PCA

PCA stands for Principal Component Analysis. It is a dimensionality reduction technique and a fundamental tool in machine learning and statistics. PCA is used to simplify complex datasets by reducing the number of variables (or dimensions) while retaining the most important information and patterns in the data. Here's how PCA works:

1. **Data Standardization**:
   - Before applying PCA, it's common practice to standardize the data by subtracting the mean and scaling to unit variance. Standardization ensures that all variables have the same scale, which is important for PCA to work effectively.
2. **Covariance Matrix Calculation**:
   - PCA calculates the covariance matrix of the standardized data. The covariance matrix represents the relationships and dependencies between the variables.
3. **Eigendecomposition**:
   - PCA performs an eigendecomposition (eigenvalue decomposition) of the covariance matrix. This decomposition results in a set of eigenvectors and corresponding eigenvalues.
4. **Selecting Principal Components**:
   - The eigenvectors represent the principal components of the data. These components are orthogonal (uncorrelated) and sorted by their corresponding eigenvalues in descending order. The eigenvector with the highest eigenvalue is the first principal component, the second highest eigenvalue corresponds to the second principal component, and so on.
   - By selecting a subset of these principal components, you can reduce the dimensionality of the data while preserving as much variance (information) as possible.
5. **Projecting Data**:
   - To reduce the dimensionality of the data, you can project it onto a lower-dimensional subspace defined by the selected principal components. This projection retains the most significant information in the data while reducing noise and redundancy.
6. **Explained Variance**:
   - PCA provides information about the explained variance for each principal component. This information helps you understand how much of the total variance in the data is retained by each component. You can use this to determine how many principal components to keep to achieve a desired level of data compression or dimensionality reduction.

PCA is commonly used in various applications, including:

- **Data Visualization**: Reducing high-dimensional data to 2D or 3D for visualization and exploration.
- **Noise Reduction**: Removing noise and irrelevant features from data.

- **Feature Engineering**: Creating new features (principal components) that capture the most important information in the data.
- **Dimensionality Reduction**: Reducing the computational complexity of machine learning models and improving their generalization.
- **Face Recognition**: Reducing the dimensionality of image data while retaining essential facial features.

PCA is a powerful technique but should be used with careful consideration, as it may not always be appropriate for all datasets or tasks. It assumes linear relationships between variables and may not capture non-linear patterns effectively. In such cases, nonlinear dimensionality reduction techniques like t-SNE or autoencoders may be more suitable.

Dimension reduction is a fundamental technique in machine learning and data analysis that involves reducing the number of variables (dimensions) in a dataset while preserving important information. This reduction can lead to several benefits, including simplifying the data, improving model performance, reducing computational complexity, and enhancing interpretability. There are two primary approaches to dimension reduction:

1. **Feature Selection**:
   - Feature selection involves selecting a subset of the original features (variables) and discarding the rest. The selected features are considered the most relevant or informative for the task at hand.
   - Common methods for feature selection include:
     - Univariate feature selection: Selecting features based on statistical tests or measures like chi-squared, ANOVA, or mutual information.
     - Recursive feature elimination (RFE): Iteratively removing the least important features based on the performance of a machine learning model.
     - Feature importance scores from tree-based models: Extracting feature importance scores from decision trees or ensemble methods like Random Forest.
     - Correlation-based methods: Removing highly correlated features to reduce redundancy.
   - Feature selection is particularly useful when there are clear domain-specific reasons to focus on specific variables or when you want to improve the interpretability of a model.
2. **Feature Extraction**:
   - Feature extraction transforms the original features into a new set of features, typically of lower dimension, while retaining as much relevant information as possible. These new features are called "derived features" or "latent variables."
   - Principal Component Analysis (PCA) is a widely used technique for feature extraction. It identifies the principal components (linear combinations of the original features) that capture the maximum variance in the data. These principal components can be used as new features.
   - Other feature extraction methods include Independent Component Analysis (ICA), Linear Discriminant Analysis (LDA), and autoencoders in deep learning.

- Feature extraction is valuable when you want to reduce the dimensionality of the data while preserving important patterns and relationships. It's often used in tasks like image recognition, natural language processing, and signal processing.

The choice between feature selection and feature extraction depends on the nature of the data, the problem you're trying to solve, and your specific goals. In some cases, a combination of both techniques may be beneficial.

Dimension reduction is crucial when dealing with high-dimensional datasets, as the "curse of dimensionality" can lead to increased computational complexity and overfitting in machine learning models. By reducing the number of features, you can often improve the efficiency, accuracy, and interpretability of your models while maintaining or even enhancing their predictive power.

