5. Dimensionality Reduction

COMP3314
Machine Learning

Motivation

- Many ML problems have thousands or even millions of features
- As a result we have an intractable problem
  - Training is slow
  - Finding a solution is difficult (Curse of Dimensionality)
  - Data visualization is impossible
- Solution
  - Dimensionality Reduction using feature extraction
  - Often possible without losing much relevant information
    - E.g., Merge neighboring pixels of the MNIST dataset

The Curse of Dimensionality

High Dimensional Weirdness

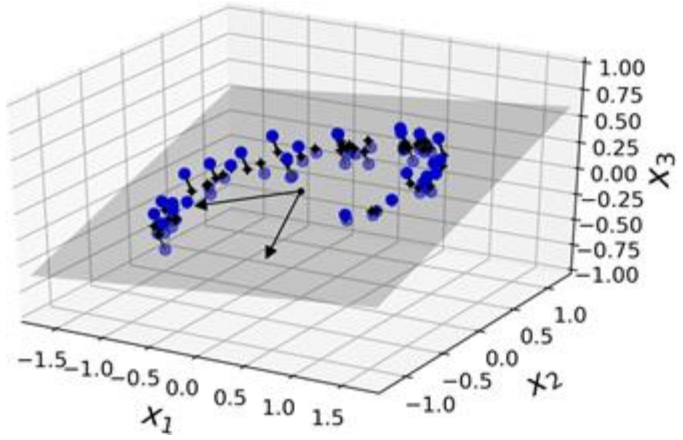
- 2D
  - Pick a random point in a unit square will have <0.4% chance of being located <0.001 from a border
- 10,000D
  - Pick a random point in a unit hypercube will have >99.99999% chance of being located <0.001 from a border
- I.e., the high-dimensional unit hypercube can be said to consist almost entirely of borders with almost no middle

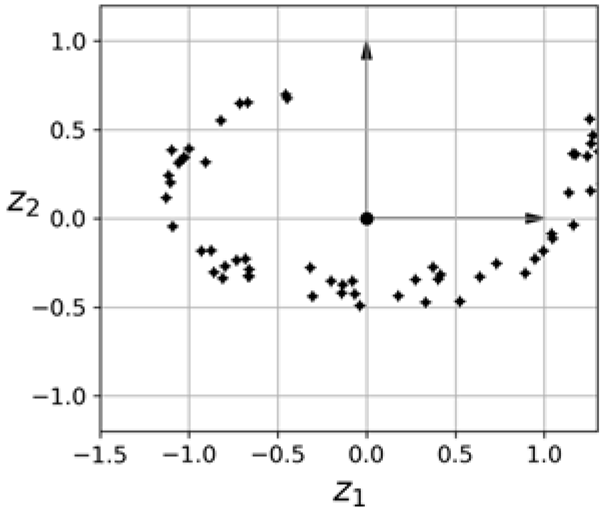
High Dimensional Weirdness

- 2D
  - Pick two random points in a unit square
  - Distance between them will be 0.52 on average
- 1,000,000D
  - Pick two random points in a unit hypercube
  - Distance between them will be 408.25 on average
- How can two points be so far apart when they both lie within the same unit hypercube?
- As a result, new test samples will likely be far away from training samples in high dimensional space
  - Overfitting risk is much higher in high-dimensional space

Idea: Projection

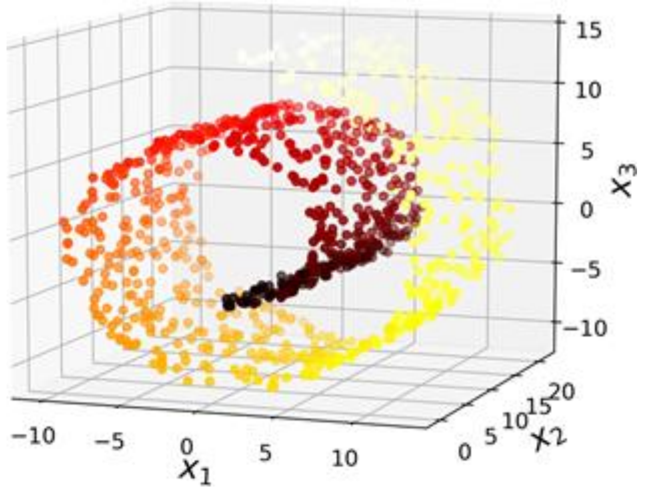
- In most problems, training instances are not spread out uniformly across all dimensions
- Many features are almost constant, while others are highly correlated
- As a result, all training instances lie within (or close to) a much lower-dimensional subspace of the high-dimensional space

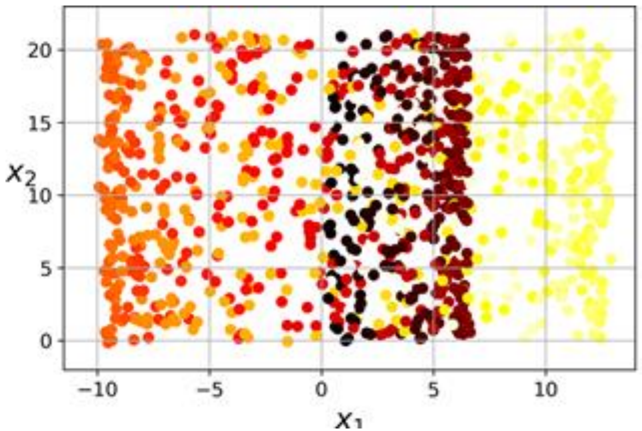




When Projection Fails

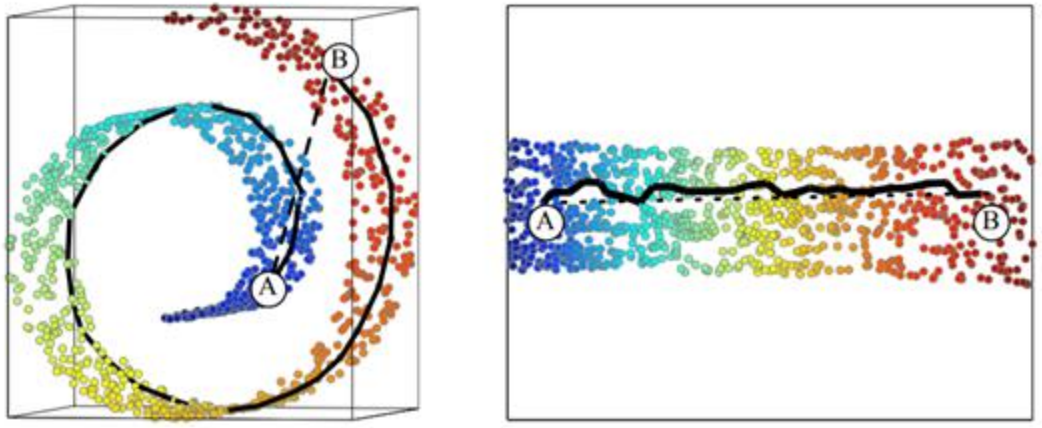
- Projection is not always the best approach
- Consider the following toy dataset to illustrate this problem
    - The Swiss roll
- Simply projecting onto a plane (e.g., dropping x3) would squash different layers





