* 1D. Gambar sebelah kanan menunjukan hasil proyeksi dataset terhadap 3 sumbu berbeda tadi. Dapat kita lihat bahwa proyeksi pada garis tebal C_1 akan mempertahankan variansi maksimum data, sementara proyeksi pada garis titik-titik mempertahankan variansi yang sangat kecil dari data, dan proyeksi pada garis putus-putus variansi yang dihasilkan berada diantaranya.



Gambar 7.7: Memilih subspace sebagai target koordinat proyeksi

Berdasarkan Gambar 7.7, sangat beralasan jika memilih sumbu proyeksi yang akan mempertahankan sejumlah variansi maksimum, karena dengannya tidak akan terlalu banyak informasi yang hilang dibanding proyeksi terhadap sumbu lain. Cara lain untuk menjustifikasinya adalah bahwa pilihan sumbu proyeksi terbaik tadi akan meminimalkan jarak kuadrat rata-rata (*mean squared distance*) antara dataset original dan dataset hasil proyeksi pada sumbu tersebut. Inilah ide sederhana yang mendasari prinsip PCA.

3.2 *Principle Component*

PCA mengidentifikasi sumbu yang menangkap (*capture*) variansi terbesar pada training set, seperti garis tebal C_1 pada Gambar 7.7. Algoritma ini juga dapat mengidentifikasi sumbu kedua yang ortogonal (tegak lurus) terhadap sumbu pertama, dan dapat menangkap variansi sisa pada training set seperti garis C_2 pada Gambar 7.7. Jika dataset berdimensi tinggi maka akan dilanjutkan dengan sumbu ketiga, dipilih yang ortogonal terhadap sumbu pertama dan kedua, dan seterusnya sebanyak jumlah dimensi pada dataset.

Sumbu ke-*i* disebut dengan principle component (PC) ke-*i* pada data. Pada Gambar 7.7, sumbu C_1 merupakan PC ke-1, sumbu C_2 merupakan PC ke-2. Pada Gambar 7.2, dua PC pertama adalah dua sumbu ortogonal dimana dua panah terletak (bidang), dan PC ke-3 adalah sumbu yang tegak lurus pada bidang tersebut.

Catatan. Untuk setiap *principle component*, PCA menemukan unit vektor yang berpusat pada nol yang menunjuk ke arah PC. Karena 2 unit vektor (arah C_1 dan arah berlawanan $-C_1$) terletak pada sumbu yang sama, sehingga arah unit vektor-unit vektor yang ditemukan oleh PCA tidak stabil. Jika kita merubah sedikit training set sedikit dan mengeksekusi algoritma PCA lagi, unit-unit vektor bisa jadi mengarah pada arah yang berlawanan dengan unit-unit vektor sebelumnya. Tetapi, secara umum keduanya akan tetap terletak pada sumbu yang sama. Pada beberapa kasus, pasangan unit vektor bisa jadi berotasi atau bertukar (jika variansi sepanjang dua sumbu ini mempunyai nilai yang cukup dekat), tetapi secara umum tidak akan merubah bidang yang dijangkau (*span*) kedua vektor tersebut.

Bagaimanakan menemukan PC dari sebuah training set? Seperti yang sudah kita ketahui, terda-

pat teknik faktorisasi matriks standar yang disebut dengan *Singular Value Decomposition* (SVD). SVD dapat mendekomposisi matriks training set X ke dalam perkalian tiga matriks $X\Sigma V^T$, dimana V adalah matriks yang berisi unit vektor-unit vektor yang mendefinisikan PC yang kita cari, seperti yang diperlihatkan pada Persamaan (7.1).

Persamaan (7.1). Matriks *principle componet (*Principle Component Matrix*)

$$V = \begin{bmatrix} | & | & & | \\ c_1 & c_2 & \cdots & c_n \\ | & | & & | \end{bmatrix}$$

Kode `Python` berikut menggunakan fungsi`SVD()` pada `Numpy` untuk mendapatkan PC dari traning set, dan mengekstraksi dua unit vektor yang mendefinisikan 2 PC pertama.