Importing Packages

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Importing Datasets

```
from sklearn import datasets

dir(datasets)
```

```
['__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__getattr__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '_arff_parser',
 '_base',
 '_california_housing',
 '_covtype',
 '_kddcup99',
 '_lfw',
 '_olivetti_faces',
 '_openml',
```

```
 '_rcv1',
 '_samples_generator',
 '_species_distributions',
 '_svmlight_format_fast',
 '_svmlight_format_io',
 '_twenty_newsgroups',
 'clear_data_home',
 'dump_svmlight_file',
 'fetch_20newsgroups',
 'fetch_20newsgroups_vectorized',
 'fetch_california_housing',
 'fetch_covtype',
 'fetch_kddcup99',
 'fetch_lfw_pairs',
 'fetch_lfw_people',
 'fetch_olivetti_faces',
 'fetch_openml',
 'fetch_rcv1',
 'fetch_species_distributions',
 'get_data_home',
 'load_breast_cancer',
 'load_diabetes',
 'load_digits',
 'load_files',
 'load_iris',
 'load_linnerud',
 'load_sample_image',
 'load_sample_images',
 'load_svmlight_file',
 'load_svmlight_files',
 'load_wine',
 'make_biclusters',
 'make_blobs',
 'make_checkerboard',
 'make_circles',
 'make_classification',
 'make_friedman1',
 'make_friedman2',
 'make_friedman3',
 'make_gaussian_quantiles',
 'make_hastie_10_2',
 'make_low_rank_matrix',
 'make_moons',
 'make_multilabel_classification',
 'make_regression',
 'make_s_curve',
 'make_sparse_coded_signal',
 'make_sparse_spd_matrix',
 'make_sparse_uncorrelated',
```

```
 'make_spd_matrix',
 'make_swiss_roll',
 'textwrap']
```

```python
from sklearn.datasets import load_digits
```

```python
digits=load_digits()
```

```python
digits
```

```
{'data': array([[ 0.,  0.,  5., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ..., 10.,  0.,  0.],
        [ 0.,  0.,  0., ..., 16.,  9.,  0.],
        ...,
        [ 0.,  0.,  1., ...,  6.,  0.,  0.],
        [ 0.,  0.,  2., ..., 12.,  0.,  0.],
        [ 0.,  0., 10., ..., 12.,  1.,  0.]]),
 'target': array([0, 1, 2, ..., 8, 9, 8]),
 'frame': None,
 'feature_names': ['pixel_0_0',
  'pixel_0_1',
  'pixel_0_2',
  'pixel_0_3',
  'pixel_0_4',
  'pixel_0_5',
  'pixel_0_6',
  'pixel_0_7',
  'pixel_1_0',
  'pixel_1_1',
  'pixel_1_2',
  'pixel_1_3',
  'pixel_1_4',
  'pixel_1_5',
  'pixel_1_6',
  'pixel_1_7',
  'pixel_2_0',
  'pixel_2_1',
  'pixel_2_2',
  'pixel_2_3',
  'pixel_2_4',
  'pixel_2_5',
  'pixel_2_6',
  'pixel_2_7',
  'pixel_3_0',
  'pixel_3_1',
  'pixel_3_2',
  'pixel_3_3',
  'pixel_3_4',
  'pixel_3_5',
  'pixel_3_6',
```

