به نام خدا

خلاصه فصل ۸ کتاب

HANDS-ON MACHINE LEARNING

DIMENSIONALITY REDUCTION

- خیلی از مسائل machine learning شامل اینن که دیتاهاشون کلی فیچر داره هر نمونشون. این یه مشکل بزرگی که داره اینه باعث میشه فرایند یادگیری مدلمون خیلی کند بشه.
- مثلا اگه دیتاست MNIST رو بخوایم برسی کنیم، هر عکس یه سری پیکسل داشت که ۲۸ در ۲۸ بود. یه راه برای این مسئله اینه که بگیم پیکسل های نزدیک دور عکس همیشه سفیده پس میتونیم اینارو حذف کنیم. یا معمولا دو تا پیکسل کنار هم خیلی وابسته به همن. میتونیم دو تاشونو برداریم تبدیل کنیم به یه پیکسل بدون اینکه خیلی دیتای خاصی هم از دست بدیم.
- به طور کلی اینکه بیایم تعداد فیچر هارو کمتر کنیم همیشه کار خوبی نیست چون به هرحال یه سری دیتا داریم از دست میدیم و ممکنه پرفورمنس مدل رو بیاره پایین. البته بعضی وقتا هم باعث میشه یه سری نویز حذف بشه و پرفورمنس مدل حتی بالاتر هم بره.
- حتی یه خوبی دیگه هم این که اگه بتونیم بعد دیتارو تا ۳ تا بیاریم میتونیم plot کنیم و visualize نگاه کنیم دیتامون رو.

THE CURSE OF DIMENSIONALITY

- یه همچین نگرشی میتونیم از ابعاد مختلف تو فضا داشته باشیم:

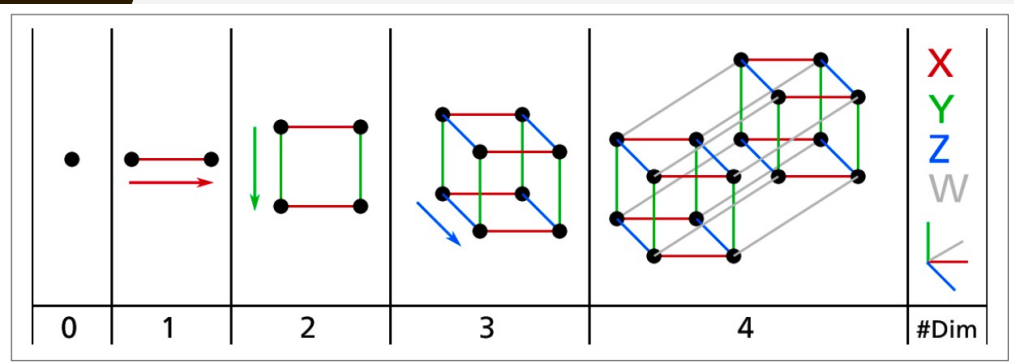

Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)[2]

یه نکته مهم که هستش اینه که به طور رندوم اگه دو تا نقطه تو ۲ بعد برداریم فاصله مساوی ۰.۵۲ دارن اما تو ۱۰۰۰۰ بعد این فاصله میرسه به ۴۰۸.۲۵. اینجا یه نکته خیلی مهم رو میفهمم که دیتاهامون وقتی تعداد فیچرشون بالا باشه(فضا بزرگی داشته باشن)
فاصلشون از هم خیلی زیاده و شباهتی به هم ندارن.و یه نمونه خیلی احتمال کمی داره با یه نمونه دیگه شبیه باشه. همین باعث چون generalization ضعیفی داریم احتمال overfit بره بالا

پس اگه بخوایم یه مدل خوبی برای این تایپ دیتاها داشته باشیم باید از تعداد نمونه های زیادی استفاده کنیم که متاسفانه تو دنیای واقعیت نمونه پیدا کردن خیلی کار ساده ای نیست. پس اهمیت dimension reduction رو فهمیدیم.

MAIN APPORACHES FOR DIMENSIONALITY REDUCTION

- به طور کلی دو روش وجود داره برای این کار: projection – manifold learning

Projection: تو اکثر مسائل دنیای واقعی، نمونه ها واقعا خیلی مستقل از هم نیستن. یعنی چی؟ یعنی اینکه اطلاعاتشون تو فضای n بعدی معمولا به هم مرتبطه و میتونیم دیتاهارو با یک زیر فضا کوچیک تر پوشش بدیم.