- Membuat data sintetik 3D

[72]:
```
import numpy as np
np.random.seed(4)
m = 60
w1, w2 = 0.1, 0.3
noise = 0.1

angles = np.random.rand(m) * 3 * np.pi / 2 - 0.5
X = np.empty((m, 3))
X[:, 0] = np.cos(angles) + np.sin(angles)/2 + noise * np.random.randn(m) / 2
X[:, 1] = np.sin(angles) * 0.7 + noise * np.random.randn(m) / 2
X[:, 2] = X[:, 0] * w1 + X[:, 1] * w2 + noise * np.random.randn(m)
```

- Menggunakan SVD dan ekstraksi dua unit vektor pertama

[73]:
```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

[74]:
```
print(c1, c2)
```

```
[0.93636116 0.29854881 0.18465208] [-0.34027485  0.90119108  0.2684542 ]
```

Perintah `X_centered = X - X.mean(axis=0)` digunakan karena PCA mengasumsikan dataset terpusat pada titik origin, mis. (0,0) pada dua dimensi. Ketika kita menggunakan `Scikit Learn` dengan *class* `PCA` maka pemusatan ini akan dilakukan secara otomatis dengan `Scikit Learn`. Jika anda mengimplementasikan PCA sendiri, langkah pemusatan ini tidak boleh dilupakan.

3.3 Memproyeksikan ke d Dimensi

Setelah kita mengidentifikasi PC, kita bisa melakukan pengurangan dimensi dari dataset menjadi d dimensi dengan memproyeksikannya ke dalam *hyperplane* yang terdefinisi oleh sejumlah d dimensi pertama dari PC. Memilih *hyperplane* ini memastikan bahwa proyeksi akan mempertahankan variansi data sebanyak mungkin. Misalkan, pada Gambar 7.2 dataset 3D akan diproyek-

sikan ke dalam bidang 2D terdefinisi oleh 2 PC yang mempertahankan variansi bagian terbesar dari dataset. Sebagai hasilnya, proyeksi 2D terlihat sangat serupa dengandata aslinya di 2D.

Untuk memproyeksikan training set ke dalam *hyperplane* dan mendapatkan dataset yang telah dikurangi X_{d-proj} dari d dimensi, maka hitunglah matriks perkalian dari matriks training set X dengan matriks W_d yang berisi sejumlah d kolom pertama dari matriks V, seperti yang ditunjukan pada Persamaan (7.2).

Persamaan (7.2). Memproyeksikan training set X ke dalam dimensi yang lebih kecil d

$$X_{d-proj} = XW_d$$

Kode `Python` berikut akan memproyeksikan training set ke dalam sebuah bidang yang didefinisikan oleh 2 PC.

[75]:
```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

[76]:
```
X2D_using_svd = X2D
```

3.4 Menggunakan `Scikit Learn`

Class `Scikit-Learn` menggunakan dekomposisi SVD dalam mengimplementasikan PCA, seperti yang kita lihat di atas. Kode berikut mengimplementasikan PCA untuk mengurangi dimensi dataset menjadi dua dimensi (catatan bahwa `Scikit-Learn` secara otomatis mengimplementasikan pemusatan pada data).