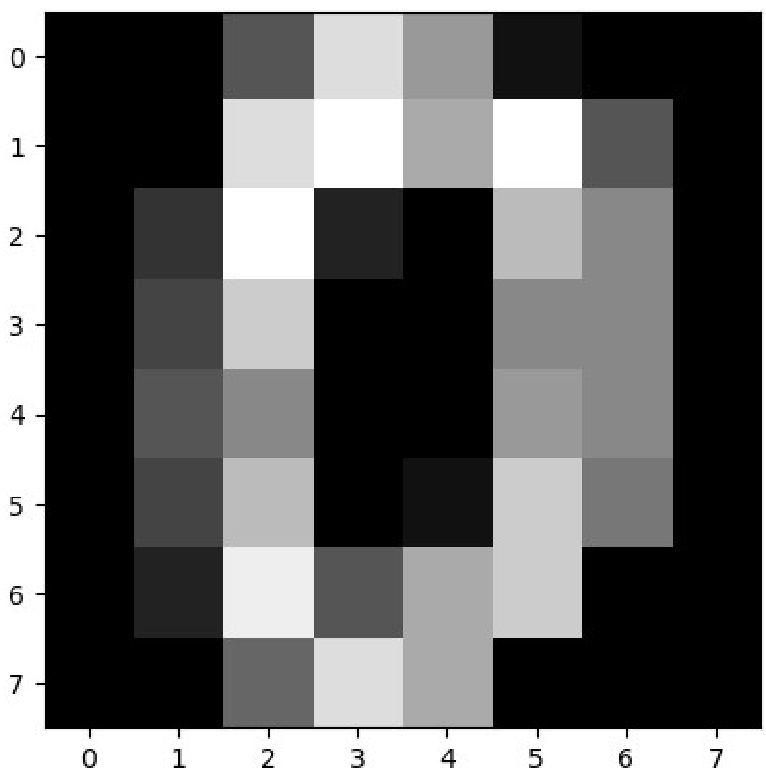
```
  'pixel_3_7',
  'pixel_4_0',
  'pixel_4_1',
  'pixel_4_2',
  'pixel_4_3',
  'pixel_4_4',
  'pixel_4_5',
  'pixel_4_6',
  'pixel_4_7',
  'pixel_5_0',
  'pixel_5_1',
  'pixel_5_2',
  'pixel_5_3',
  'pixel_5_4',
  'pixel_5_5',
  'pixel_5_6',
  'pixel_5_7',
  'pixel_6_0',
  'pixel_6_1',
  'pixel_6_2',
  'pixel_6_3',
  'pixel_6_4',
  'pixel_6_5',
  'pixel_6_6',
  'pixel_6_7',
  'pixel_7_0',
  'pixel_7_1',
  'pixel_7_2',
  'pixel_7_3',
  'pixel_7_4',
  'pixel_7_5',
  'pixel_7_6',
  'pixel_7_7'],
 'target_names': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
 'images': array([[[ 0.,  0.,  5., ...,  1.,  0.,  0.],
         [ 0.,  0., 13., ..., 15.,  5.,  0.],
         [ 0.,  3., 15., ..., 11.,  8.,  0.],
         ...,
         [ 0.,  4., 11., ..., 12.,  7.,  0.],
         [ 0.,  2., 14., ..., 12.,  0.,  0.],
         [ 0.,  0.,  6., ...,  0.,  0.,  0.]],

        [[ 0.,  0.,  0., ...,  5.,  0.,  0.],
         [ 0.,  0.,  0., ...,  9.,  0.,  0.],
         [ 0.,  0.,  3., ...,  6.,  0.,  0.],
         ...,
         [ 0.,  0.,  1., ...,  6.,  0.,  0.],
         [ 0.,  0.,  1., ...,  6.,  0.,  0.],
         [ 0.,  0.,  0., ..., 10.,  0.,  0.]],
```

```
        [[ 0.,  0.,  0., ..., 12.,  0.,  0.],
         [ 0.,  0.,  3., ..., 14.,  0.,  0.],
         [ 0.,  0.,  8., ..., 16.,  0.,  0.],
         ...,
         [ 0.,  9., 16., ...,  0.,  0.,  0.],
         [ 0.,  3., 13., ..., 11.,  5.,  0.],
         [ 0.,  0.,  0., ..., 16.,  9.,  0.]],

        ...,

        [[ 0.,  0.,  1., ...,  1.,  0.,  0.],
         [ 0.,  0., 13., ...,  2.,  1.,  0.],
         [ 0.,  0., 16., ..., 16.,  5.,  0.],
         ...,
         [ 0.,  0., 16., ..., 15.,  0.,  0.],
         [ 0.,  0., 15., ..., 16.,  0.,  0.],
         [ 0.,  0.,  2., ...,  6.,  0.,  0.]],

        [[ 0.,  0.,  2., ...,  0.,  0.,  0.],
         [ 0.,  0., 14., ..., 15.,  1.,  0.],
         [ 0.,  4., 16., ..., 16.,  7.,  0.],
         ...,
         [ 0.,  0.,  0., ..., 16.,  2.,  0.],
         [ 0.,  0.,  4., ..., 16.,  2.,  0.],
         [ 0.,  0.,  5., ..., 12.,  0.,  0.]],

        [[ 0.,  0., 10., ...,  1.,  0.,  0.],
         [ 0.,  2., 16., ...,  1.,  0.,  0.],
         [ 0.,  0., 15., ..., 15.,  0.,  0.],
         ...,
         [ 0.,  4., 16., ..., 16.,  6.,  0.],
         [ 0.,  8., 16., ..., 16.,  8.,  0.],
         [ 0.,  1.,  8., ..., 12.,  1.,  0.]]]),
 'DESCR': ".. _digits_dataset:\n\nOptical recognition of handwritten
digits dataset\n--------------------------------------------------\n\
n**Data Set Characteristics:**\n\n    :Number of Instances: 1797\
n    :Number of Attributes: 64\n    :Attribute Information: 8x8 image
of integer pixels in the range 0..16.\n    :Missing Attribute Values:
None\n    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)\n    :Date:
July; 1998\n\nThis is a copy of the test set of the UCI ML hand-
written digits
datasets\nhttps://archive.ics.uci.edu/ml/datasets/Optical+Recognition+
of+Handwritten+Digits\n\nThe data set contains images of hand-written
digits: 10 classes where\neach class refers to a digit.\n\
nPreprocessing programs made available by NIST were used to extract\
nnormalized bitmaps of handwritten digits from a preprinted form. From
a\ntotal of 43 people, 30 contributed to the training set and
different 13\nto the test set. 32x32 bitmaps are divided into
```

```
nonoverlapping blocks of\n4x4 and the number of on pixels are counted
in each block. This generates\nan input matrix of 8x8 where each
element is an integer in the range\n0..16. This reduces dimensionality
and gives invariance to small\ndistortions.\n\nFor info on NIST
preprocessing routines, see M. D. Garris, J. L. Blue, G.\nT. Candela,
D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.\nL.
Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,\
n1994.\n\n|details-start|\n**References**\n|details-split|\n\n- C.
Kaynak (1995) Methods of Combining Multiple Classifiers and Their\n
Applications to Handwritten Digit Recognition, MSc Thesis, Institute
of\n  Graduate Studies in Science and Engineering, Bogazici
University.\n- E. Alpaydin, C. Kaynak (1998) Cascading Classifiers,
Kybernetika.\n- Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and
A. Kai Qin.\n  Linear dimensionalityreduction using relevance weighted
LDA. School of\n  Electrical and Electronic Engineering Nanyang
Technological University.\n  2005.\n- Claudio Gentile. A New
Approximate Maximal Margin Classification\n  Algorithm. NIPS. 2000.\n\
n|details-end|"}
```

```
plt.imshow(digits.images[0],cmap=plt.cm.gray)
```

```
<matplotlib.image.AxesImage at 0x151288eda60>
```