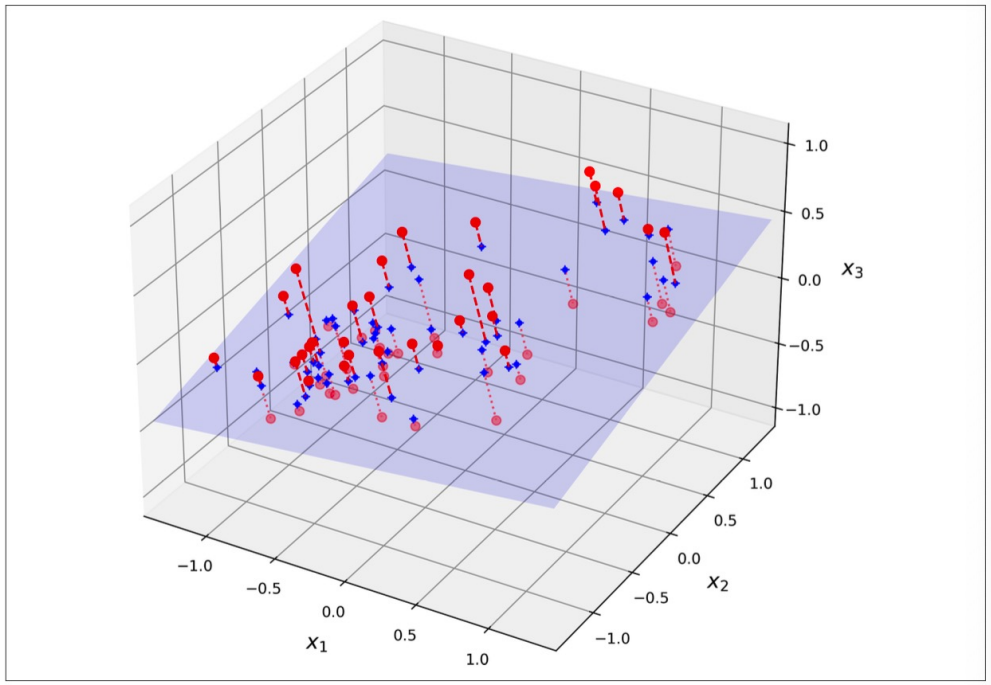

Figure 8-2. A 3D dataset lying close to a 2D subspace

الان داریم دیتاهای اصلی رو نشون میدیم با نقطه های قرمز میتونیم همشون رو بیاریم روی صفحه ای که مشخص کردیم که اون نقاط ابی رنگ میشه.

Figure 8-3. Ta-da! We have just reduced the dataset's dimensionality from 3D to 2D. Note that the axes correspond to new features z_1 and z_2: they are the coordinates of the projections on the plane.

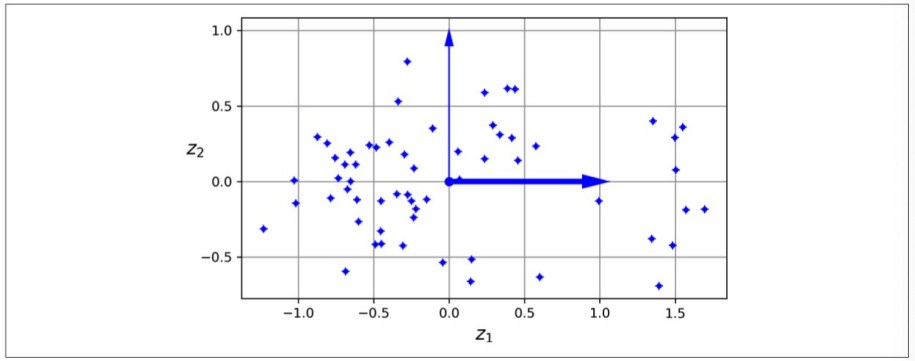

Figure 8-3. The new 2D dataset after projection

MANIFOLD LEARNING

- اما خب شاید نتونیم از projection تو بعضی از موارد مثل شکل زیر استفاده کنیم.

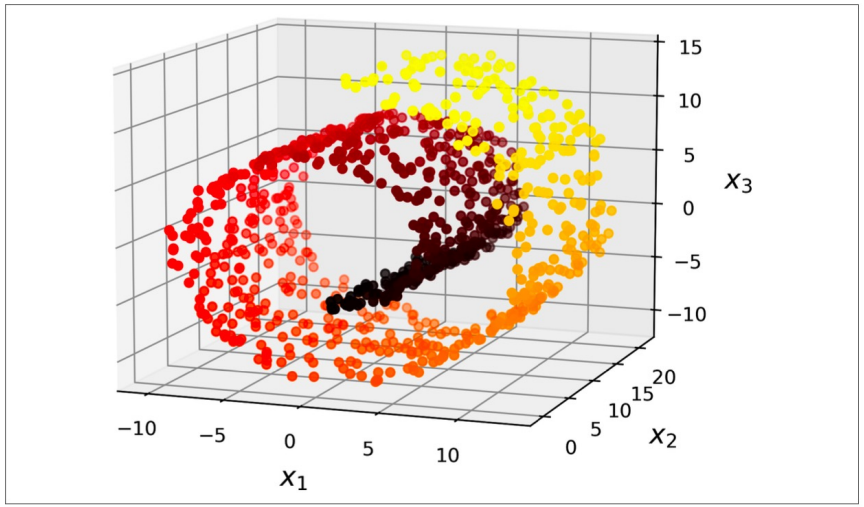

Figure 8-4. Swiss roll dataset

اگه بیایم از projection استفاده کنیم و مثلا x3 رو کلا حذف کنیم تا به یه صفحه برسیم دیتاها رو هم میفته که شکل سمت چپ رو درست میکنه در صورتی که ما برامون خوبه به شکل سمت راست برسه تا بتونیم به راحتی جداشون کنیم