[77]:
```
from sklearn.decomposition import PCA

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

[78]:
```
X2D[:5]
```

[78]:
```
array([[ 1.26203346,  0.42067648],
       [-0.08001485, -0.35272239],
       [ 1.17545763,  0.36085729],
       [ 0.89305601, -0.30862856],
       [ 0.73016287, -0.25404049]])
```

[79]:
```
X2D_using_svd[:5]
```

[79]:
```
array([[-1.26203346, -0.42067648],
       [ 0.08001485,  0.35272239],
       [-1.17545763, -0.36085729],
       [-0.89305601,  0.30862856],
       [-0.73016287,  0.25404049]])
```

Setelah *fitting transform* pada dataset, PCA memberikan akses pada PC.

[80]:
```
pca.components_
```

[80]:
```
array([[-0.93636116, -0.29854881, -0.18465208],
       [ 0.34027485, -0.90119108, -0.2684542 ]])
```

Bandingkan dengan dua PC yang dihitung menggunakan metode SVD berikut.

[81]:
```
Vt[:2]
```

[81]:
```
array([[ 0.93636116,  0.29854881,  0.18465208],
       [-0.34027485,  0.90119108,  0.2684542 ]])
```

Dapat kita perhatikan bahwa sumbunya terbalik (positif jadi negatif dan sebaliknya).

3.5 *Explained Variance Ratio*

Informasi lain yang bermanfaat adalah rasio variansi dari setiap *principle component* (*explained variance ratio*), yang dapat diakses melalui variabel `explained_variance_ratio_`. Rasio ini menunjukan proporsi variansi dataset yang terletak sepanjang tiap *principle component*. Misalkan, kita akan melihat rasio variansi dari dua PC berdasarkan dataset 3D pada Gambar 7.2.

[82]:
```
pca.explained_variance_ratio_
```

[82]:
```
array([0.84248607, 0.14631839])
```

Output di atas menyatakan bahwa 84,2% dari variansi dataset terletak sepanjang PC pertama, dan 14,6% terletak sepanjang PC kedua, dan sisanya 1,2% berada pada PC ketiga. Oleh sebab itu cukup beralasan jika mengasumsikan PC ketiga hanya membawa sedikit informasi.

3.6 Memilih Jumlah Dimensi yang Tepat

Daripada memilih jumlah dimensi secara sembarang, akan lebih sederhana memilih jumlah dimensi berdasarkan porsi hasil penjumlahan yang cukup besar dari variansi (misalkan 95%. Kecuali untuk keperluan visualisasi data, biasanya kita ingin mengurangi dimensi menjadi sekitar 2 atau 3. Kode berikut menunjukan PCA tanpa mengurangi jumlah dimensi, kemudian menghitung jumlah dimensi yang dibutuhkan untu mempertahankan variansi sebesar 95% dari variansi training set.

[83]:
```
pca = PCA()
pca.fit(X)
```

[83]:
```
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
```

[84]:
```
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95)+1
```

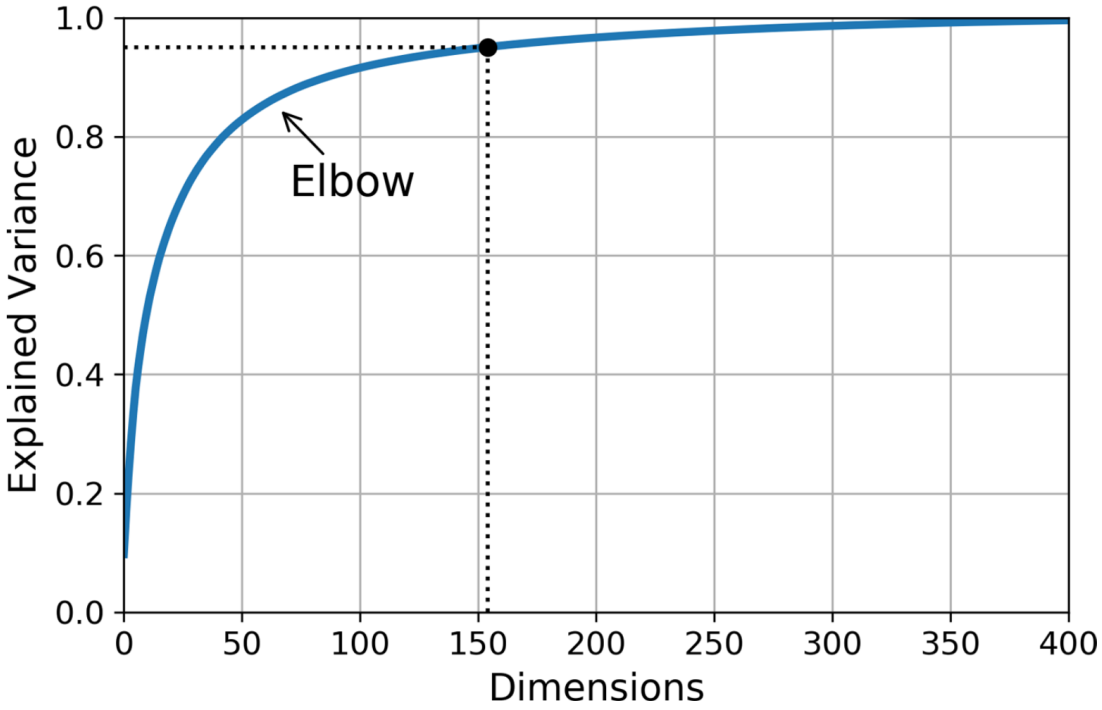
[85]:
```
d
```

[85]: 2

