CAPAIAN PEMBELAJARAN PRAKTIKUM

1. Menjelaskan konsep-konsep dari pembelajaran mesin untuk kemudian bisa diterapkan pada berbagai permasalahan – (C2);
2. Mengimplementasikan algoritma dan/atau metode di pembelajaran mesin sehingga bisa mendapatkan pemecahan masalah dan model yang sesuai – (C3);
3. Menghasilkan suatu model terbaik dari permasalahan dan terus melakukan perbaikan jika terdapat perkembangan di bidang rekayasa cerdas – (C6);

WEEK 03 Principal Component Analysis

Principal Component Analysis

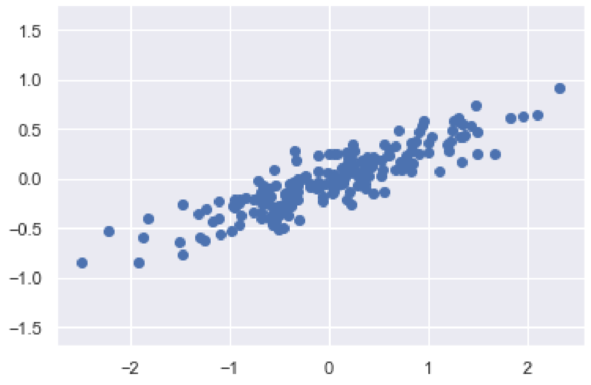
Principal Component Analysis (PCA) is a fast and flexible unsupervised learning method for dimensionality reduction in data. Its behavior is easiest to visualize by looking at a two-dimensional dataset. Consider the following 200 points.

Lets begin with the standard import :

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

Generate the 200 points :

```
rng = np.random.RandomState(1)
X = np.dot(rng.rand(2, 2), rng.randn(2, 200)).T
plt.scatter(X[:, 0], X[:, 1])
plt.axis('equal');
```



By eye, it is clear that there is a nearly linear relationship between the x and y variable.

In PCA, one quantifies this relationship by finding a list of the principal axes in the data and using those axes to describe the data set. Using Scikit-Learn's PCA estimator, we can compute this as follows:

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

```
PCA(copy=True, iterated_power='auto', n_components=2, random
_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
```

The “fit” learns some quantifiers from the data, most importantly the “components” and “explained variance”.

```
print(pca.components_)
```

```
[[-0.94446029 -0.32862557]
 [-0.32862557  0.94446029]]
```

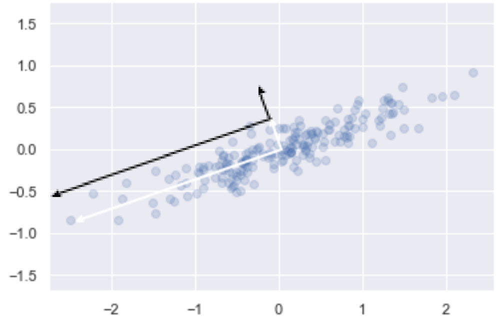
```
print(pca.explained_variance_)
```

```
[0.7625315 0.0184779]
```

To see what these numbers mean, let’s visualize them as vectors over the input data using the “component” to define the direction of the vector and the “explained variance” to define the squared-length of the vector.

```
def draw_vector(v0, v1, ax=None):
    ax = ax or plt.gca()
    arrowprops=dict(arrowstyle='->',
                    linewidth=2,
                    shrinkA=0, shrinkB=0)
    ax.annotate('', v1, v0, arrowprops=arrowprops)

# plot data
plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
for length, vector in zip(pca.explained_variance_, pca.components_):
    v = vector * 3 * np.sqrt(length)
    draw_vector(pca.mean_, pca.mean_ + v)
plt.axis('equal');
```



These vectors represent the principal axes of the data and the length shown in the picture above. It is an indication of how “important” the axis is in describing the distribution of the data, it is a measure of the variance of the data when projected onto that axis. The projection of each data point onto the principal axes is the “principal component” of the data.