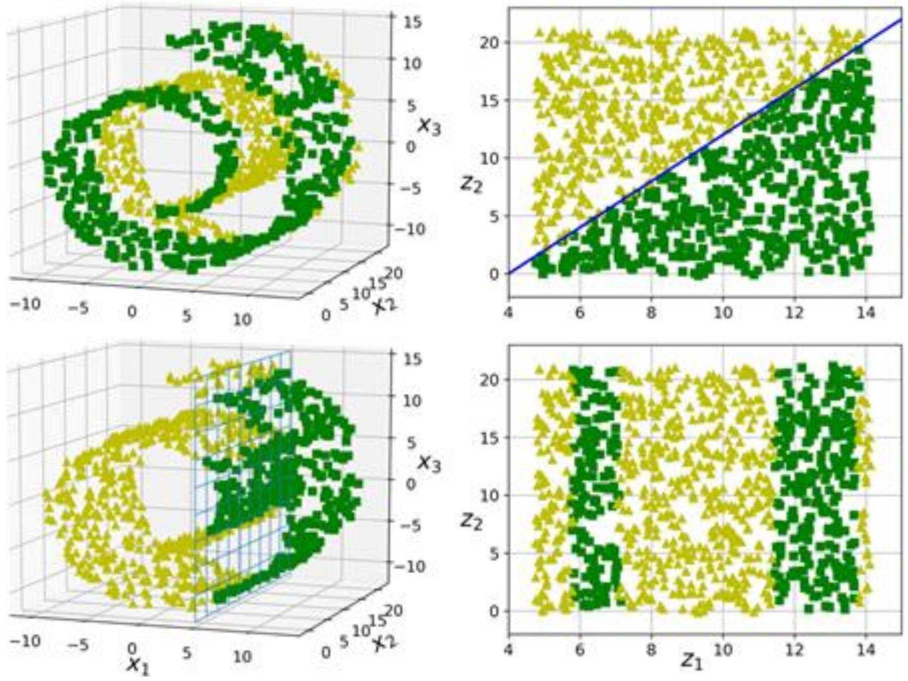
Solution: Manifold Learning

- The Swiss roll is an example of 2D manifold
  - A 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space
- It is possible to learn the manifold on which the training instances lie and then to unroll the swiss roll



Manifold Learning

- Note
  - The decision boundary may not always be simpler in lower dimensions

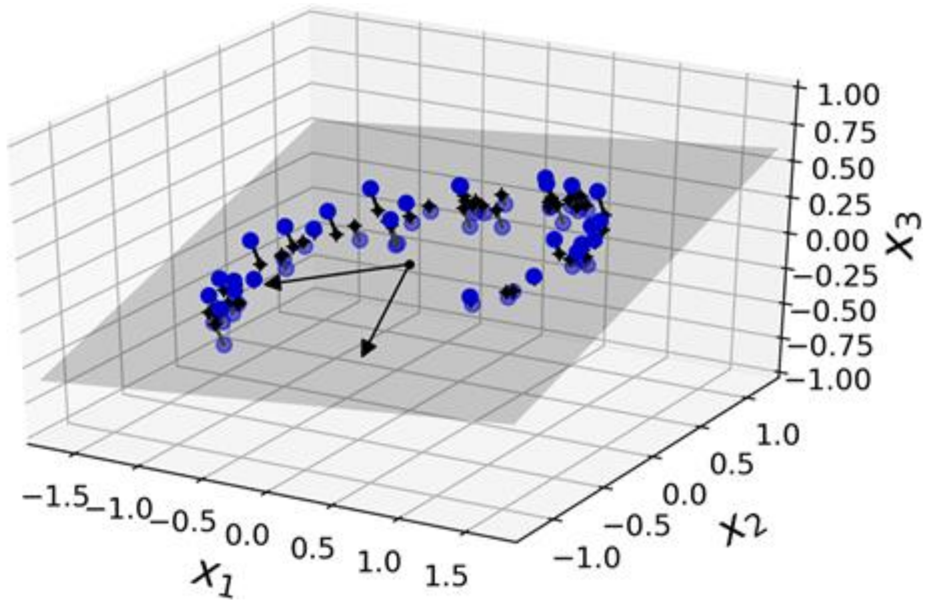


Outline

- PCA
  - Principal Component Analysis
  - Projects data points onto (few) principal components
- LLE
  - Locally Linear Embedding
  - Powerful nonlinear dimensionality reduction technique
  - Manifold Learning technique that does not rely on projections

PCA - Principal Component Analysis

- By far the most popular dimensionality reduction algorithm
- Identifies a hyperplane and then projects data onto it



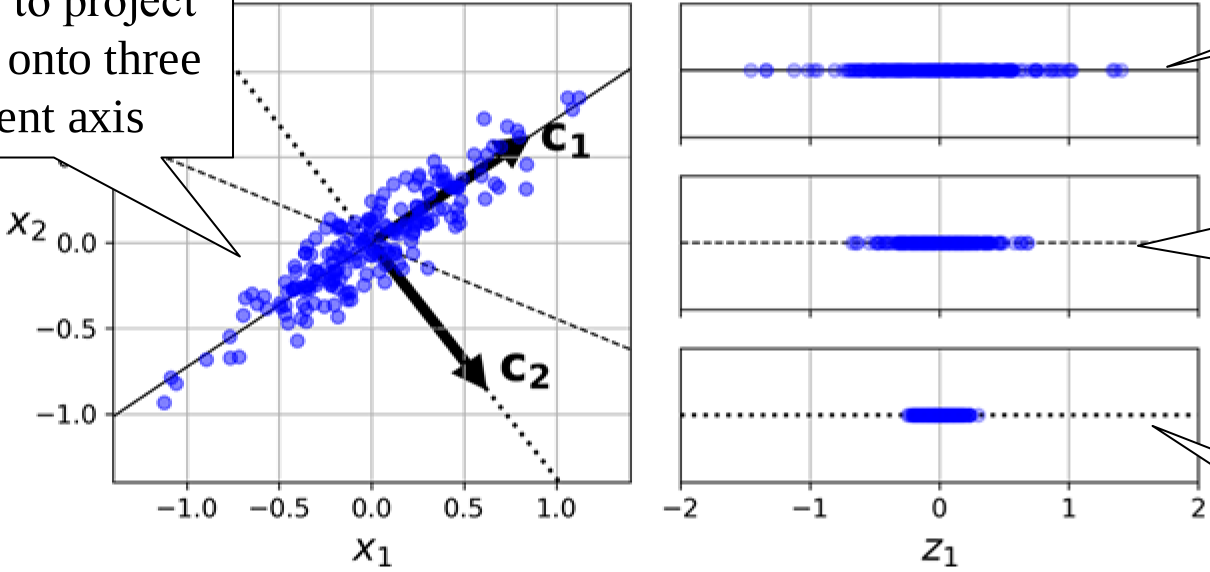
How to choose the hyperplane?