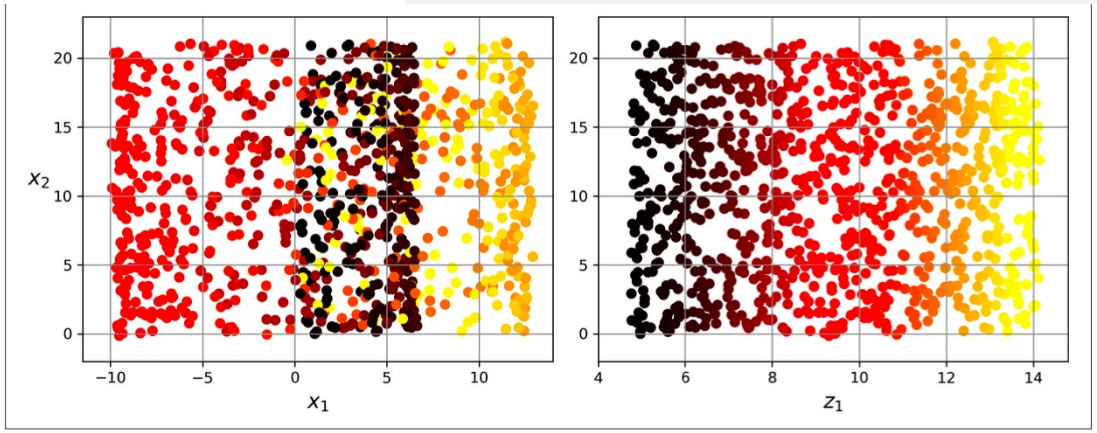

Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

CONTINUE

- در واقع شکل سمت راستی رو اینطوری میتونیم فرض بگیریم که یه کاغذ (2D) میتونیم به اون شکل اولیه تو ۳ بعد درش بیاریم. در واقع یک d-dimensional manifold یک جزئی از فضای n بعدی هستش. تو این مثال ما d=2 و n=3 هستش که به تنهایی یک صفحه ۲ بعدیه ولی میتونی به شکل یک همچین شکلی تو ۳ بعد دربیاد.
- خیلی از dimension reduction algorithms میان با استفاده از این روش روی training instance فیت میشن که به این کار manifold learning میگن.
- بخوایم یه مثال بزنیم تو دیتاست MNIST تقریبا یه سری فرض داریم که باعث میشه هر عکس رندومی درست کنیم به عنوان دست خط تلقی نشه. مثلا اینکه حاشیه های عکس سفیده. خط های متصل به هم داره و ... که باعث میشه یه زیر فضای کوچیک تری از ۳ بعد رو شامل بشه نمونه دست خط ها.

CONTINUE

- البته این manifold کردن همیشه جواب نیست. اینجا دو تا مثال میزنیم که برای بالایی جوابه اما برای پایینی تو ۳بعد کار کردن راحت تره. خلاصش اینه dimension reduction خیلی به دیتاست ربط داره که باعث بشه کارمون راحت تر بشه یا نه.

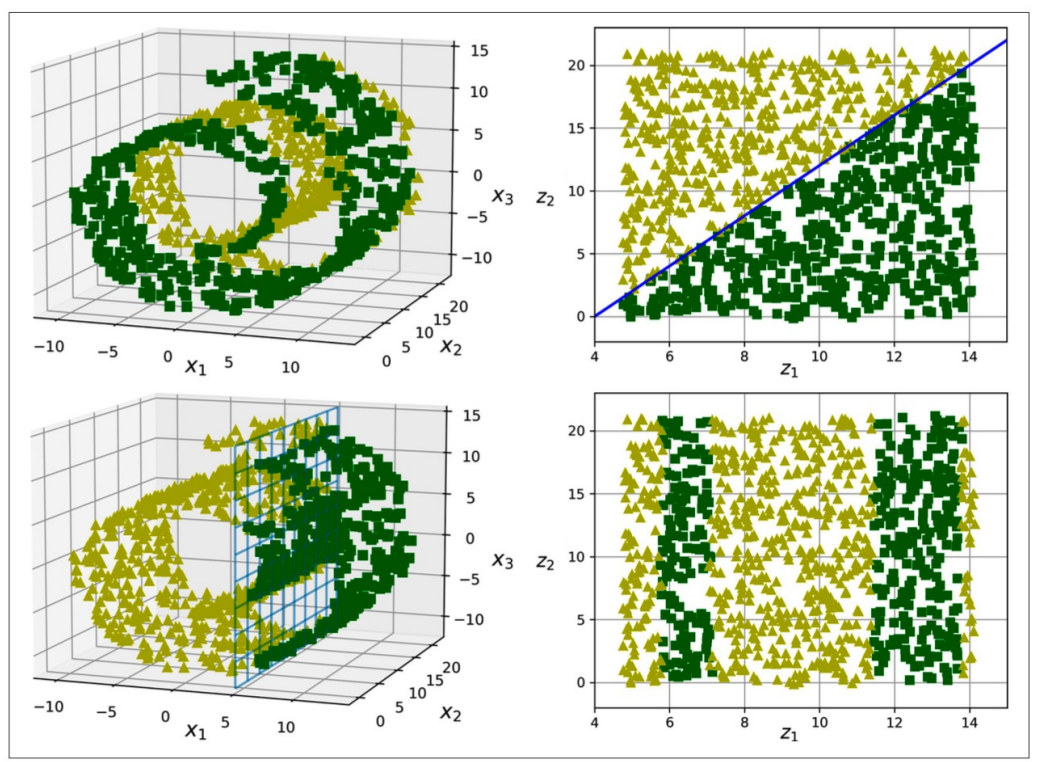

Figure 8-6. The decision boundary may not always be simpler with lower dimensions

PCA

- یکی از مهم ترین الگوریتم ها تو این حوزه pca یا همون principal component analysis هستش. کاری که میکنه اینه که اول یه hyperplane که خیلی به نمونه هامون نزدیک باشه پیدا میکنه و بعد project میکنه دیتاها روی hyperplaneمون.
- اولین قدم برای این کار اینه که بتونیم یه hyperplane مناسب برای این کار انتخاب کنیم. یعنی چی این؟ شکل زیر نشون میده چطور میتونه ۳ تا محور متفاوت یه سری دیتا رو نمایش بدن.