Terlihat bahwa dimensi yang akan mempertahankan variansi dataset sampai dengan 95% adalah $d = 2$. Kemudian kita dapat menset `n_components = d` dan mengeksekusi PCA kembali. Tetapi ada opsi yang jauh lebih baik, yaitu daripada menspesifikasikan jumlah PCA yang ingin kita pertahankan, kita dapat menset bilangan *float* antara $0,0$ dan $1,0$ yang mengindikasikan rasio variansi yang ingin kita pertahankan.

[86]:
```
pca = PCA(n_components = 0.95)
X_reduced = pca.fit_transform(X)
```

Opsi lain adalah memplot *explained variance* sebagai fungi dari jumlah dimensi (plot `cumsum`), seperti yang ditunjukan pada Gambar 7.8. Dengan cara tersebut akan terdapa lengkungan (*elbow*) pada kurva (ingat *inertia* pada saat memilih jumlah cluster pada *K-Means*), dimana *explained variance stop* berhenti naik dengan cepat. Pada kasus ini, kita lihat bahwa pengurangan dimensi sampai dengan 100 dimensi tidak akan kehilangan terlalu banyak variansi dataset.



Gambar 7.8: Explained variance sebagai fungsi dari jumlah dimensi

3.7 Kompresi dengan PCA

Setelah pengurangan dimensi, maka ukuran training set otomatis akan berkurang. Sebagai contoh, jika kita mengimplementasikan PCA pada dataset MNIST dengan mempertahankan 95% variansi dataset, maka akan ditemukan setiap *instance* hanya akan mempunyai jumlah *feature* sebanyak 150 dibandingkan jumlah *feature* original sebanyak 784. Sehingga, dataset hasil pengurangan dimensi lebih kecil dari 20% ukuran dataset original, yang merupakan rasio kompresi yang sangat baik. Pengurangan dimensi ini akan mempercepat algoritma klasifikasi (seperti SVM) yang digunakan untuk dijit tulis tangan secara signifikan.

Pada kasus ini dimungkinkan juga untuk melakukan dekompresi dataset yang dimensinya terkurangi, kembali menjadi berdimensi 784 dengan menerapkan transformasi balik (*inverse transformation*) pada proyeksi PCA menggunakan Persamaan (7.2). Cara ini tidak akan mengembalikan data original secara sempurna karena mekanisme proyeksi akan menghilangkan sejumlah informasi (variansi sekitar 5% akan hilang). Tetapi data hasil dekompresi (rekonstruksi) tetap akan cukup dekat dengan data original. Jarak rata-rata kuadrat (*mean squared distance*) antara data original dan data hasil rekonstruksi (setelah proses kompresi dan dekompresi) disebut dengan **error rekonstruksi** (***reconstuction error***).

Persamaan (7.3). Transformasi balik PCA, mengembalikan pada dimensi original

$$X_{recovered} = X_{d-project} W_d^T$$

Kode berikut akan melakukan kompresi dataset MNIST mejadi 154 dimensi (dari 784), kemudian menggunakan metode dekompresi untuk mengembalikannya ke dataset semula dengan 784 dimensi.

- **Load Dataset dan memisahkan menjadi data training dan data test**

[87]:
```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', version=1)
mnist.target = mnist.target.astype(np.uint8)
```

[88]:
```
from sklearn.model_selection import train_test_split