🤔

Preserving the Variance

- Select axis that preserves the maximum amount of variance
  - I.e., loses less information than other projections
- Example: 1D Hyperplanes

Let's try to project this data onto three different axis



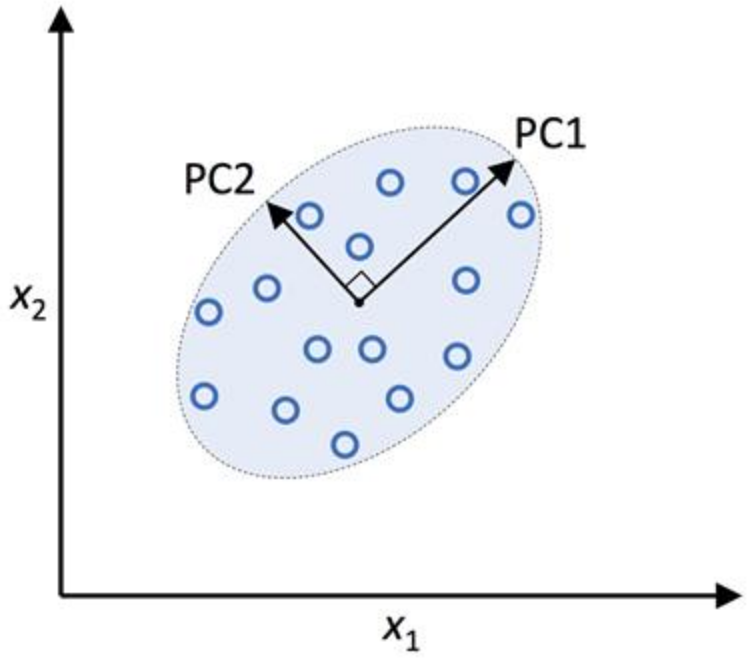
Preserves max variance

Preserves intermediate amount of variance

Preserves very little variance

Principal Components (PC)

- The first PC is the axis that accounts for the largest amount of variance
  - E.g., PC1 in the figure
- The second PC is orthogonal to the first one and accounts for the largest amount of remaining variance
  - E.g., PC2 in the figure
  - In this 2D example there is no choice
- If it were in a higher-dimensional dataset the third PC would be orthogonal to both previous axes, and a fourth, a fifth, and so on—as many axes as the number of dimensions in the dataset



How to find PCs?

- There is a standard matrix factorization technique called Singular Value Decomposition (SVD)
- It decomposes the training set matrix X into the matrix multiplication of three matrices $X = U \Sigma V^{\intercal}$, where V contains the unit vectors that define all the principal components that we are looking for
- Note that PCs are highly sensitive to data scaling
- We need to standardize the features prior to PCA if the features were measured on different scales

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

PCA - Principal Component Analysis

- An unsupervised linear transformation technique
  - Finds PCs
    - Using e.g., SVD
  - Projects data onto a subspace with fewer (or equal) dimensions using some (or all) of the found PCs
    - Multiply original data with a transformation matrix that consists of PCs, some (or all)

Projecting Down to k Dimensions

- Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to k dimensions by projecting it onto the hyperplane defined by the first k principal components
- To project the training set onto the hyperplane and obtain a reduced dataset of dimensionality k, compute the matrix multiplication of the training set vector (or matrix) **x** (or **X**) by the matrix **W**, defined as the matrix containing the first k columns of **V**
- **W** is a *d* × *k* transformation matrix
    - Maps a *d*-dimensional vector **x** to a *k*-dimensional vector **z**

$$\boldsymbol{x} = [x_1, x_2, \dots, x_d], \quad \boldsymbol{x} \in \mathbb{R}^d$$

$$\downarrow \boldsymbol{xW}, \quad \boldsymbol{W} \in \mathbb{R}^{d \times k}$$

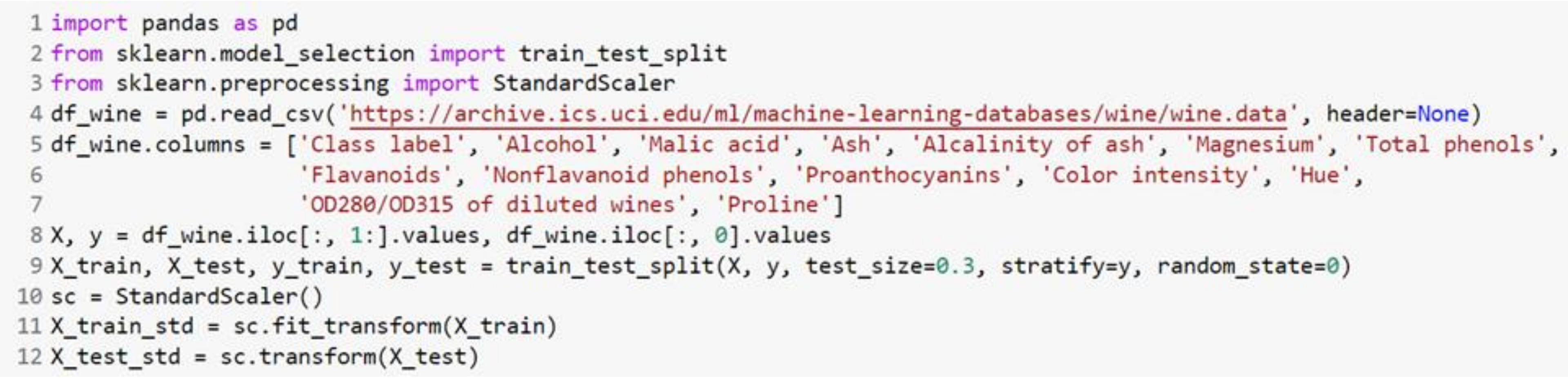
$$\mathbf{z} = [z_1, z_2, \dots, z_k], \quad \mathbf{z} \in \mathbb{R}^k$$