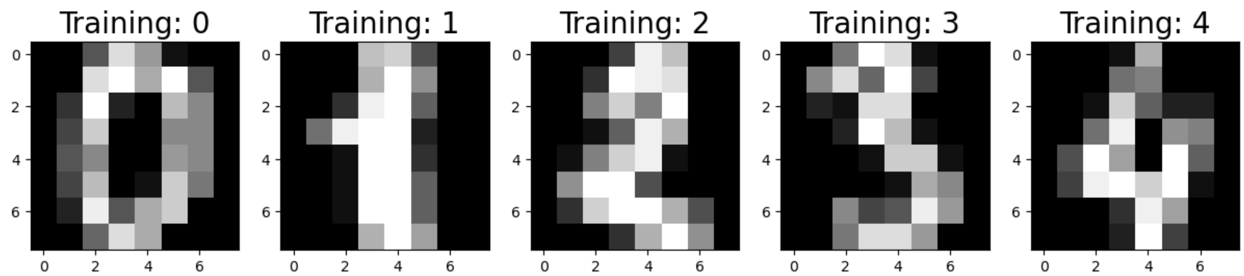



```
df=pd.DataFrame(digits.data)
```

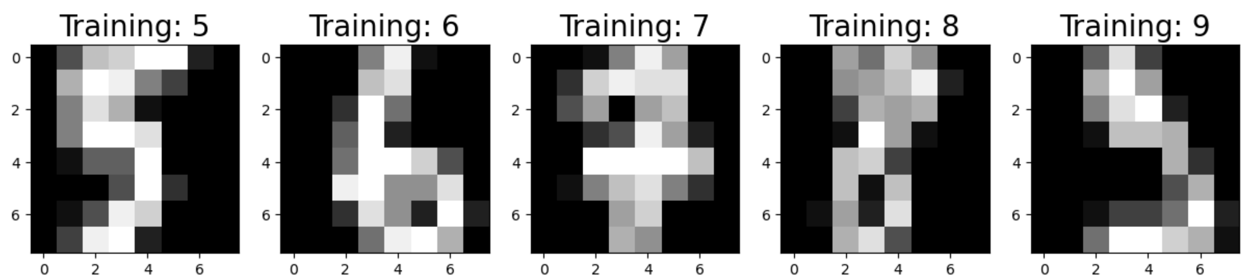
```
df
```

```
        0    1     2     3     4     5    6    7    8    9  ...   54  
55  \
0     0.0  0.0   5.0  13.0   9.0   1.0  0.0  0.0  0.0  0.0  ...  0.0  
0.0
1     0.0  0.0   0.0  12.0  13.0   5.0  0.0  0.0  0.0  0.0  ...  0.0  
0.0
2     0.0  0.0   0.0   4.0  15.0  12.0  0.0  0.0  0.0  0.0  ...  5.0  
0.0
3     0.0  0.0   7.0  15.0  13.0   1.0  0.0  0.0  0.0  8.0  ...  9.0  
0.0
4     0.0  0.0   0.0   1.0  11.0   0.0  0.0  0.0  0.0  0.0  ...  0.0  
0.0
...   ...  ...   ...   ...   ...   ...  ...  ...  ...  ...  ...  ...  
...
1792  0.0  0.0   4.0  10.0  13.0   6.0  0.0  0.0  0.0  1.0  ...  4.0  
0.0
1793  0.0  0.0   6.0  16.0  13.0  11.0  1.0  0.0  0.0  0.0  ...  1.0  
0.0
1794  0.0  0.0   1.0  11.0  15.0   1.0  0.0  0.0  0.0  0.0  ...  0.0  
0.0
1795  0.0  0.0   2.0  10.0   7.0   0.0  0.0  0.0  0.0  0.0  ...  2.0  
0.0
1796  0.0  0.0  10.0  14.0   8.0   1.0  0.0  0.0  0.0  2.0  ...  8.0  
0.0

       56   57   58    59    60    61   62   63  
0     0.0  0.0  6.0  13.0  10.0   0.0  0.0  0.0  
1     0.0  0.0  0.0  11.0  16.0  10.0  0.0  0.0  
2     0.0  0.0  0.0   3.0  11.0  16.0  9.0  0.0  
3     0.0  0.0  7.0  13.0  13.0   9.0  0.0  0.0  
4     0.0  0.0  0.0   2.0  16.0   4.0  0.0  0.0  
...   ...  ...  ...   ...   ...   ...  ...  ...  
1792  0.0  0.0  2.0  14.0  15.0   9.0  0.0  0.0  
1793  0.0  0.0  6.0  16.0  14.0   6.0  0.0  0.0  
1794  0.0  0.0  2.0   9.0  13.0   6.0  0.0  0.0  
1795  0.0  0.0  5.0  12.0  16.0  12.0  0.0  0.0  
1796  0.0  1.0  8.0  12.0  14.0  12.0  1.0  0.0  

[1797 rows x 64 columns]
```

```
plt.figure(figsize=(15,5))
for index,(image,label) in 
enumerate(zip(digits.data[0:5],digits.target[0:5])):
    plt.subplot(1,5,index+1)
    plt.imshow(np.reshape(image,(8,8)),cmap=plt.cm.gray)
    plt.title(f'Training: {label}',fontsize=20)
```




```
plt.figure(figsize=(15,5))
for index,(image,label) in
enumerate(zip(digits.data[15:20],digits.target[15:20])):
    plt.subplot(1,5,index+1)
    plt.imshow(np.reshape(image,(8,8)),cmap=plt.cm.gray)
    plt.title(f'Training: {label}',fontsize=20)
```




```
df.describe()
```

```
            0            1            2            3            4  \
count  1797.0  1797.000000  1797.000000  1797.000000  1797.000000
mean      0.0     0.303840     5.204786    11.835838    11.848080
std       0.0     0.907192     4.754826     4.248842     4.287388
min       0.0     0.000000     0.000000     0.000000     0.000000
25%       0.0     0.000000     1.000000    10.000000    10.000000
50%       0.0     0.000000     4.000000    13.000000    13.000000
75%       0.0     0.000000     9.000000    15.000000    15.000000
max       0.0     8.000000    16.000000    16.000000    16.000000

                 5            6            7            8            9
...  \
count  1797.000000  1797.000000  1797.000000  1797.000000  1797.000000
...
mean      5.781859     1.362270     0.129661     0.005565     1.993879
...
std       5.666418     3.325775     1.037383     0.094222     3.196160
...
min       0.000000     0.000000     0.000000     0.000000     0.000000
...
25%       0.000000     0.000000     0.000000     0.000000     0.000000
...
```

```
50%        4.000000     0.000000     0.000000     0.000000     0.000000
...
75%       11.000000     0.000000     0.000000     0.000000     3.000000
...
max       16.000000    16.000000    15.000000     2.000000    16.000000
...

                 54           55           56           57           58
\
count   1797.000000  1797.000000  1797.000000  1797.000000  1797.000000

mean       3.725097     0.206455     0.000556     0.279354     5.557596

std        4.919406     0.984401     0.023590     0.934302     5.103019

min        0.000000     0.000000     0.000000     0.000000     0.000000

25%        0.000000     0.000000     0.000000     0.000000     1.000000

50%        1.000000     0.000000     0.000000     0.000000     4.000000

75%        7.000000     0.000000     0.000000     0.000000    10.000000

max       16.000000    13.000000     1.000000     9.000000    16.000000


                 59           60           61           62           63

count   1797.000000  1797.000000  1797.000000  1797.000000  1797.000000

mean      12.089037    11.809126     6.764051     2.067891     0.364496

std        4.374694     4.933947     5.900623     4.090548     1.860122

min        0.000000     0.000000     0.000000     0.000000     0.000000