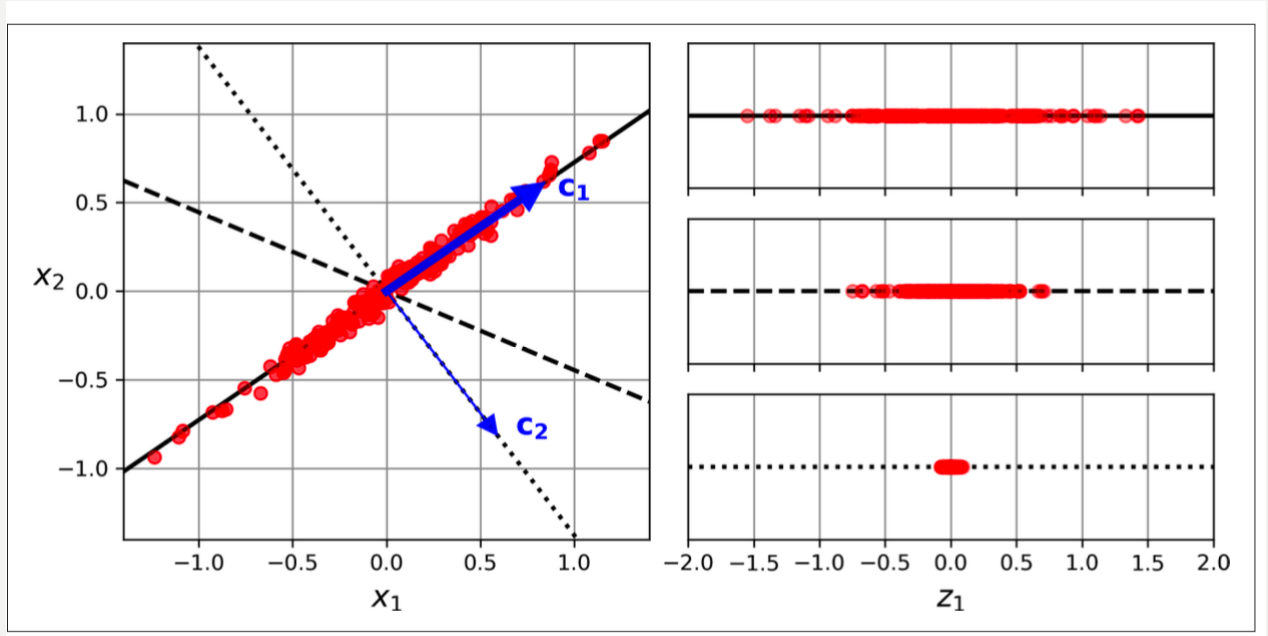

Figure 8-7. Selecting the subspace on which to project

CONTINUE

- همینجور که دیدم بالاترین محور تونسته به شکل خوبی واریانس دیتاهارو نگه داره و اون پراکندگی رو همچنان حفظ کنه اما دو تای دیگه همه دیتاها رو نزدیک به هم کرده و شبیه به هم شدن. در واقع بخوایم ایده کلی پشت pca رو بگیم اینه بتونیم MSE دیتاهامون تو فضای جدید و دیتاهامون رو مینیمم کنیم.

- Principal Components: کاری که pca برامون انجام میده اینه بیاد محوری که بیشترین واریانس رو داره برامون پیدا کنه. همچنین فقط به یه محور نیاز نداریم. میاد یه خط هم به محور اصلی عمود میکنه تا بیشترین واریانس رو تو یه بعد دیگه هندل کنه. اگه داخل فضای ۳ بعدی باشیم میتونیم دنبال یه محور سوم باشیم که به این دو محور عمود باشه و دوباره بیشترین واریانس رو داشته باشه. به هر کدوم از این محور ها principal componenet میگن.

CONTINUE

- اما سوالی که پیش میاد اینه که این محور هارو از کجا پیدا کنیم؟ با استفاده از تجزیه svd میتونیم ماتریس training set رو تجزیه کنیم.

that can decompose the training set matrix **X** into the matrix multiplication of three matrices **U** Σ $\mathbf{V}^T$, where **V** contains the unit vectors that define all the principal components that you are looking for, as shown in Equation 8-1.

Equation 8-1. Principal components matrix

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

صرفا دقت داریم که مطمئین بشیم دیتامون رو از میانگین کم کنیم بعد تجزیه بزنیم روش.

```
import numpy as np

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

PCA assumes that the dataset is centered around the origin. As you will see, Scikit-Learn's PCA classes take care of centering the data for you. If you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first.

PROJECTING DOWN TO D DIMENSIONS

- حالا که تونستیم مجور های اصلی رو پیدا کنیم میتونیم دیتامون رو به یه فضای d بعدی به طوری که d<=n هستش ببریم. برای مثال اگه فضای اولیه ۳ بعدی باشه و بیایم ۲ بردار اول داخل تجزیه svd رو انتخاب کنیم میتونیم یه صفحه ۲ بعدی داشته باشیم و دیتا رو روی این صفحه نمایش بدیم.
- برای این کار باید ضرب ماتریسی دیتاست روی ماتریسی که شامل شامل ۲ بردار ما هستن انجام بدیم.

Equation 8-2. Projecting the training set down to d dimensions

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

The following Python code projects the training set onto the plane defined by the first two principal components:

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

همین کارایی که ما کردیم رو خود scikit-learn میاد برامون انجام میده راحت تر.