Code - PCA.ipynb

- Available here on CoLab

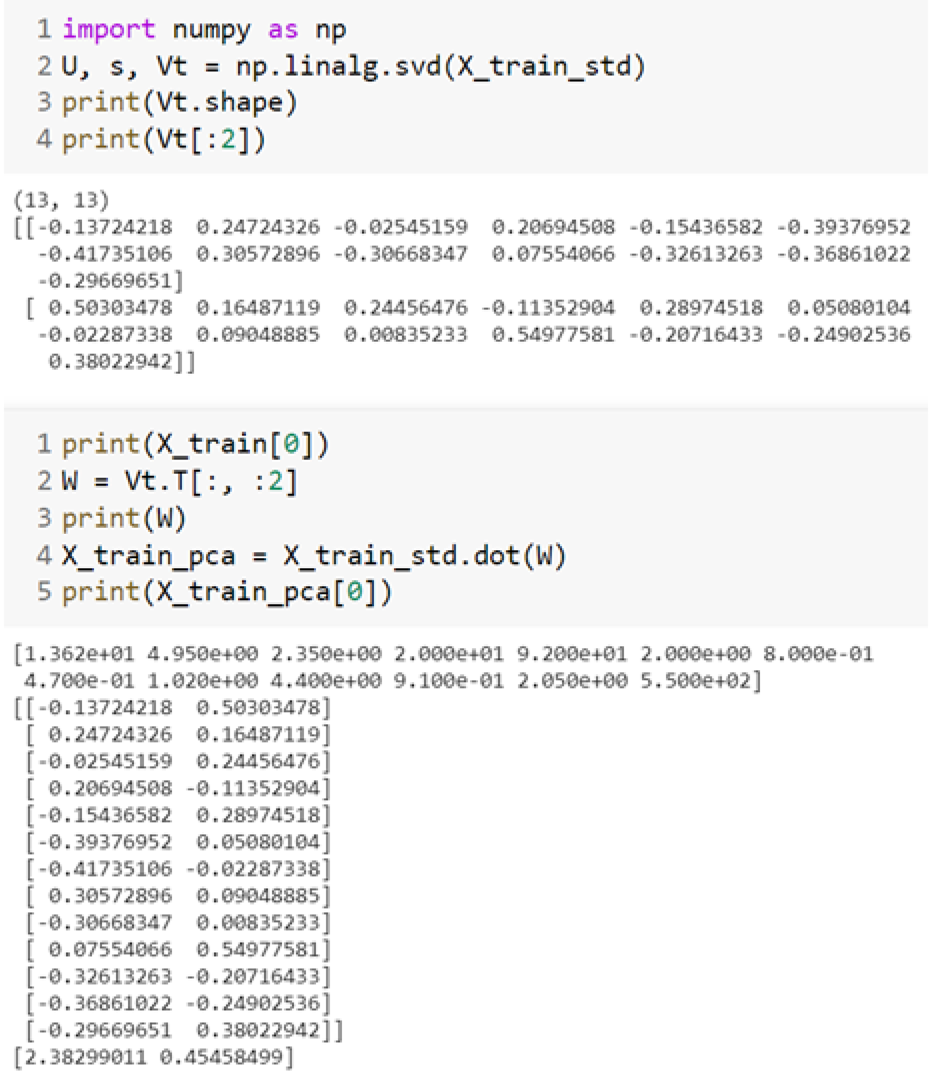
Load and Standardize Data

- Let’s apply PCA on the wine dataset
  - Load the wine dataset and split it into separate train and test sets
  - Standardize the (d=13)-dimensional dataset



```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data', header=None)
df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins', 'Color intensity', 'Hue',
                   'OD280/OD315 of diluted wines', 'Proline']
X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=0)
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
```

Projecting Down



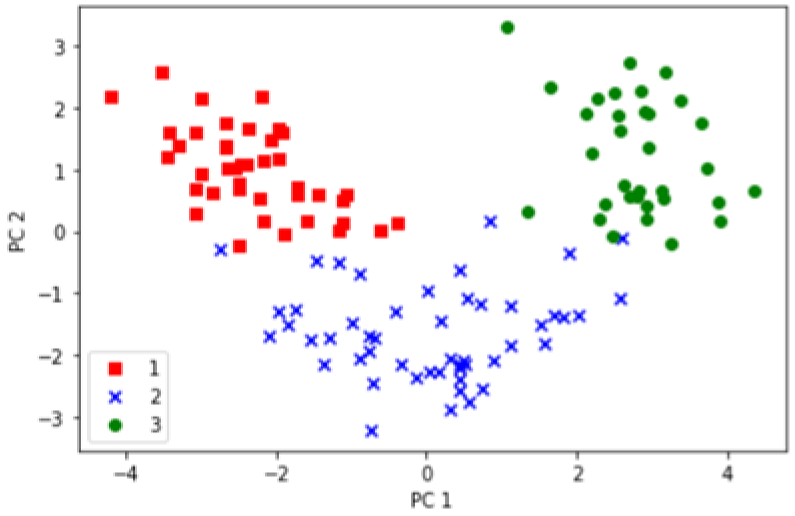
```
import numpy as np
U, s, Vt = np.linalg.svd(X_train_std)
print(Vt.shape)
print(Vt[:2])
```

```
(13, 13)
[[-0.13724218  0.24724326 -0.02545159  0.20694508 -0.15436582 -0.39376952
  -0.41735106  0.30572896 -0.30668347  0.07554066 -0.32613263 -0.36861022
  -0.29669651]
 [ 0.50303478  0.16487119  0.24456476 -0.11352904  0.28974518  0.05080104
  -0.02287338  0.09048885  0.00835233  0.54977581 -0.20716433 -0.24902536
   0.38022942]]
```

```
print(X_train[0])
W = Vt.T[:, :2]
print(W)
X_train_pca = X_train_std.dot(W)
print(X_train_pca[0])
```

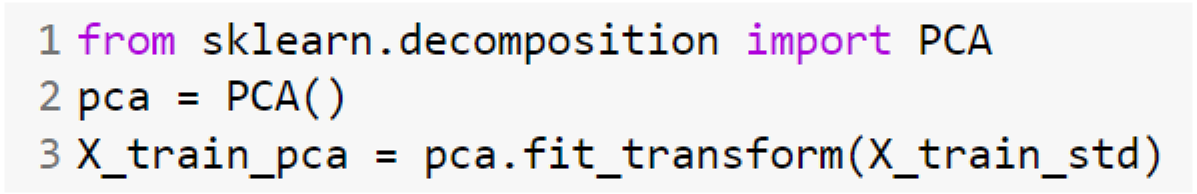
```
[1.362e+01 4.950e+00 2.350e+00 2.000e+01 9.200e+01 2.000e+00 8.000e-01
 4.700e-01 1.020e+00 4.400e+00 9.100e-01 2.050e+00 5.500e+02]
[[-0.13724218  0.50303478]
 [ 0.24724326  0.16487119]
 [-0.02545159  0.24456476]
 [ 0.20694508 -0.11352904]
 [-0.15436582  0.28974518]
 [-0.39376952  0.05080104]
 [-0.41735106 -0.02287338]
 [ 0.30572896  0.09048885]
 [-0.30668347  0.00835233]
 [ 0.07554066  0.54977581]
 [-0.32613263 -0.20716433]
 [-0.36861022 -0.24902536]
 [-0.29669651  0.38022942]]
[2.38299011 0.45458499]
```

```python
import matplotlib.pyplot as plt
colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']
for l, c, m in zip(np.unique(y_train), colors, markers):
    plt.scatter(X_train_pca[y_train == l, 0], X_train_pca[y_train == l, 1], c=c, label=l, marker=m)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```



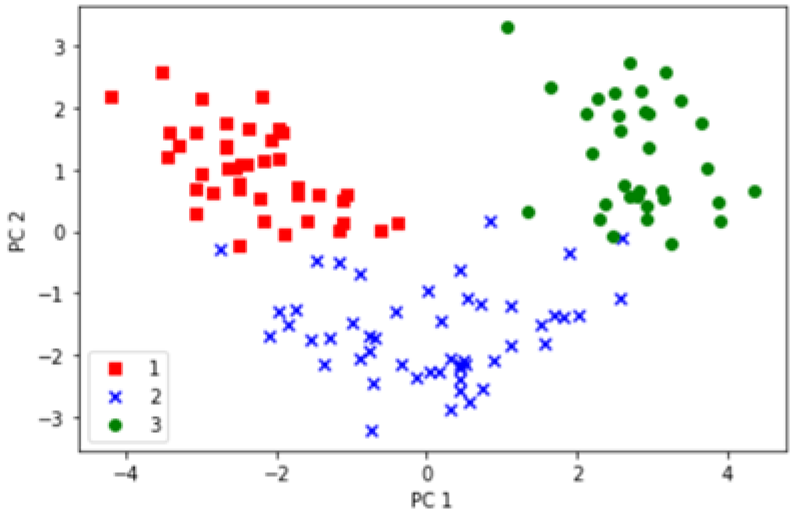
Using Scikit-Learn's PCS

- Scikit-Learn's PCA class uses SVD decomposition to implement PCA
  - Just like we did manually
- The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions



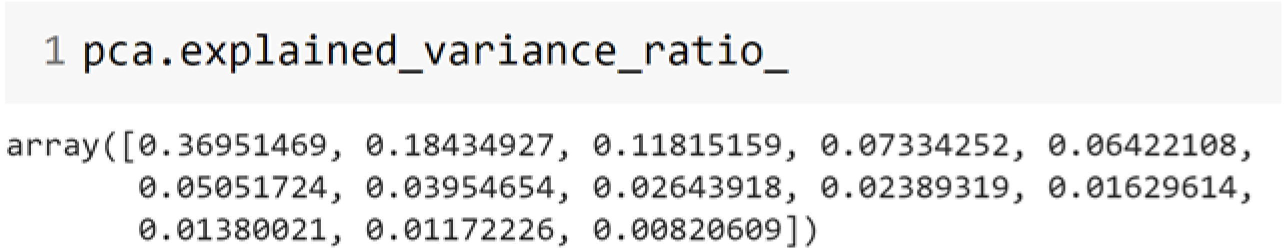
```python
from sklearn.decomposition import PCA
pca = PCA()
X_train_pca = pca.fit_transform(X_train_std)
```