25%       11.000000    10.000000     0.000000     0.000000     0.000000

50%       13.000000    14.000000     6.000000     0.000000     0.000000

75%       16.000000    16.000000    12.000000     2.000000     0.000000

max       16.000000    16.000000    16.000000    16.000000    16.000000


[8 rows x 64 columns]
```

```
df.describe().T
```

```
     count      mean       std  min  25%  50%  75%  max
0   1797.0  0.000000  0.000000  0.0  0.0  0.0  0.0  0.0
1   1797.0  0.303840  0.907192  0.0  0.0  0.0  0.0  8.0
```

```
2   1797.0   5.204786  4.754826  0.0   1.0   4.0   9.0  16.0
3   1797.0  11.835838  4.248842  0.0  10.0  13.0  15.0  16.0
4   1797.0  11.848080  4.287388  0.0  10.0  13.0  15.0  16.0
..     ...        ...       ...  ...   ...   ...   ...   ...
59  1797.0  12.089037  4.374694  0.0  11.0  13.0  16.0  16.0
60  1797.0  11.809126  4.933947  0.0  10.0  14.0  16.0  16.0
61  1797.0   6.764051  5.900623  0.0   0.0   6.0  12.0  16.0
62  1797.0   2.067891  4.090548  0.0   0.0   0.0   2.0  16.0
63  1797.0   0.364496  1.860122  0.0   0.0   0.0   0.0  16.0

[64 rows x 8 columns]
```

```
digits.data.shape
```

```
(1797, 64)
```

```
digits.images.shape
```

```
(1797, 8, 8)
```

```
x=digits.data
y=digits.target
```

```
x
```

```
array([[ 0.,  0.,  5., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ..., 10.,  0.,  0.],
       [ 0.,  0.,  0., ..., 16.,  9.,  0.],
       ...,
       [ 0.,  0.,  1., ...,  6.,  0.,  0.],
       [ 0.,  0.,  2., ..., 12.,  0.,  0.],
       [ 0.,  0., 10., ..., 12.,  1.,  0.]])
```

```
y
```

```
array([0, 1, 2, ..., 8, 9, 8])
```

```
from sklearn.preprocessing import StandardScaler
```

```
x_std=StandardScaler().fit_transform(x)
```

```
x_std.shape
```

```
(1797, 64)
```

```
x_std
```

```
array([[ 0.        , -0.33501649, -0.04308102, ..., -1.14664746,
        -0.5056698 , -0.19600752],
       [ 0.        , -0.33501649, -1.09493684, ...,  0.54856067,
        -0.5056698 , -0.19600752],
       [ 0.        , -0.33501649, -1.09493684, ...,  1.56568555,
         1.6951369 , -0.19600752],
```

```
       ...,
       [ 0.        , -0.33501649, -0.88456568, ..., -0.12952258,
        -0.5056698 , -0.19600752],
       [ 0.        , -0.33501649, -0.67419451, ...,  0.8876023 ,
        -0.5056698 , -0.19600752],
       [ 0.        , -0.33501649,  1.00877481, ...,  0.8876023 ,
        -0.26113572, -0.19600752]])
```

```
x1=x_std.T
```

```
x1
```

```
array([[ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [-0.33501649, -0.33501649, -0.33501649, ..., -0.33501649,
        -0.33501649, -0.33501649],
       [-0.04308102, -1.09493684, -1.09493684, ..., -0.88456568,
        -0.67419451,  1.00877481],
       ...,
       [-1.14664746,  0.54856067,  1.56568555, ..., -0.12952258,
         0.8876023 ,  0.8876023 ],
       [-0.5056698 , -0.5056698 ,  1.6951369 , ..., -0.5056698 ,
        -0.5056698 , -0.26113572],
       [-0.19600752, -0.19600752, -0.19600752, ..., -0.19600752,
        -0.19600752, -0.19600752]])
```

```
cov_mat=np.cov(x_std.T)
```

```
cov_mat
```

```
array([[ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  1.00055679,  0.55692803, ..., -0.02988686,
         0.02656195, -0.04391324],
       [ 0.        ,  0.55692803,  1.00055679, ..., -0.04120565,
         0.07263924,  0.08256908],
       ...,
       [ 0.        , -0.02988686, -0.04120565, ...,  1.00055679,
         0.64868875,  0.26213704],
       [ 0.        ,  0.02656195,  0.07263924, ...,  0.64868875,
         1.00055679,  0.62077355],
       [ 0.        , -0.04391324,  0.08256908, ...,  0.26213704,
         0.62077355,  1.00055679]])
```

```
eig_vals,eig_vecs=np.linalg.eig(cov_mat)
```

```
eig_vals
```

```
array([7.34477606, 5.83549054, 5.15396118, 3.96623597, 2.9663452 ,
       2.57204442, 2.40600941, 2.06867355, 1.82993314, 1.78951739,
       1.69784616, 1.57287889, 1.38870781, 1.35933609, 1.32152536,
```

```
        1.16829176, 1.08368678, 0.99977862, 0.97438293, 0.90891242,
        0.82271926, 0.77631014, 0.71155675, 0.64552365, 0.59527399,
        0.5765018 , 0.52673155, 0.5106363 , 0.48686381, 0.45560107,
        0.44285155, 0.42230086, 0.3991063 , 0.39110111, 0.36094517,
        0.34860306, 0.3195963 , 0.29406627, 0.27692285, 0.05037444,
        0.06328961, 0.258273  , 0.24783029, 0.2423566 , 0.07635394,
        0.08246812, 0.09018543, 0.09840876, 0.10250434, 0.11188655,
        0.11932898, 0.12426371, 0.13321081, 0.14311427, 0.217582  ,
        0.15818474, 0.16875236, 0.20799593, 0.17612894, 0.2000909 ,
        0.18983516, 0.        , 0.        , 0.        ])
```

```
eig_vecs
```