X = mnist["data"]
y = mnist["target"]

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

- **Mempertahankan** 95% **dari variansi**

[89]:
```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

[90]:
```
pca.n_components_
```

[90]:
```
154
```

[91]:
```
np.sum(pca.explained_variance_ratio_)
```

[91]:
```
0.9503684424557437
```

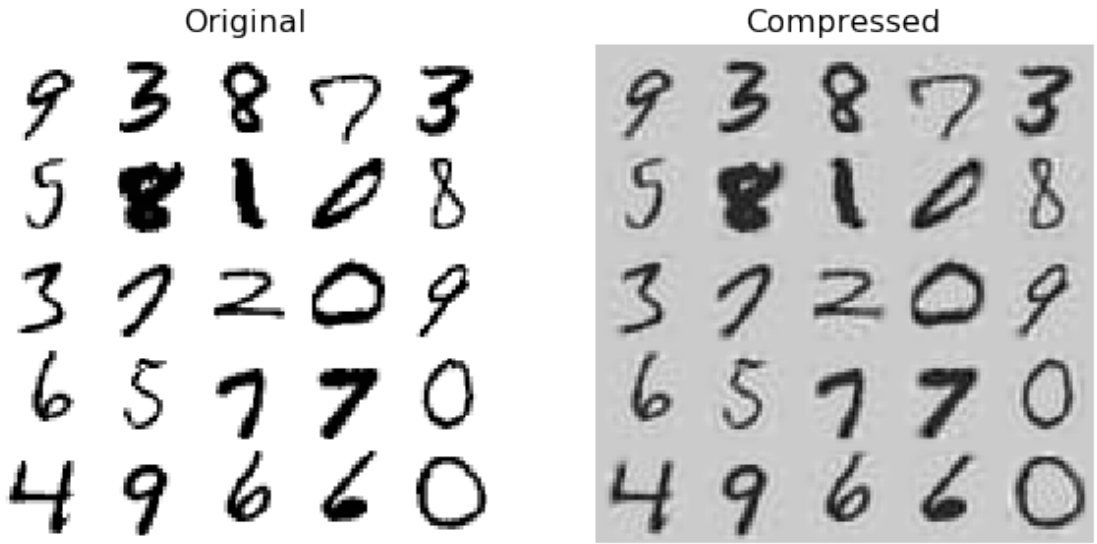
[92]:
```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

- **Perbandingan gambar dijit original dan hasil rekonstruksi**

[93]:
```
def plot_digits(instances, images_per_row=5, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = mpl.cm.binary, **options)
    plt.axis("off")
```

[95]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
plt.subplot(121)
plot_digits(X_train[::2100])
plt.title("Original", fontsize=16)
plt.subplot(122)
plot_digits(X_recovered[::2100])
plt.title("Compressed", fontsize=16)
```

[95]:
```
Text(0.5, 1.0, 'Compressed')
```



[96]:
```
X_reduced_pca = X_reduced
```

Gambar di atas menunjukan beberapa dijit dari dataset original (gambar kiri), dan dataset hasil rekonstruksi (kompresi dan dekompresi). Gambar ini menunjukan adanya sedikit kehilangan kualitas image, tetapi dijit-dijit sebagian besar masih sesuai dengan dataset original.

3.8 PCA yang diacak (*Randomized PCA*)

Jika kita set *hyperparameter* `svd_solver="randomized"`, `Scikit Learn` akan menggunakan algoritma stokastik yang disebut dengan ***Randomized PCA*** yang dapat secara cepat menemukan sejumlah d PC pertama. Kompleksitas komputasinya adalah $O\left(m \times d^2\right) + O\left(d^3\right)$, sedangkan dengan menggunakan full SVD kompleksitas komputasi adalah $O\left(m \times n^2\right) + O\left(n^3\right)$. Sehingga *randomized PCA* jauh lebih cepat dibandingkan dengan full SVD ketika $d < n$.

[97]:
```
rnd_pca = PCA(n_components = 154, svd_solver = "randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

Harga default dari `svd_solver` adalah `"auto"`, dimana `Scikit Learn` secara otomatis menggunakan algoritma *randomized PCA* dengan syaratt jumlah training m atau jumlah *feature* n lebih besar dari 500 dan d lebih kecil dari 80% dari m atau n. Jikat syarat tersebut tidak terpenuhi maka full SVD akan digunakan. Jika ingin memaksa `Scikit Learn` menggunakan full SVD, kita bisa menset 'svd_solver="full"'.