PCA as dimensionality reduction

Using PCA for dimensionality reduction involves zeroing out more or more of the smallest principal component, resulting in a lower dimensional projection of the data that preserves the maximal data variance.

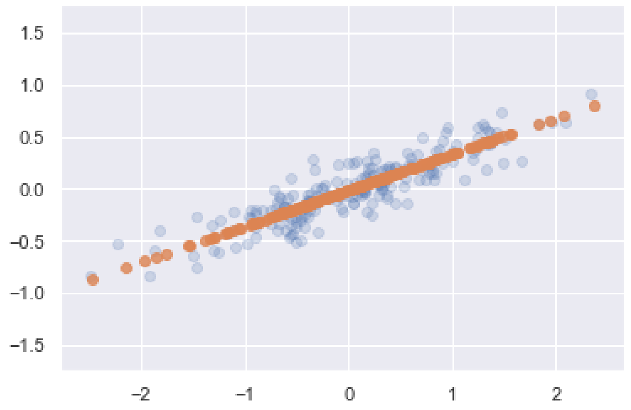
Here is an example of using PCA as dimensionality reduction transform:

```
pca = PCA(n_components=1)
pca.fit(X)
X_pca = pca.transform(X)
print("original shape:   ", X.shape)
print("transformed shape:", X_pca.shape)
```

```
original shape:    (200, 2)
transformed shape: (200, 1)
```

The transformed data has been reduced to a single dimension. To understand the effect of the dimensionality reduction, we can perform the inverse transform of this reduced data and plot it along with the original data.

```
X_new = pca.inverse_transform(X_pca)
plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
plt.scatter(X_new[:, 0], X_new[:, 1], alpha=0.8)
plt.axis('equal');
```



The light points are the original data, while the dark points are the projected version. This makes clear what a PCA dimensionality reduction means: the information along the least important principal axis or axes is removed, leaving only the component(s) of the data with the highest variance. The fraction of variance that is cut out (proportional to the spread of points about the line formed in this figure) is roughly a measure of how much "information" is discarded in this reduction of dimensionality.

This reduced-dimension dataset is in some senses "good enough" to encode the most important relationships between the points: despite reducing the dimension of the data by 50%, the overall relationship between the data points are mostly preserved. How much does the projected data retain the information from the original data?

Answer: (show your Python code here along with the answer)

Choosing the number of component

A vital part of using PCA in practice is the ability to estimate how many components are needed to describe the data. We can determine this by looking at the cumulative explained variance ratio as a function of the number of component.

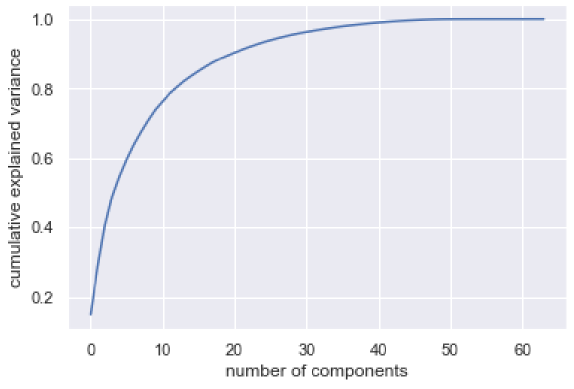
For this, we use the digit dataset from sklearn. First, we load the dataset:

```python
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

```
(1797, 64)
```

Calculating the cumulative explained variance ratio:

```python
pca = PCA().fit(digits.data)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```