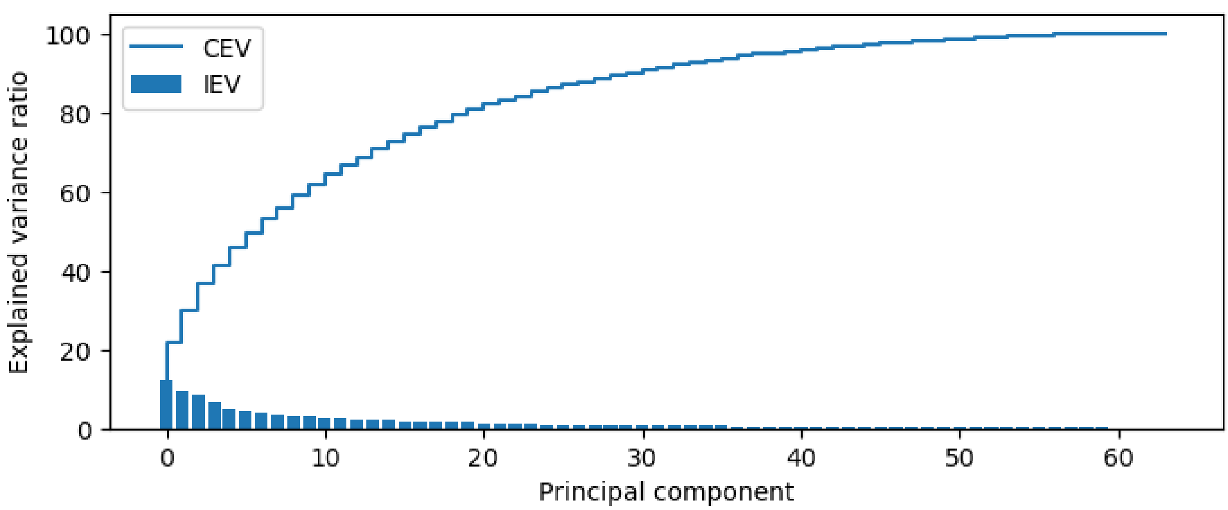
```
array([[ 0.        ,  0.        ,  0.        , ...,  1.        ,
         0.        ,  0.        ],
       [ 0.18223392, -0.04702701,  0.02358821, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.285868  , -0.0595648 , -0.05679875, ...,  0.        ,
         0.        ,  0.        ],
       ...,
       [ 0.103198  ,  0.24261778, -0.02227952, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.1198106 ,  0.16508926,  0.10036559, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.07149362,  0.07132924,  0.09244589, ...,  0.        ,
         0.        ,  0.        ]])
```

```
tot=sum(eig_vals)
var_exp=[(i/tot)*100  for i in sorted (eig_vals,reverse=True)]
#individual explained variance
var_exp
```

```
[12.033916097734924,
 9.561054403097929,
 8.444414892624557,
 6.498407907524173,
 4.860154875966378,
 4.214119869271917,
 3.9420828035673727,
 3.389380924638341,
 2.99822101162524,
 2.932002551252232,
 2.781805463550298,
 2.5770550925820013,
 2.275303315764233,
 2.227179739514354,
 2.1652294318492604,
 1.9141666064421274,
 1.7755470851681776,
 1.6380692742844212,
```

```
 1.5964601688623297,
 1.4891911870878158,
 1.3479695658179398,
 1.2719313702347623,
 1.1658373505919524,
 1.0576465985363175,
 0.9753159471981054,
 0.9445589897320013,
 0.8630138269707204,
 0.8366428536685098,
 0.7976932484112416,
 0.7464713709260621,
 0.725582151370273,
 0.6919112454811898,
 0.6539085355726164,
 0.6407925738459953,
 0.5913841117223396,
 0.5711624052235232,
 0.5236368034166312,
 0.4818075864451403,
 0.45371925985844797,
 0.42316275323278074,
 0.40605306997903756,
 0.397084808275829,
 0.356493303142619,
 0.34078718147030146,
 0.32783533528795433,
 0.3110320073453561,
 0.28857529410893434,
 0.2764892635235449,
 0.25917494088146487,
 0.23448300553563436,
 0.2182568577120083,
 0.2035976345253764,
 0.19551242601981672,
 0.18331849919718188,
 0.16794638749558172,
 0.16123606225672593,
 0.14776269410608878,
 0.13511841133708571,
 0.12510074249730258,
 0.10369573015571854,
 0.08253509448180095,
 0.0,
 0.0,
 0.0]

cum_var_exp=np.cumsum(var_exp)
cum_var_exp
```