```python
import matplotlib.pyplot as plt
colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']
for l, c, m in zip(np.unique(y_train), colors, markers):
    plt.scatter(X_train_pca[y_train == l, 0], X_train_pca[y_train == l, 1], c=c, label=l, marker=m)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```



Explained Variance Ratio

- Another useful piece of information is the explained variance ratio of each principal component
  - Available via the explained_variance_ratio_ variable
- The ratio indicates the proportion of the dataset's variance that lies along each principal component



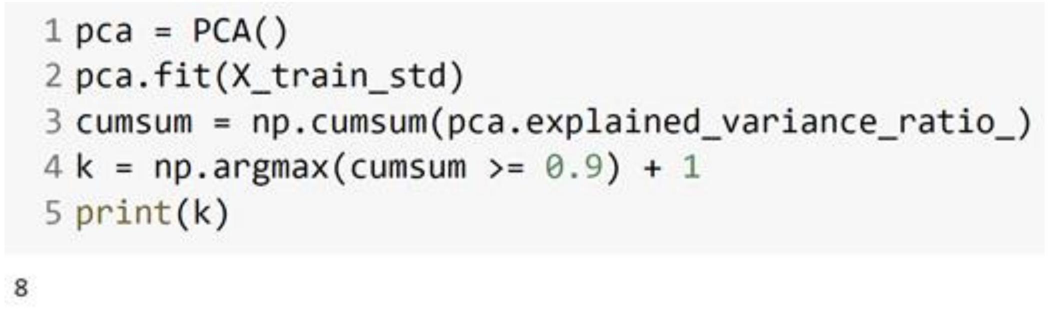
```
pca.explained_variance_ratio_
```

```
array([0.36951469, 0.18434927, 0.11815159, 0.07334252, 0.06422108,
       0.05051724, 0.03954654, 0.02643918, 0.02389319, 0.01629614,
       0.01380021, 0.01172226, 0.00820609])
```

- This output tells us that 36.9% of the dataset's variance lies along the first PC, and 18.4% lies along the second PC, etc

Choosing the Right Number of Dimensions

- Choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%)
  - Unless, of course, you are reducing dimensionality for data visualization—in that case you will want to reduce the dimensionality down to 2 or 3
- The following code performs PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 90% of the training set's variance

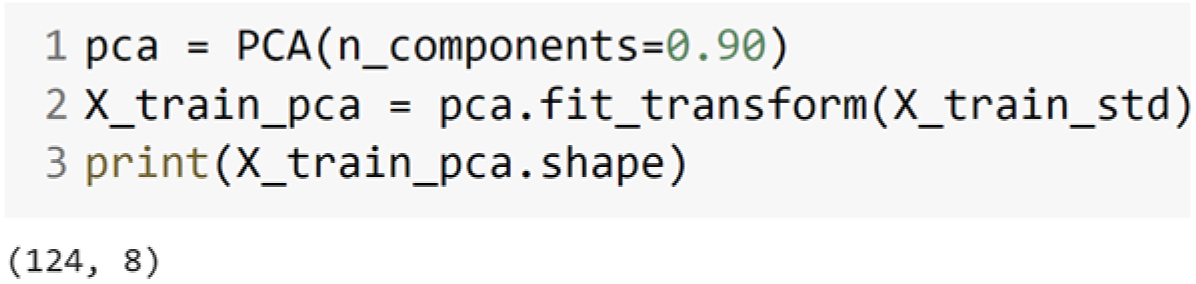


```
pca = PCA()
pca.fit(X_train_std)
cumsum = np.cumsum(pca.explained_variance_ratio_)
k = np.argmax(cumsum >= 0.9) + 1
print(k)
```

```
8
```

Choosing the Right Number of Dimensions

- You could then set n_components=k and run PCA again
  - But there is a much better option: instead of specifying the number of principal components you want to preserve, you can set n_components to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve



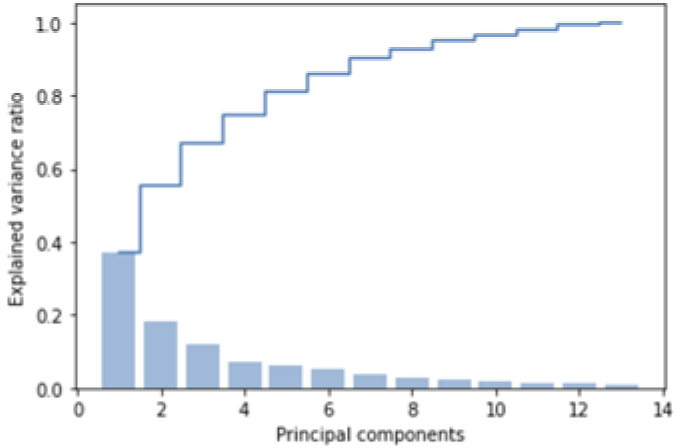
```
pca = PCA(n_components=0.90)
X_train_pca = pca.fit_transform(X_train_std)
print(X_train_pca.shape)
```

```
(124, 8)
```

Choosing the Right Number of Dimensions

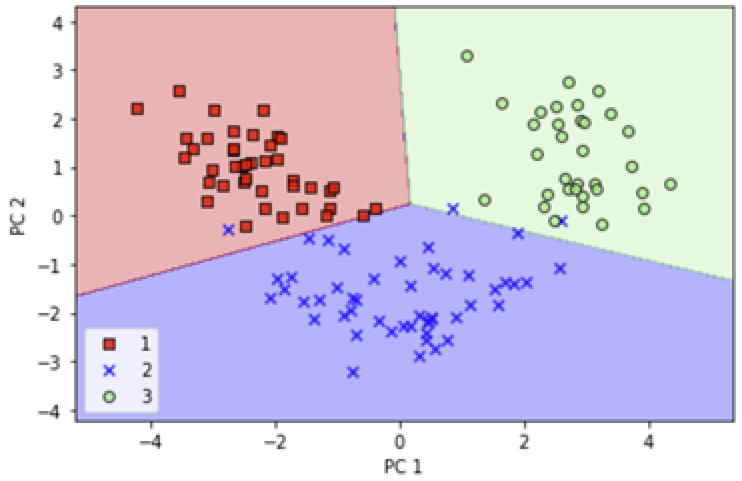
- Yet another option is to plot the explained variance as a function of the number of dimensions

```
plt.bar(range(1, 14), pca.explained_variance_ratio_, alpha=0.5, align='center')
plt.step(range(1, 14), np.cumsum(pca.explained_variance_ratio_), where='mid')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal components')
plt.show()
```