```
from sklearn.decomposition import PCA

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

EXPLAINED VARIANCE RATIO

- یه اطلاعات خیلی مفیدی که میتونیم از روی تجزیه بگیریم اینه که میزان واریانس داده ها روی هر محور چقدر هستش.

```
>>> pca.explained_variance_ratio_
array([0.7578477 , 0.15186921])
```

- این نتیجه داره بهمون میگه ۷۶ درصد واریانس دیتاست روی اولین محوریه که بدست اوردیم و دومین محور ۱۵ درصد. در نتیجه سومین محور ۹ درصد که با این ۳ محور میتونیم کل دیتاست رو تو همون فضای ۳ بعد نگه داریم.
- نکته ای که وجود داریم اینه از کجا بفهمیم چند تا محور باید انتخاب کنیم؟ یه راه اینه بیایم بگیم به همون تعدادی محور نیاز داریم که مثلا ۹۵ درصد واریانس رو بتونه برامون حفظ کنه.

CONTINUE

```python
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
```

- کد زیر میاد همچین کاری برامون میکنه.

- همچنین میتونیم مستقیم همین خواسته که ۹۵ درصد واریانس رو حفظ کنه به pca بگیم.

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

The actual number of components is determined during training, and it is stored in the n_components_ attribute:

```
>>> pca.n_components_
154
```

که میبینیم همون نتیجه کد بالایی رو باز برامون درست کرده

CONTINUE

- یه روش دیگه هم اینه که بیایم نمودار واریانس برحسب تعداد محور ها نشون بدیم. از یه جا به بعد دیگه خیلی درصدمون بالاتر خیلی نمیره اما تعداد محور ها خیلی رشد میکنه که میتونیم از همون نقطه بذاریم برای تعداد محور هایی که نیاز داریم.

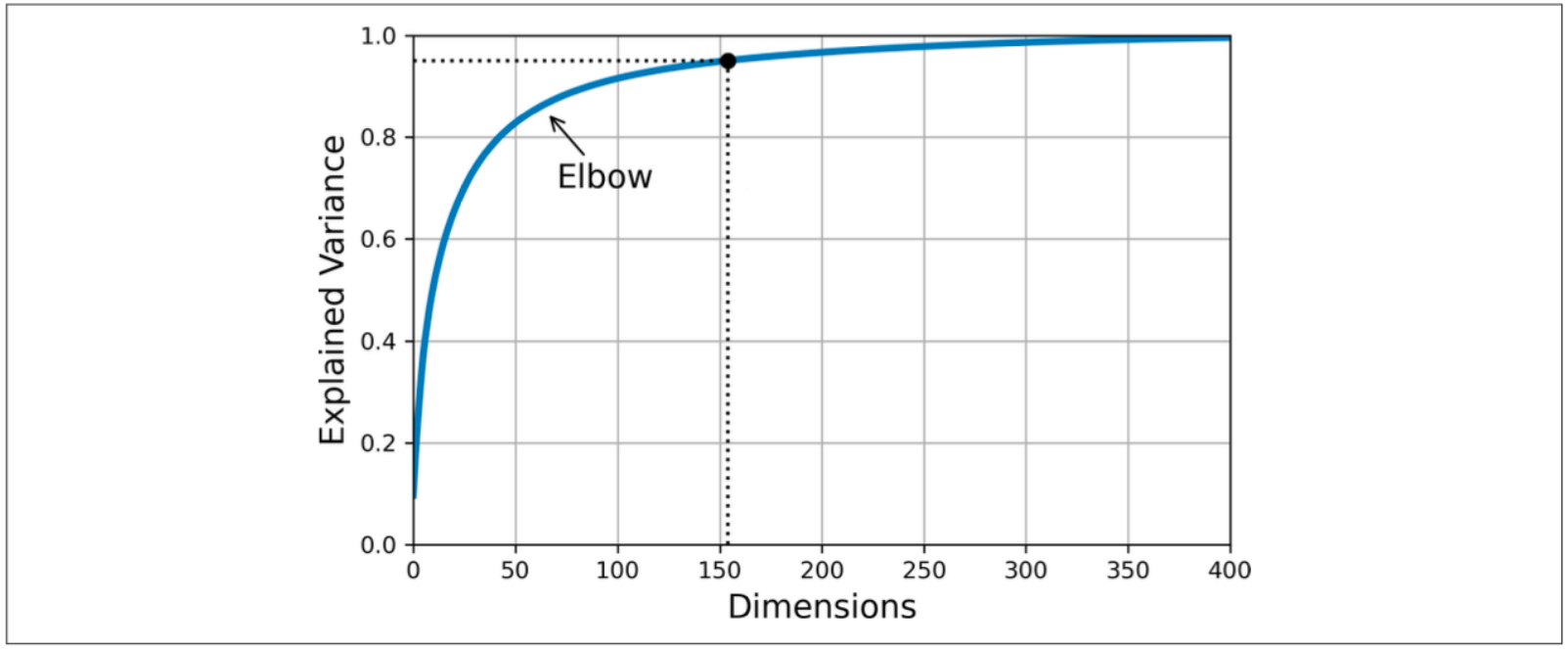

Figure 8-8. Explained variance as a function of the number of dimensions

CONTINUE

- همچنین میتونیم تعداد محور ها رو با استفاده randomizedSearch پیدا کنیم و حتی این فرایند رو بذاریم داخل یه pipeline.

```
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])
```

Let's look at the best hyperparameters found:

```
>>> print(rnd_search.best_params_)
{'randomforestclassifier__n_estimators': 465, 'pca__n_components': 23}
```

PCA FOR COMPRESSION

- یه کار دیگه ای که میتونیم بکنیم با pca اینه که حجم دیتاها رو کم کنیم. دیدیم که روی دیتاست MNIST دیتاست با ۱۵۴ تا فیچر میتونیم ۹۵ درصد واریانس داده رو نگه داریم و دیگه به اون ۷۸۴ فیچر اولیه نیاز نداریم. خیلی درصد خوبیه که با ۲۰ درصد فیچر ها میتونیم ۹۵ درصد اطلاعات دیتاها رو نگه داریم.