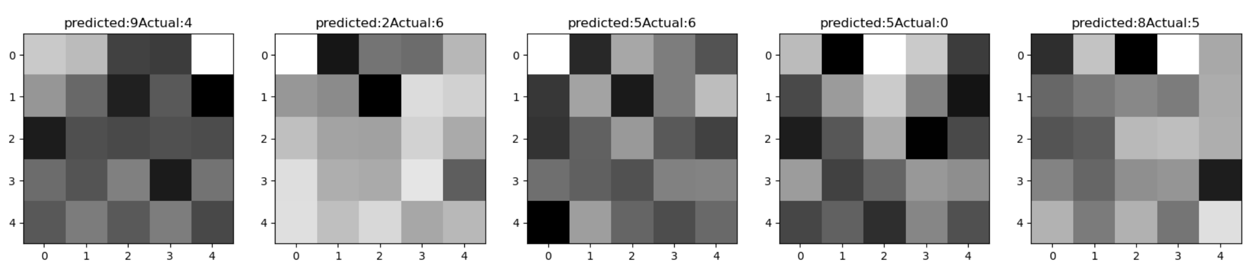
```
array([ 12.0339161 ,  21.5949705 ,  30.03938539,  36.5377933 ,
        41.39794818,  45.61206805,  49.55415085,  52.94353177,
        55.94175279,  58.87375534,  61.6555608 ,  64.23261589,
        66.50791921,  68.73509895,  70.90032838,  72.81449499,
        74.59004207,  76.22811135,  77.82457152,  79.3137627 ,
        80.66173227,  81.93366364,  83.09950099,  84.15714759,
        85.13246353,  86.07702252,  86.94003635,  87.77667921,
        88.57437245,  89.32084382,  90.04642598,  90.73833722,
        91.39224576,  92.03303833,  92.62442244,  93.19558485,
        93.71922165,  94.20102924,  94.6547485 ,  95.07791125,
        95.48396432,  95.88104913,  96.23754243,  96.57832961,
        96.90616495,  97.21719696,  97.50577225,  97.78226151,
        98.04143645,  98.27591946,  98.49417632,  98.69777395,
        98.89328638,  99.07660488,  99.24455127,  99.40578733,
        99.55355002,  99.68866843,  99.81376918,  99.91746491,
       100.        , 100.        , 100.        , 100.        ])

plt.figure(figsize=(8,3))
plt.bar(range(len(cum_var_exp)),var_exp,label='IEV')
plt.step(range(len(cum_var_exp)),cum_var_exp,label='CEV')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal component')
plt.legend()
plt.show()
```




```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x_std,y,train_size=0.8)

from sklearn.tree import DecisionTreeClassifier

from sklearn.decomposition import PCA
pca=PCA(n_components=25)
```

```
pca_x_train=pca.fit_transform(x_train)
pca_x_test=pca.transform(x_test)

rf=DecisionTreeClassifier().fit(pca_x_train,y_train)

predicted=rf.predict(pca_x_test)

predicted

array([6, 1, 7, 5, 7, 5, 0, 2, 7, 5, 9, 0, 9, 4, 7, 6, 6, 4, 6, 5, 2,
0,
       9, 3, 0, 2, 5, 3, 5, 8, 1, 1, 3, 3, 0, 8, 1, 1, 8, 7, 8, 5, 1,
7,
       6, 8, 4, 6, 8, 6, 2, 4, 1, 4, 0, 2, 6, 7, 8, 5, 0, 9, 6, 9, 4,
2,
       8, 0, 1, 9, 4, 8, 3, 5, 3, 1, 5, 2, 9, 5, 9, 7, 4, 7, 8, 4, 6,
8,
       1, 9, 0, 9, 4, 6, 1, 2, 8, 1, 9, 5, 5, 4, 1, 6, 2, 8, 3, 4, 4,
9,
       9, 2, 6, 4, 8, 2, 0, 8, 8, 8, 1, 2, 6, 7, 5, 8, 9, 8, 2, 7, 7,
5,
       0, 9, 6, 7, 2, 1, 9, 6, 9, 2, 0, 8, 0, 3, 5, 3, 7, 9, 7, 1, 8,
8,
       3, 6, 0, 1, 5, 0, 7, 1, 9, 9, 8, 2, 7, 7, 2, 5, 7, 2, 7, 1, 3,
1,
       5, 3, 1, 5, 3, 6, 4, 5, 0, 8, 7, 3, 3, 9, 6, 3, 5, 7, 5, 9, 5,
9,
       4, 3, 6, 4, 0, 7, 1, 7, 6, 0, 2, 1, 5, 9, 5, 5, 1, 7, 9, 9, 8,
1,
       1, 2, 3, 7, 9, 3, 5, 8, 8, 5, 9, 3, 9, 4, 1, 4, 7, 8, 5, 4, 6,
9,
       8, 7, 1, 4, 4, 1, 9, 0, 3, 8, 6, 8, 4, 4, 6, 4, 3, 5, 4, 4, 3,
0,
       3, 9, 0, 3, 6, 5, 1, 3, 1, 8, 7, 1, 8, 5, 0, 5, 8, 4, 8, 4, 0,
3,
       4, 5, 1, 5, 3, 0, 7, 2, 2, 9, 1, 7, 5, 8, 4, 0, 4, 1, 2, 6, 9,
3,
       1, 3, 6, 3, 1, 2, 3, 0, 3, 8, 9, 4, 1, 4, 2, 1, 7, 5, 6, 7, 5,
5,
       8, 1, 7, 0, 9, 6, 3, 4, 0, 4, 2, 5, 6, 2, 9, 0, 6, 1, 3, 0, 8,
0,
       1, 6, 2, 2, 2, 7, 9, 5])

y_train

array([1, 0, 8, ..., 2, 8, 9])

from sklearn.metrics import
accuracy_score,confusion_matrix,classification_report

accuracy_score(predicted,y_test)
```

```
0.8222222222222222

classification_report(y_test,predicted)

'              precision    recall  f1-score   support\n\n           0
0.87      0.84      0.86        32\n           1       0.81      0.87