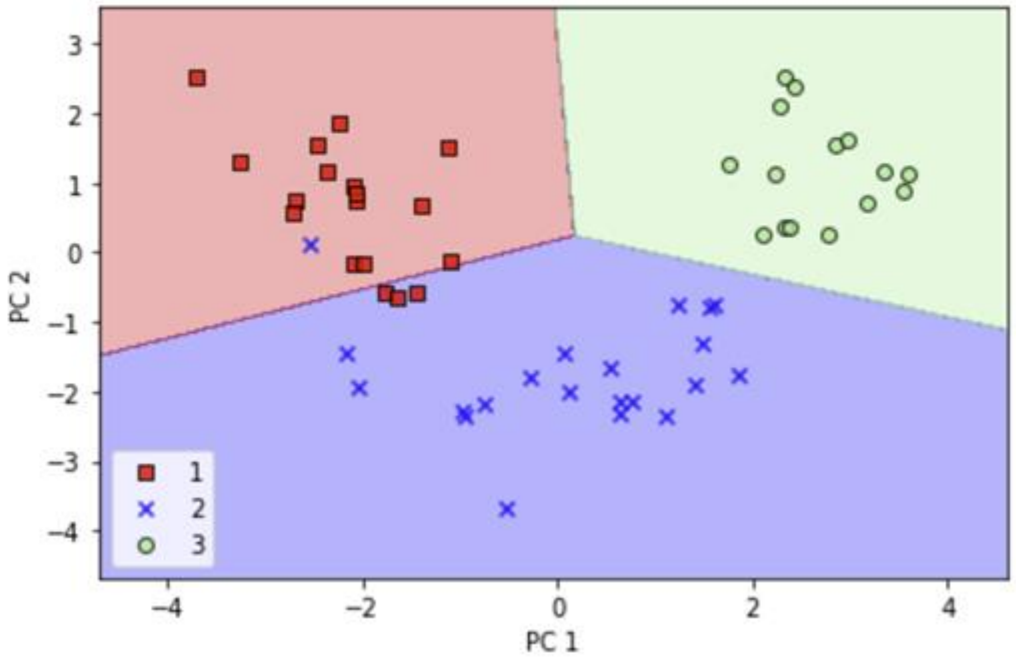


```python
from sklearn.linear_model import LogisticRegression
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)
lr = LogisticRegression(solver='liblinear', multi_class='ovr')
lr = lr.fit(X_train_pca, y_train)
```

```python
plot_decision_regions(X_train_pca, y_train, classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```

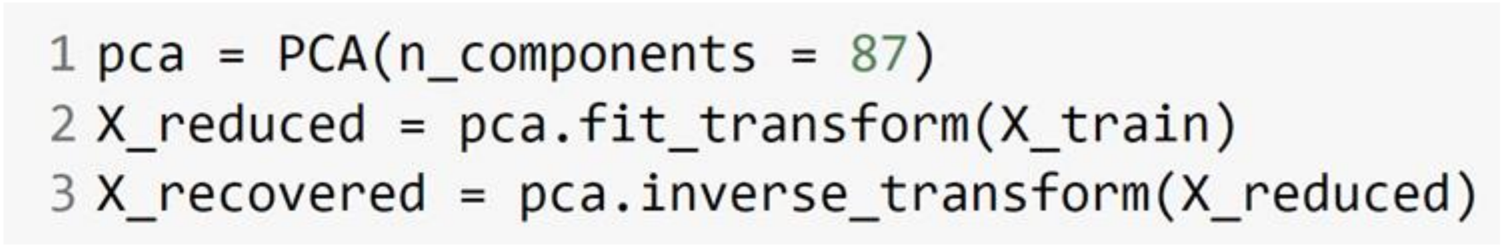


```python
plot_decision_regions(X_test_pca, y_test, classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```



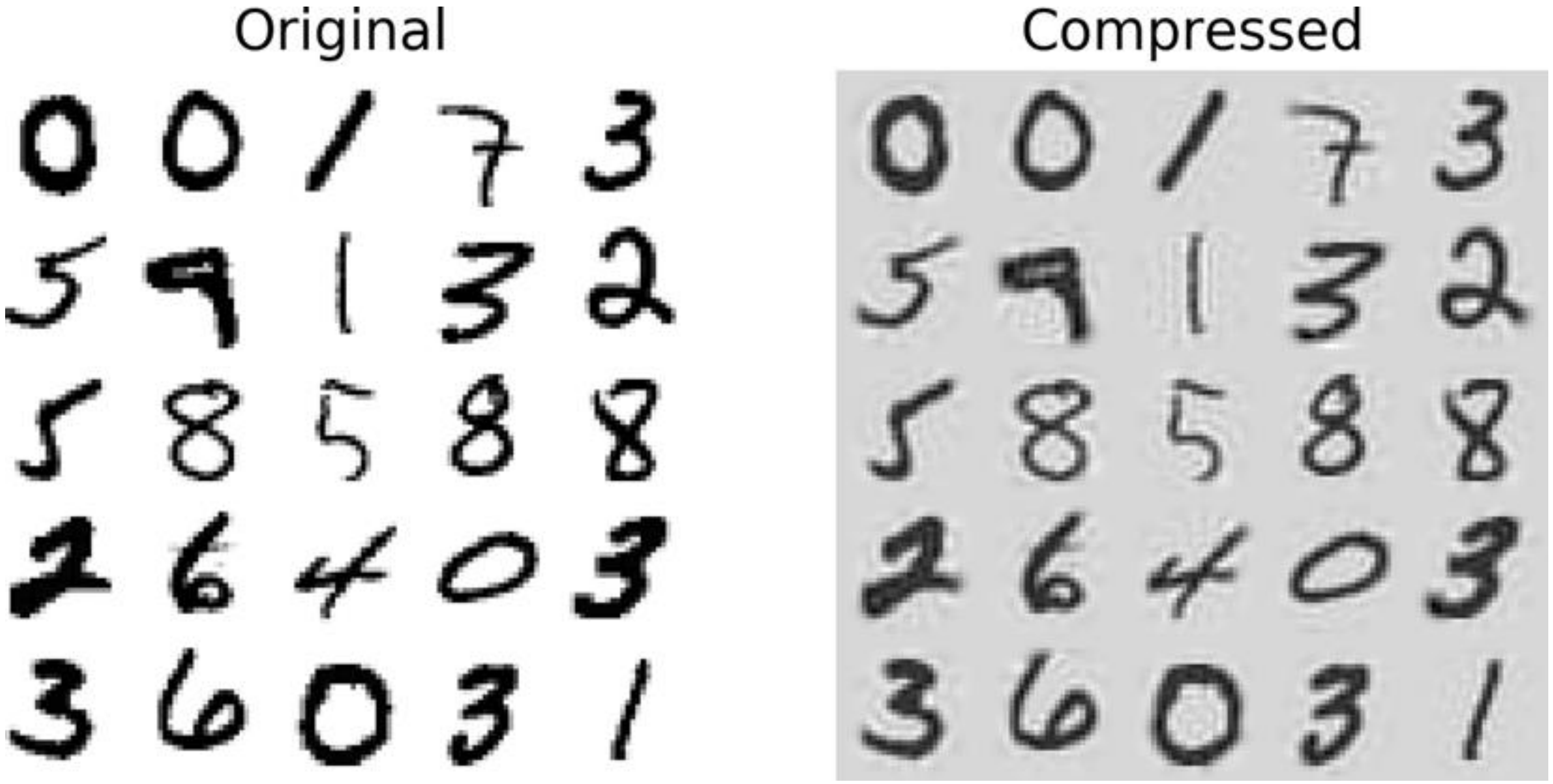
PCA for Compression

- Let’s apply PCA to the MNIST dataset while preserving 90% of its variance
  - 87 features instead of the original 784 features
  - This size reduction can speed up a classification algorithm (such as an SVM classifier) tremendously
- It is also possible to decompress the reduced dataset back to 784 dimensions
  - This won’t give you back the original data, since the projection lost a bit of information (within the 10% variance that was dropped)
  - The following code compresses the MNIST dataset down to 87 dimensions, then uses the inverse_transform() method to decompress it back to 784 dimensions