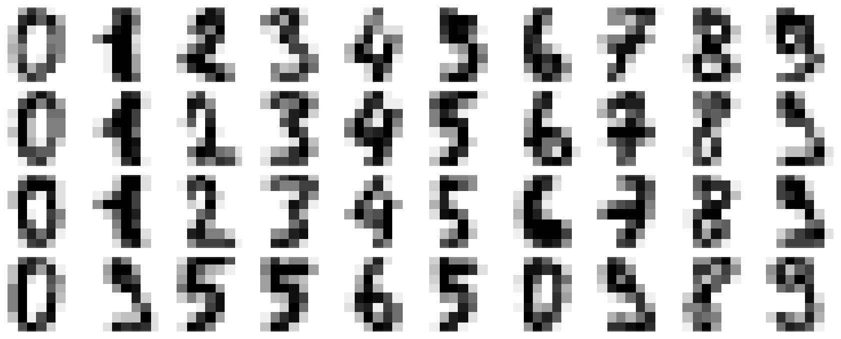
PCA as Noise Filtering

PCA can also be used as a filtering approach for noisy data. The idea is this : any components with variance much larger than the effect of the noise should be relatively unaffected by the noise. So if we reconstruct the data using just the largest subset of principal components, we should be preferentially keeping the signal and throwing out the noise.

With using a digit dataset, first we will plot several of the input noise-free data:

```python
def plot_digits(data):
    fig, axes = plt.subplots(4, 10, figsize=(10, 4),
                             subplot_kw={'xticks':[], 'yticks':[]},
                             gridspec_kw=dict(hspace=0.1, wspace=0.1))
    for i, ax in enumerate(axes.flat):
        ax.imshow(data[i].reshape(8, 8),
                  cmap='binary', interpolation='nearest',
                  clim=(0, 16))
plot_digits(digits.data)
```

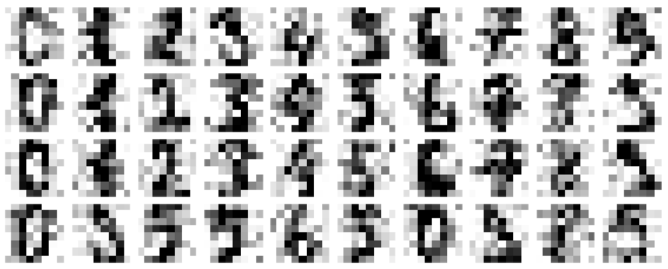
So, the figure of digits without noise is:



Now let's add some random noise to create a noisy dataset, and replot it:

```python
np.random.seed(42)
noisy = np.random.normal(digits.data, 4)
plot_digits(noisy)
```

The output of Gaussian random noise added is :



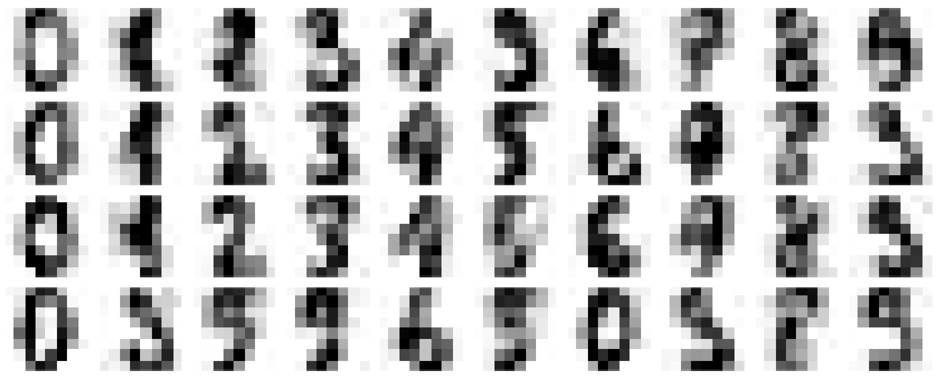
It's clear by eye that the images are noisy and contain spurious pixels. Let's train a PCA on the noisy data, requesting that the projection preserve 50% of the variance:

```python
pca = PCA(0.50).fit(noisy)
pca.n_components_
```

```
12
```

Here 50% of the variance amounts to 12 principal components. Now we can compute these components and then use the inverse of the transform to reconstruct the filtered digits:

```
components = pca.transform(noisy)
filtered = pca.inverse_transform(components)
plot_digits(filtered)
```



From the figure (digit “denoised” using PCA), this signal preserving/noise filtering property makes a PCA very useful feature selection routine. For example, rather than training a classifier on very high-dimensional data, we might instead training the classifier on the lower dimensional representation, which will automatically serve to filter out random noise in the inputs.