3.9 *Incremental PCA*

Salah satu permasalahan pada implementasi-implementasi PCA sebelumnya adalah mereka membutuhkan keseluruhan training set masuk ke dalam memori untuk menjalankan algoritma. Algoritma ***Incremental PCA*(IPCA)** dibuat untuk mengatasi permasalahan ini, dimana training set dapat dipecah-pecah menjadi *mini-batch* dan satu persatu dapat dijadikan input pada algoritma IPCA.

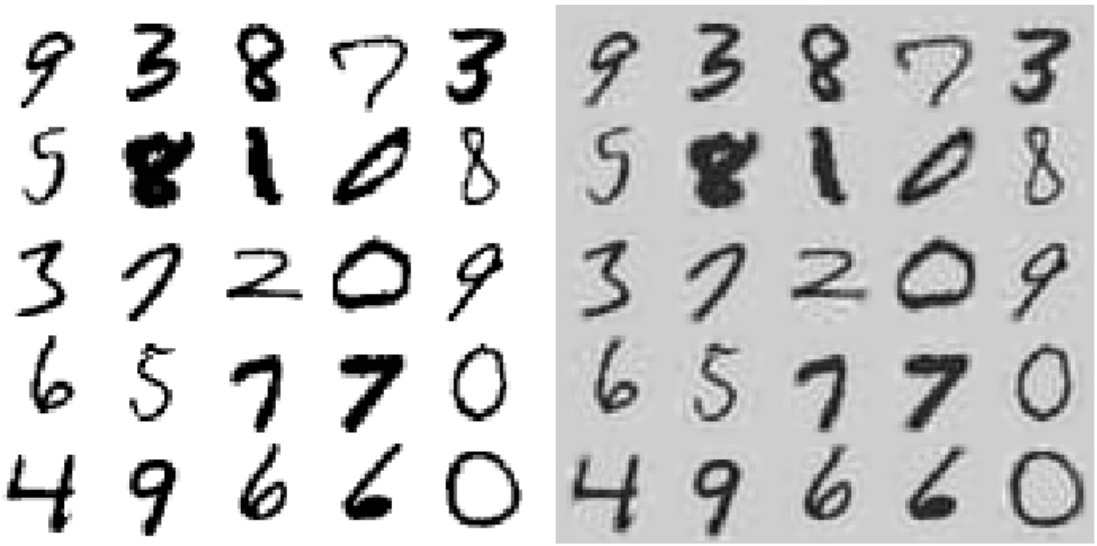
Metode ini sangat bermanfaat untuk training set yang sangat besar dan untuk mengimplementasikan PCA secara onlie (*on the fly* saat *instance* baru datang). Kode berikut memisahakan dataset MNIST ke dalam 100 *mini-batches* (menggunakan fungsi `array_split()` dari `Numpy`) dan dimasukan ke dalam *class* `IncrementalPCA` dari `Scikit Learn` untuk mengurangi dimensi dataset MNIST menjadi 154 dimensi. Sebagai catatan kita harus memanggil metode `partial_fit()` untuk tiap *mini-batch*, sebagai pengganti metode `fit()` yang menggunakan keseluruhan dataset sekaligus.