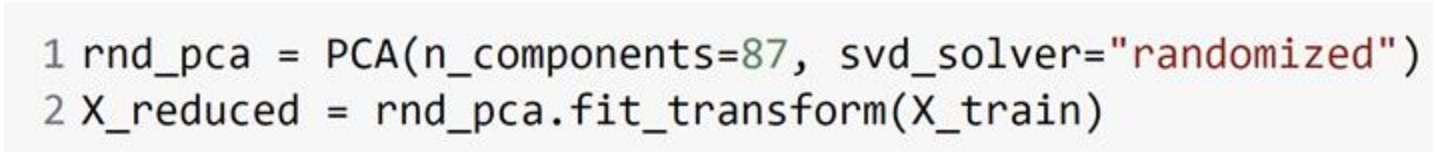
```
pca = PCA(n_components = 87)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

PCA for Compression



Randomized PCA

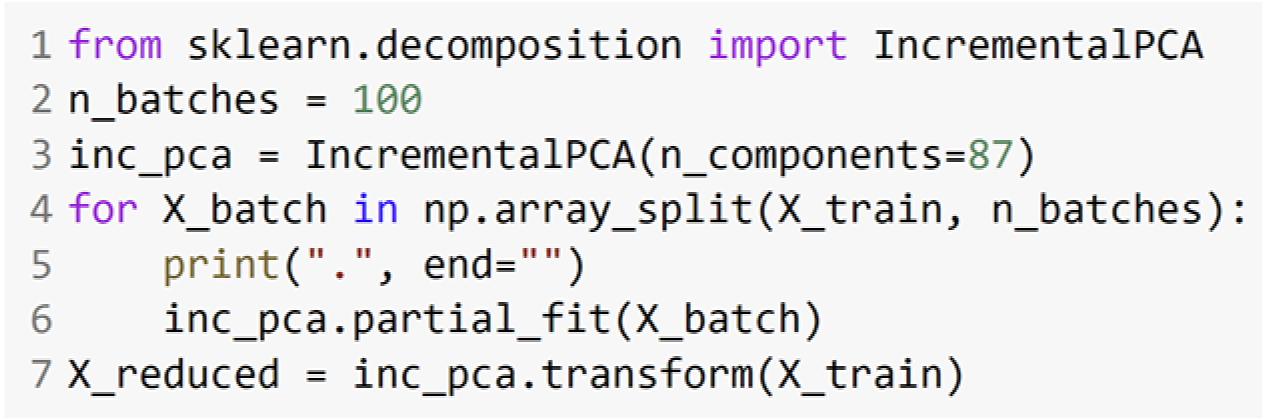
- If you set the svd_solver hyperparameter to "randomized", Scikit-Learn uses a stochastic algorithm called Randomized PCA that quickly finds an approximation of the first d principal components
  - It is dramatically faster than full SVD when k is much smaller than d
- By default, svd_solver is actually set to "auto"
  - Scikit-Learn automatically uses the randomized PCA algorithm if d is greater than 500 and k is less than 80% d, or else it uses the full SVD approach
  - If you want to force Scikit-Learn to use full SVD, you can set the svd_solver hyperparameter to "full"



```python
rnd_pca = PCA(n_components=87, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

Incremental PCA

- The previous PCA implementations require the whole training set to fit in memory
- Incremental PCA (IPCA) allows you to feed an IPCA algorithm one mini-batch at a time
  - Useful for large training sets and online training (i.e., on the fly, as new data arrive)
- The following code splits the MNIST dataset into 100 mini-batches (using NumPy's array_split() function) and feeds them to Scikit-Learn's IncrementalPCA class
  - Note that you must call the partial_fit() method with each mini-batch, rather than the fit() method with the whole training set



```python
from sklearn.decomposition import IncrementalPCA
n_batches = 100
inc_pca = IncrementalPCA(n_components=87)
for X_batch in np.array_split(X_train, n_batches):
    print(".", end="")
    inc_pca.partial_fit(X_batch)
X_reduced = inc_pca.transform(X_train)
```

Outline

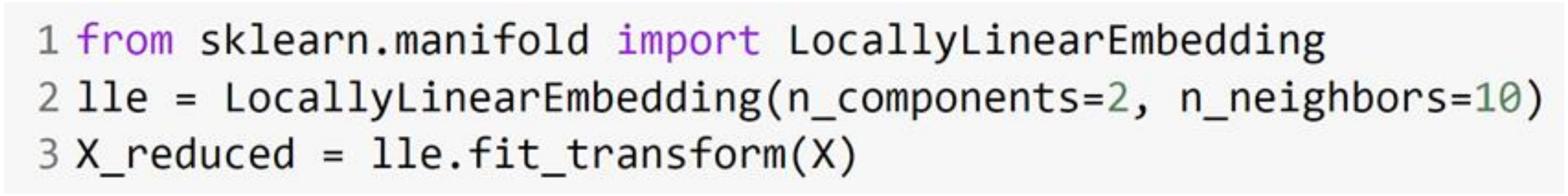
- PCA
  - Principal Component Analysis
  - Projects data points onto (few) principal components
- LLE
  - Locally Linear Embedding
  - Powerful nonlinear dimensionality reduction technique
  - Manifold Learning technique that does not rely on projections