میتونیم با استفاده از inverse گرفتن از دیتایی که حجمش رو کمتر کردیم دوباره به ۷۸۴ فیچر برسیم ولی نکته ای که هست دقیقا دیتای اصلی نمیشه چون تو فرایند کم کردن حجم دیدیم ۵ درصد اطلاعات رو کلا از دست میدیم ولی خب نزدیکه به دیتای اصلی. به میانگین اختلاف بین دیتای اصلی و ریکاور شده reconstruction error میگن.

The `inverse_transform()` method lets us decompress the reduced MNIST dataset back to 784 dimensions:

```
X_recovered = pca.inverse_transform(X_reduced)
```

Figure 8-9 shows a few digits from the original training set (on the left), and the corresponding digits after compression and decompression. You can see that there is a slight image quality loss, but the digits are still mostly intact.

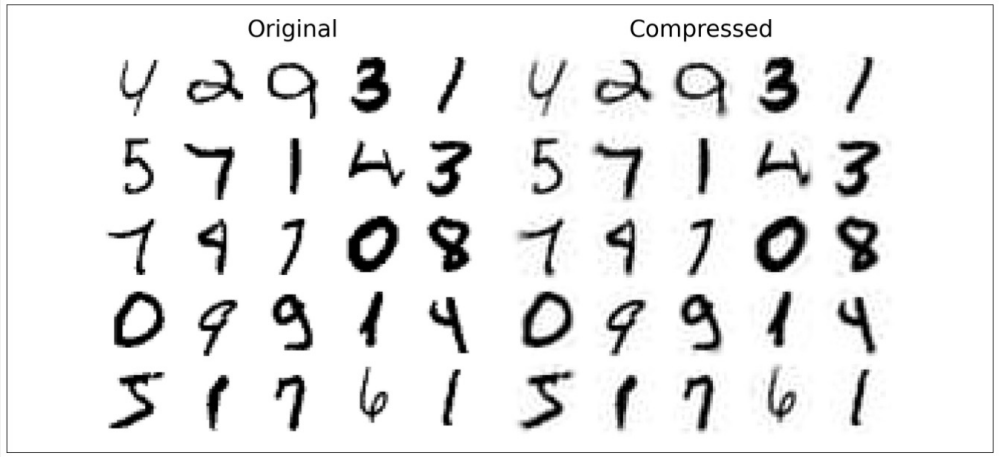

Figure 8-9. MNIST compression that preserves 95% of the variance

The equation for the inverse transformation is shown in Equation 8-3.

Equation 8-3. PCA inverse transformation, back to the original number of dimensions

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

RANDOMIZED

- صرفا اومده یه روش گفته هست که سریعتر از svd عادیه ولی دقتش کمتره.

If you set the `svd_solver` hyperparameter to `"randomized"`, Scikit-Learn uses a stochastic algorithm called *randomized PCA* that quickly finds an approximation of the first d principal components. Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$ for the full SVD approach, so it is dramatically faster than full SVD when d is much smaller than n:

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

By default, `svd_solver` is actually set to `"auto"`: Scikit-Learn automatically uses the randomized PCA algorithm if $\max(m, n) > 500$ and `n_components` is an integer smaller than 80% of $\min(m, n)$, or else it uses the full SVD approach. So the preceding code would use the randomized PCA algorithm even if you removed the `svd_solver="randomized"` argument, since $154 < 0.8 \times 784$. If you want to force Scikit-Learn to use full SVD for a slightly more precise result, you can set the `svd_solver` hyperparameter to `"full"`.

INCREMENTAL PCA

- یه مشکل pca اینه که کل دیتاست رو باید داخل موری نگه داریم چون همه این دیتاست رو باید داخل یه ماتریس فیت کنیم. یه incremental PCA داریم که میاد training set رو به mini-batch های مختلف تقسیم میکنه. کد زیر میاد MNIST رو به ۱۰۰ تا mini-batch تقسیم میکنیم و به IPCA پاس میدیم.

before. Note that you must call the `partial_fit()` method with each mini-batch, rather than the `fit()` method with the whole training set:

```
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)
```

RANDOM PROJECTION

- تا اینجا انواع pca رو گفتیم یه نکته بدی داره اینه که هرچقدرم از ipca استفاده کنیم بلاخره با دیتاهایی که تو رنج صدهزار فیچر دارن مثل عکس ها pca خیلی اروم عمل میکنه. اینجا باید random projection بزنیم.
- همینجور که از اسمش مشخصه میاد به طور رندوم یه projection خطی میزنیم تا ابعاد رو کمتر کنیم. شاید به نظر برسه که این کار خیلی خوب نباشه ولی به طور ریاضیاتی نشون دادن نمونه هایی که از هم دور بودن تو فضای قبل از کم کردن، همچنان تو فضای جدید دورن از هم یا نمونه های شبیه به هم همچنان شبیه به هم میمونن. اما همونطور که قبل مطرح بود باید اینجا هم مطرح کنیم که چند تا بعد نیاز داریم.