[98]:
```
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):
    print(".", end="") # not shown in the book
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

```
...
...
```

[99]:
```
X_recovered_inc_pca = inc_pca.inverse_transform(X_reduced)
```

[100]:
```
plt.figure(figsize=(10, 5))
plt.subplot(121)
plot_digits(X_train[::2100])
plt.subplot(122)
plot_digits(X_recovered_inc_pca[::2100])
plt.tight_layout()
```



[101]:
```
X_reduced_inc_pca = X_reduced
```

Jika kita bandingkan hasil transformasi MNIST menggunakan PCA reguler dan *incremental PCA*, maka hasil untuk *means* akan sama.

[102]:
```
np.allclose(pca.mean_, inc_pca.mean_)
```

[102]:
```
True
```

Tetapi hasil tidak betul-betul identik. *Incremental PCA* memberikan solusi aproksimasi yang sangat bagus, tetapi tidak sempurna.

[103]:
```
np.allclose(X_reduced_pca, X_reduced_inc_pca)
```

[103]:
```
False
```

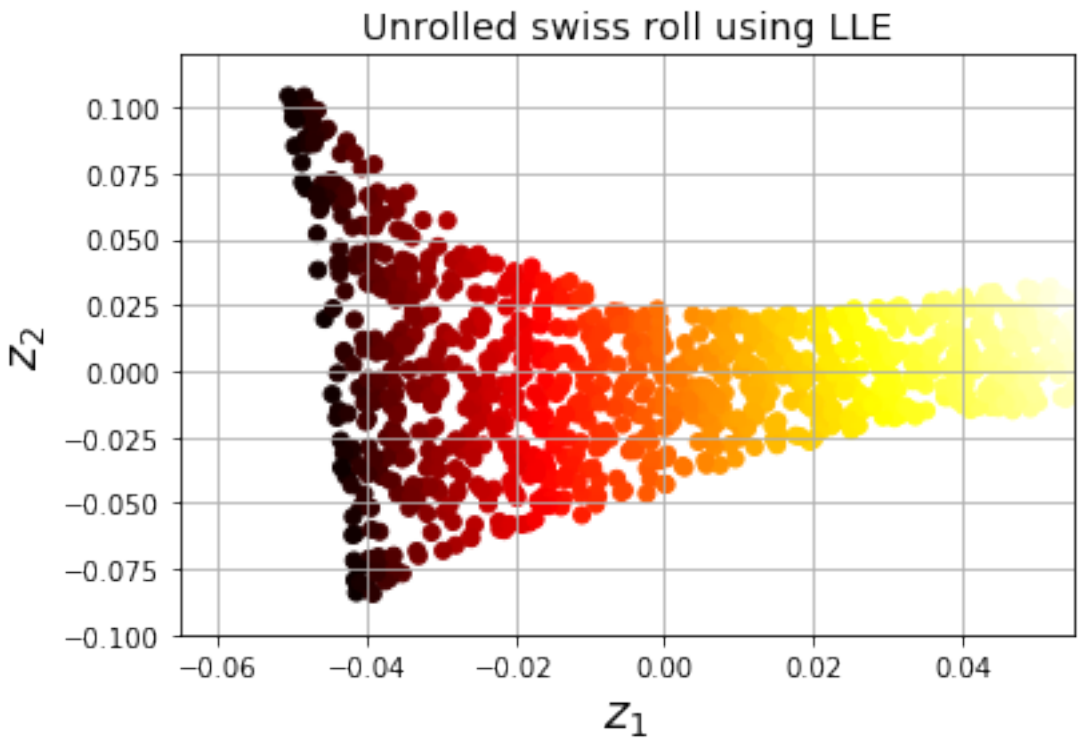
4 *Locally Linear Embedding (LLE)*

LLE merupakan teknik pengurangan dimensi nonlinier (*nonlinear dimensionality reduction*) yang menggunakan metode *manifold learning* yang tidak bergantung pada proyeksi seperti PCA. LLE

bekerja pertama dengan mengukur bagaimana masing-masing training *instance* berhubungan secara linier satu sama lain dengan tetangga-tetangga terdekatnya. Kemudian setelahnya mencari representasi training set pada dimensi yang lebih rendah tetapi hubungan lokal antar *instance* tetap dijaga maksimal. Pendekatan ini khususnya baik untuk *unrolling twisted manifold*, terutama ketika tidak terlalu banyak noise.

[12]:
```
        from sklearn.manifold import LocallyLinearEmbedding

        lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10,␣
↪random_state=42)
        X_reduced = lle.fit_transform(X)
```

[13]:
```
        plt.title("Unrolled swiss roll using LLE", fontsize=14)
        plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot)
        plt.xlabel("$z_1$", fontsize=18)
        plt.ylabel("$z_2$", fontsize=18)
        plt.axis([-0.065, 0.055, -0.1, 0.12])
        plt.grid(True)
        plt.show()
```



Gambar di atas menunjukan hasil *unroll* dari dataset *Swiss roll* pada dimensi 2D. Dapat kita lihat bahwa *Swiss roll* betul-betul di-*unroll* dan jarak antara *instances* secara lokal tetap terjaga. Tetapi jarak tidak terjaga untuk skala yang lebih besar. Bagian sebelah kiri dari dataset *Swiss roll* original

mengalami peregangan, sedangkan bagian sebelah kanan mengalami pemampatan. Kendatipun demikian, LLE sudah memberikan kinerja yang baik untuk memodelkan *manifold*.

Berikut cara kerja LLE. Untuk setiap training *instance* $x^{(i)}$, algoritma mengidentifikasi sejumlah k tetangga terdekat (pada kode sebelumnya $k = 10$ `n_neighbors=10`), kemudian mencoba untuk merekonstruksi $x^{(i)}$ sebagai fungsi linier dari tetangga-tetangga terdekat ini. Secara khusus, algoritma menemukan bobot $w_{i,j}$ sehingga kuadrat jarak antara $x^{(j)}$ dan $\sum_{j=1}^{m} w_{i,j} x^{(j)}$ sekecil mungkin, dan mengasumsikan $w_{i,j} = 0$ jika $x^{(j)}$ bukan salah satu diantara k tetangga terdekat dari $x^{(i)}$. Sehingga langkah pertama dari LLE adalah masalah *constrained optimization* pada Persamaan (7.4).

Persamaan (7.4). Langkah pertama algoritma LLE

$$\hat{W} = \underset{W}{\operatorname{argmin}} \quad \sum_{i=1}^{m} \left(x_{(i)} - \sum_{j=1}^{m} w_{i,j} x^{(j)} \right)^2$$

$$\text{subject to} \quad \begin{cases} w_{i,j} = 0 \text{ jika } x^{(j)} \text{ bukan salah satu k tetangga terdekat dari } x^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1; \text{for } i = 1, \ldots, m \end{cases}$$

Setelah langkah ini, matriks pembobot $\hat{W}$ (yang berisi pembobot-pembobot $hatw_{i,j}$ mengkodekan hubungan linier antara training *instances*. Langkah kedua adalah memetakan training *instances* ke dalam ruang d dimensi (dimana $d < n$) dan pada saat yang bersamaan tetap menjaga hubungan lokal semaksimal mungkin. Jika $z^{(i)}$ adalah *image* dari $x^{(i)}$ pada ruang d dimensi, maka jarak kuadrat antara $z^{(i)}$ dan $\sum_{j=1}^{m} w_{i,j} z^{(j)}$ sekecil mungkin. Ide ini akan membawa kita pada permasalahan *constrained optimization* kembali yang dinyatakan dengan Persamaan (7.5).

Persamaan (7.5). Langkah kedua algoritma LLE

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

Berkebalikan dengan langkah yang pertama dimana menjaga *instance* tetap dan mencoba menemukan pembobot optimal, pada langkah dua menjaga pembobot tetap dan mencari posisi optimal dari image *instance* pada dimensi yang lebih kecil. Sebagai catatan Z adalah matriks yang berisi semua $z^{(i)}$.

Implementasi LLE pada `Scikit Learn` mempunyai kompleksitas komputasi $O\left(m \log(m) n \log k\right)$ untuk menemukan k tetangga terdekat, $O\left(mnk^3\right)$ untuk optimisasi bobot, dan $O\left(dm^2\right)$ untuk merekonstruksi representasi pada dimensi rendah. Tetapi faktor m^2 pada $O\left(dm^2\right)$ akan membuat algoritma buruk untuk penskalaan dengan dataset-dataset yang besar.