Code - LLE.ipynb

- Available here on CoLab

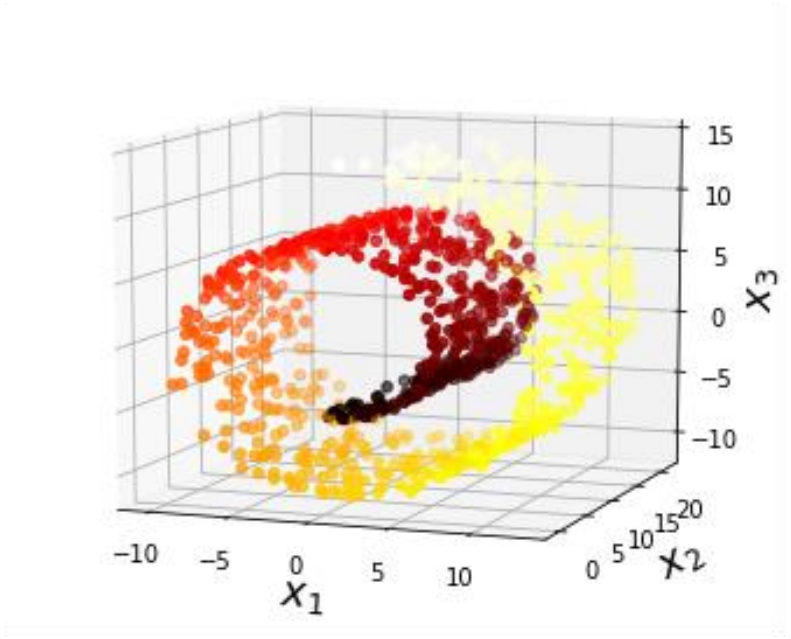
LLE

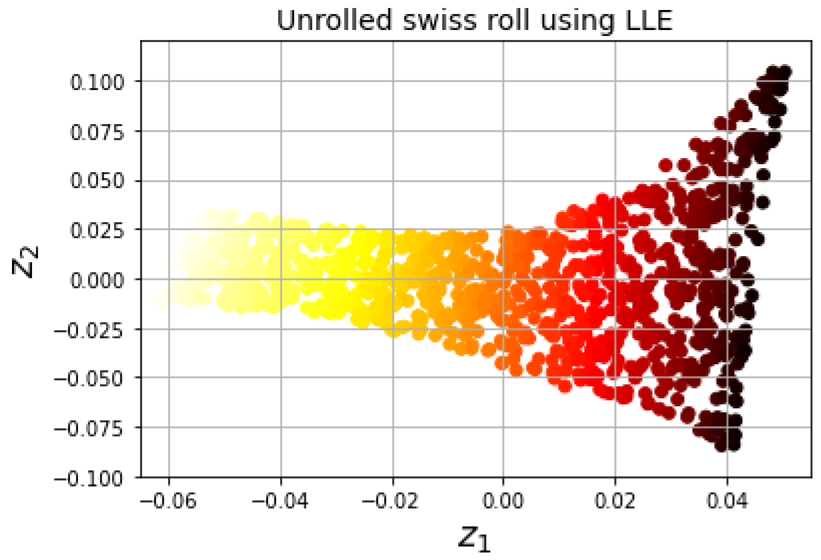
- How it works
  - Measures how each training instance linearly relates to its closest neighbors
  - Then looks for a low-dimensional representation of the training set where these local relationships are best preserved
- This approach makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise



```python
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

Example: Unrolling the Swiss roll





LLE - Details

- For each training sample $\mathbf{x}^{(i)}$, the algorithm identifies its n_neighbors closest neighbors
  - E.g., n_neighbors = 10
- Then it tries to reconstruct $\mathbf{x}^{(i)}$ as a linear function of these neighbors
- More specifically, it finds 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 closest neighbors of $\mathbf{x}^{(i)}$

LLE - Details

- Thus the first step of LLE is the constrained optimization problem below, where **W** is the weight matrix containing all the weights $w_{i,j}$
- The second constraint simply normalizes the weights for each training instance $\mathbf{x}^{(i)}$

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

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

LLE - Details

- After this step, the weight matrix **W**ˆ (containing the weights wˆ$_{i,j}$) encodes the local linear relationships between the training instances
- The second step is to map the training instances into a k-dimensional space (where k < d) while preserving these local relationships as much as possible
- If $\mathbf{z}^{(i)}$ is the image of $\mathbf{x}^{(i)}$ in this k-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

LLE - Details

- This idea leads to the following unconstrained optimization problem
- 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 **Z** is the matrix containing all $\mathbf{z}^{(i)}$

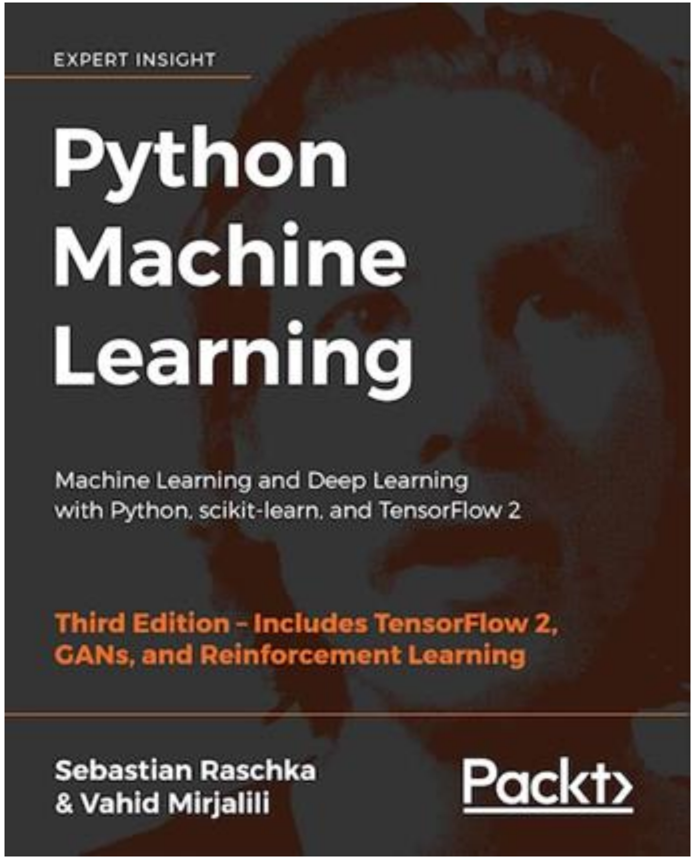
$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{z}^{(i)} - \sum_{j=1}^{m} \widehat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

Other Dimensionality Reduction Techniques

- There are many other dimensionality reduction techniques, several of which are available in Scikit-Learn
- Here are some of the most popular ones
    - Random Projections
    - Multidimensional Scaling (MDS)
    - Isomap
    - t-Distributed Stochastic Neighbor Embedding (t-SNE)
    - Linear Discriminant Analysis (LDA)

References

- Most materials in this chapter are based on
  - Book
  - Code


EXPERT INSIGHT

Python Machine Learning

Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2

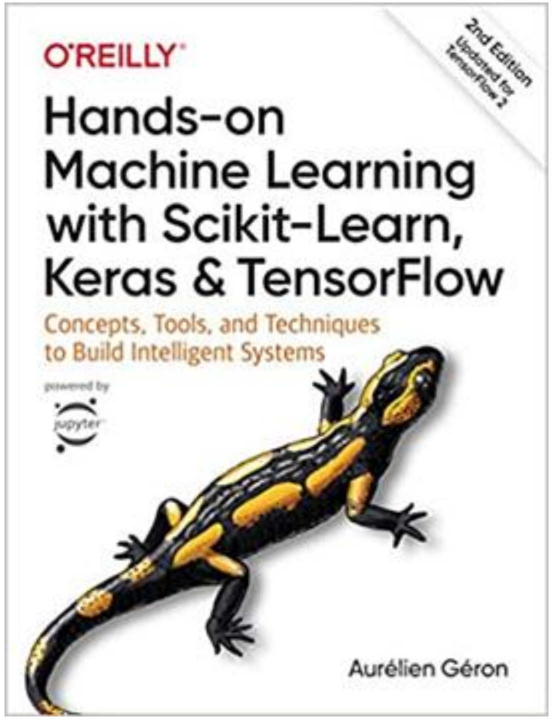
Third Edition - Includes TensorFlow 2, GANs, and Reinforcement Learning

Sebastian Raschka & Vahid Mirjalili

Packt>

References

- Some materials in this chapter are based on
  - Book
  - Code


O'REILLY
2nd Edition Updated for TensorFlow 2
Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow
Concepts, Tools, and Techniques to Build Intelligent Systems
powered by jupyter
Aurélien Géron

Exercise 1

- What are the main motivations for reducing a dataset’s dimensionality?
  - What are the main drawbacks?
- What is the curse of dimensionality?
- Once a dataset’s dimensionality has been reduced, is it possible to reverse the operation?
  - If so, how? If not, why?
- Can PCA be used to reduce the dimensionality of a highly nonlinear dataset?
- Suppose you perform PCA on a 1,000-dimensional dataset, setting the explained variance ratio to 95%
  - How many dimensions will the resulting dataset have?

Exercise 2

- In what cases would you use vanilla PCA, Incremental PCA, Randomized PCA?
- How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?
- Does it make any sense to chain two different dimensionality reduction algorithms?

Exercise 3

- Load the MNIST dataset and split it into a training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing)
- Train a Random Forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set
- Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%
- Train a new Random Forest classifier on the reduced dataset and see how long it takes
- Was training much faster?
- Next, evaluate the classifier on the test set
- How does it compare to the previous classifier?