ability—that distances won't change by more than a given tolerance. For example, if you have a dataset containing m = 5,000 instances with n = 20,000 features each, and you don't want the squared distance between any two instances to change by more than ε = 10%,[6] then you should project the data down to d dimensions, with $d \geq 4 \log(m) / (½ \varepsilon^2 - ⅓ \varepsilon^3)$, which is 7,300 dimensions. That's quite a significant dimensionality reduction! Notice that the equation does not use n, it only relies on m and ε. This equation is implemented by the `johnson_lindenstrauss_min_dim()` function:

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

CONTINUE

- حالا که d رو پیدا کردیم، چطوری دیتای قدیمی رو دیتای جدید مپ کنیم؟

Now we can just generate a random matrix **P** of shape $[d, n]$, where each item is sampled randomly from a Gaussian distribution with mean 0 and variance $1 / d$, and use it to project a dataset from n dimensions down to d:

```
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
```

- خوبی این روش همینطور که دیدیم اینه که نیازی به process کردن روی دیتاست نداریم و فقط از دیتاست وقتی استفاده میکنیم که میخوایم ببریمش به فضا جدید اونم با استفاده از یه ضرب ماتریسی. این کارارو خود scikit-learn هم پیاده سازی کرده.

```
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
```

CONTINUE

- همچنین یه روش دومی هم برای randomPorjection هست که کاری که میکنه اینه مثل حالت قبل میاد d رو تقریب میزنه و میاد یه ماتریس رندوم میسازه. اما یه نکته ای که هست اینه که به جای استفاده از یه توزیع گوسی برای ساخت عدد های ماتریس از ماتریس اسپارس استفاده میکنه که باعث میشه بتونه با حافظه خیلی کمتری ماتریس رو بسازه و سریعتر از روش قبلی باشه. همچنین اگه ورودی ما یه ماتریس اسپارس باشه فرمت اسپارس رو بعد از reduction هم حفظ میکنه هم اینکه کیفیت بهتری داره نسبت به قبلی.

The ratio r of nonzero items in the sparse random matrix is called its *density*. By default, it is equal to $1/\sqrt{n}$. With 20,000 features, this means that only 1 in ~141 cells in the random matrix is nonzero: that's quite sparse! You can set the `density` hyperparameter to another value if you prefer. Each cell in the sparse random matrix has a probability r of being nonzero, and each nonzero value is either $-v$ or $+v$ (both equally likely), where $v = 1/\sqrt{dr}$.

If you want to perform the inverse transform, you first need to compute the pseudo-inverse of the components matrix using SciPy's `pinv()` function, then multiply the reduced data by the transpose of the pseudo-inverse:

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

Computing the pseudo-inverse may take a very long time if the components matrix is large, as the computational complexity of `pinv()` is $O(dn^2)$ if $d < n$, or $O(nd^2)$ otherwise.

LLE

- به عنوان آخرین روش میخوایم locally linear embedding که یه تکنیک nonlinear dimensionality reduction هست رو بگیم. در واقع یک تکنیک manifold learning هستش که وابسته به projection نیست. در کل LLE میاد اول اندازه گیری میکنه هر نمونه به طور خطی وابسته به نزدیک ترین همسایش هست و بعد به این نگاه میکنه که چطوری میتونه یه فضا low-dimensional پیدا کنه براش به طوری که بهترین مپ رخ بده. این روش به طور کلی برای unroll کردن خوبه. همچین دیتاستی داشتیم:

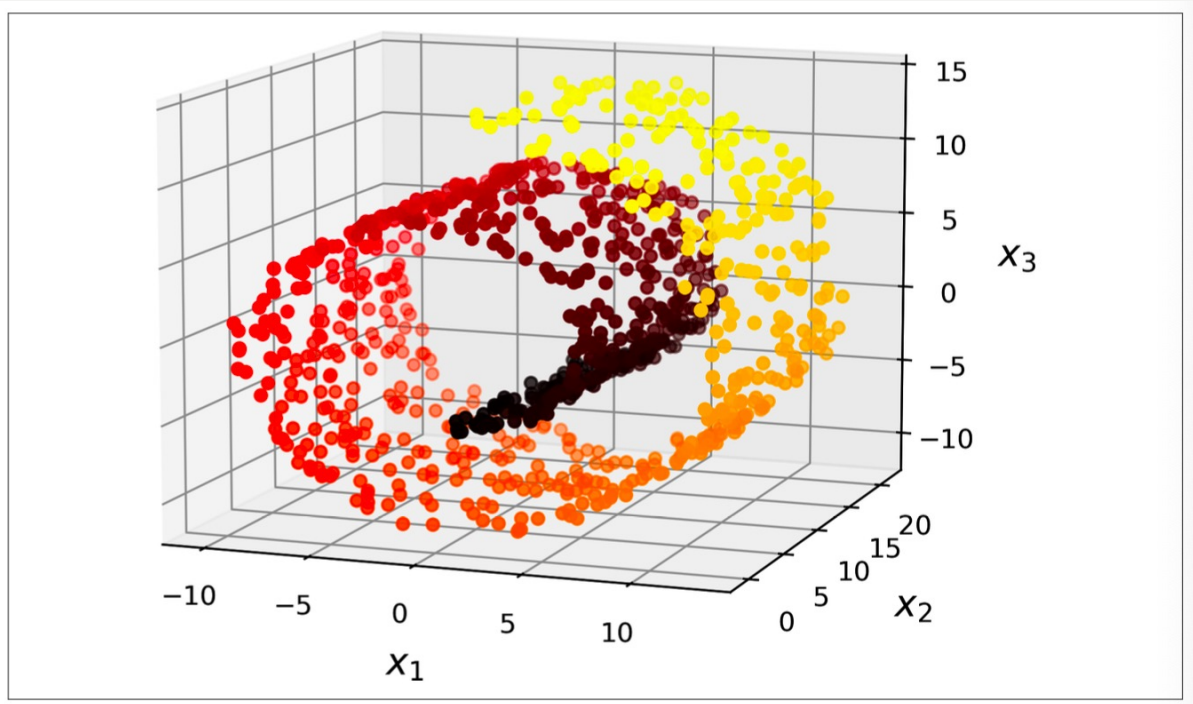

Figure 8-4. Swiss roll dataset

CONTINUE

- کد زیر میاد دیتاست صفحه قبل رو unroll میکنه. یه t هم داریم که تو مثال الان بدردمون نمیخوره صرفا میاد مختصات نقاط رو بهمون میده.

```
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)
```

- خروجی کارمون رو ببینیم:

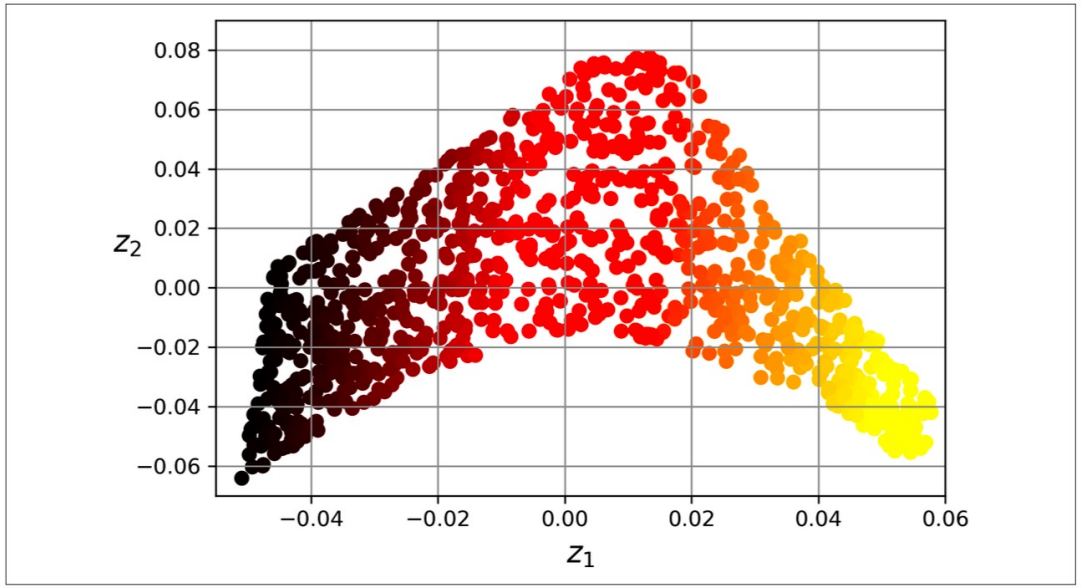

Figure 8-10. Unrolled Swiss roll using LLE

CONTINUE

- اما LLE چطوری ساخته میشه؟ ریاضیاتش تو شکل زیر هست.

Here's how LLE works: for each training instance $\mathbf{x}^{(i)}$, the algorithm identifies its k-nearest neighbors (in the preceding code $k = 10$), then tries to reconstruct $\mathbf{x}^{(i)}$ as a linear function of these neighbors. More specifically, it tries to find the weights $w_{i,j}$ such that the squared distance between $\mathbf{x}^{(i)}$ and $\sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)}$ is as small as possible, assuming $w_{i,j} = 0$ if $\mathbf{x}^{(j)}$ is not one of the k-nearest neighbors of $\mathbf{x}^{(i)}$. Thus the first step of LLE is the constrained optimization problem described in Equation 8-4, where $\mathbf{W}$ is the weight matrix containing all the weights $w_{i,j}$. The second constraint simply normalizes the weights for each training instance $\mathbf{x}^{(i)}$.

Equation 8-4. LLE step 1: linearly modeling local relationships

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

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

CONTINUE

After this step, the weight matrix $\widehat{\mathbf{W}}$ (containing the weights $\widehat{w}_{i,j}$) encodes the local linear relationships between the training instances. The second step is to map the training instances into a d-dimensional space (where $d < n$) while preserving these local relationships as much as possible. If $\mathbf{z}^{(i)}$ is the image of $\mathbf{x}^{(i)}$ in this d-dimensional space, then we want the squared distance between $\mathbf{z}^{(i)}$ and $\sum_{j=1}^{m} \widehat{w}_{i,j}\mathbf{z}^{(j)}$ to be as small as possible. This idea leads to the unconstrained optimization problem described in Equation 8-5. It looks very similar to the first step, but instead of keeping the instances fixed and finding the optimal weights, we are doing the reverse: keeping the weights fixed and finding the optimal position of the instances' images in the low-dimensional space. Note that $\mathbf{Z}$ is the matrix containing all $\mathbf{z}^{(i)}$.

Equation 8-5. LLE step 2: reducing dimensionality while preserving relationships

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

همونطور که دیدیم LLE تفاوت هایی با projection داره اما خب توانایی های خوبی هم داره مخصوصا تو دیتاهایی که غیرخطی هستن.