0.84        39\n           2       0.80      0.80      0.80        30\
n           3       0.71      0.69      0.70        36\n           4
0.92      0.87      0.89        38\n           5       0.83      0.81
0.82        43\n           6       0.88      0.88      0.88        32\
n           7       0.94      0.85      0.89        39\n           8
0.69      0.73      0.71        37\n           9       0.79      0.88
0.83        34\n\n    accuracy                           0.82
360\n   macro avg       0.82      0.82      0.82       360\nweighted
avg       0.83      0.82      0.82       360\n'

def get_misclassifed_index(y_pred,y_test):
    misclassification=[]
    for index,(predicted,actual) in enumerate(zip(y_pred,y_test)):
        if predicted!=actual:
            misclassification.append(index)
    return misclassification

misclassification=get_misclassifed_index(predicted,y_test)

len(misclassification)

64

misclassification[0:5]

[22, 25, 26, 28, 29]

def plot_misclassification(misclassification):
    plt.figure(figsize=(20,4))
    for index,wrong in enumerate(misclassification[0:5]):
        plt.subplot(1,5,index+1)
        plt.imshow(np.reshape(pca_x_test[wrong],
(5,5)),cmap=plt.cm.gray)
        plt.title('predicted:{}Actual:
{}'.format(predicted[wrong],y_test[wrong]))

plot_misclassification(misclassification)
```



```python
def get_classifed_index(y_pred,y_test):
    classification=[]
    for index,(predicted,actual) in enumerate(zip(y_pred,y_test)):
        if predicted==actual:
            classification.append(index)
    return classification
```

```python
classification=get_classifed_index(predicted,y_test)
```

```python
for i in range(len(classification)):
    print(classification[i])
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
23
24
27
31
32
33
34
36
37
38
39
40
41
42
43
44
```

```
45
46
47
49
50
51
53
54
56
57
58
59
60
61
63
65
66
67
68
69
70
71
72
73
74
75
76
77
79
80
81
82
83
84
85
86
87
89
91
93
94
95
96
97
99
100
101
103
104
```

```
105
106
107
108
109
110
111
112
113
114
116
117
118
120
122
123
124
125
126
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
145
146
147
148
151
152
153
156
157
158
160
161
163
164
```

165
166
167
168
169
170
171
172
173
174
175
176
178
179
180
181
182
183
184
185
186
187
188
190
191
192
193
195
196
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217

```
219
220
221
223
224
225
226
229
230
232
234
235
236
237
238
239
240
241
242
243
245
246
248
249
252
254
255
256
257
260
261
262
263
264
265
266
268
269
270
271
274
276
277
278
279
280
281
282
283
```

```
284
285
286
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
306
307
308
309
310
312
313
317
318
319
320
321
322
323
324
325
326
329
331
332
333
334
335
336
337
338
339
340
342
```

```
343
345
346
347
348
349
350
351
352
354
357
358
359
```

```
len(classification)
```

```
288
```

```
def plot_classification(classification):
    plt.figure(figsize=(20,4))
    for index,correct in enumerate(classification[0:5]):
        plt.subplot(1,5,index+1)
        plt.imshow(np.reshape(pca_x_test[correct],
(5,5)),cmap=plt.cm.gray)
        plt.title('predicted:{}Actual:
{}'.format(predicted[correct],y_test[correct]))

plot_classification(classification)
```