The idea of PCA is simple — reduce the number of variables of a data set, while preserving as much information as possible.

Principal components are new variables that are constructed as linear combinations or mixtures of the initial variables. These combinations are done in such a way that the new variables (i.e., principal components) are uncorrelated and most of the information within the initial variables is squeezed or compressed into the first components.

HOW DO YOU DO A PRINCIPAL COMPONENT ANALYSIS?

step 1 : Standardize the range of continuous initial variables

step 2 : Compute the covariance matrix to identify correlations

step 3 : Compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components

step 4 : Create a feature vector to decide which principal components to keep (sorting the eigen values in descending order)

step 5 : Recast the data along the principal components axes

In [1]:
```
from sklearn import datasets
from sklearn.datasets import load_digits
```

In [2]:
```
dataset = load_digits()
dataset
```

```
  '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',
```

In [3]:
```
dataset.keys()
```

Out[3]:
```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'im
ages', 'DESCR'])
```

In [4]:
```
import numpy as np
import pandas as pd
df = pd.DataFrame(dataset.data, columns = dataset.feature_names)
df.head()
```

Out[4]:

	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
0	0.0	0.0	5.0	13.0	9.0	1.0	0.0	0.0	
1	0.0	0.0	0.0	12.0	13.0	5.0	0.0	0.0	
2	0.0	0.0	0.0	4.0	15.0	12.0	0.0	0.0	
3	0.0	0.0	7.0	15.0	13.0	1.0	0.0	0.0	
4	0.0	0.0	0.0	1.0	11.0	0.0	0.0	0.0	

5 rows × 64 columns

In [5]:
```
df['Target'] = dataset.target
```

In [6]:
```
df.shape
```

Out[6]:
```
(1797, 65)
```

In [7]:
```
print("Target is :" , dataset.target[0])  # printing only first row means f
dataset.data[0]
```

```
Target is : 0
```

Out[7]:
```
array([ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.,  0.,  0., 13., 15., 10.,
       15.,  5.,  0.,  0.,  3., 15.,  2.,  0., 11.,  8.,  0.,  0.,  4.,
       12.,  0.,  0.,  8.,  8.,  0.,  0.,  5.,  8.,  0.,  0.,  9.,  8.,
        0.,  0.,  4., 11.,  0.,  1., 12.,  7.,  0.,  0.,  2., 14.,  5.,
       10., 12.,  0.,  0.,  0.,  0.,  6., 13., 10.,  0.,  0.,  0.])
```

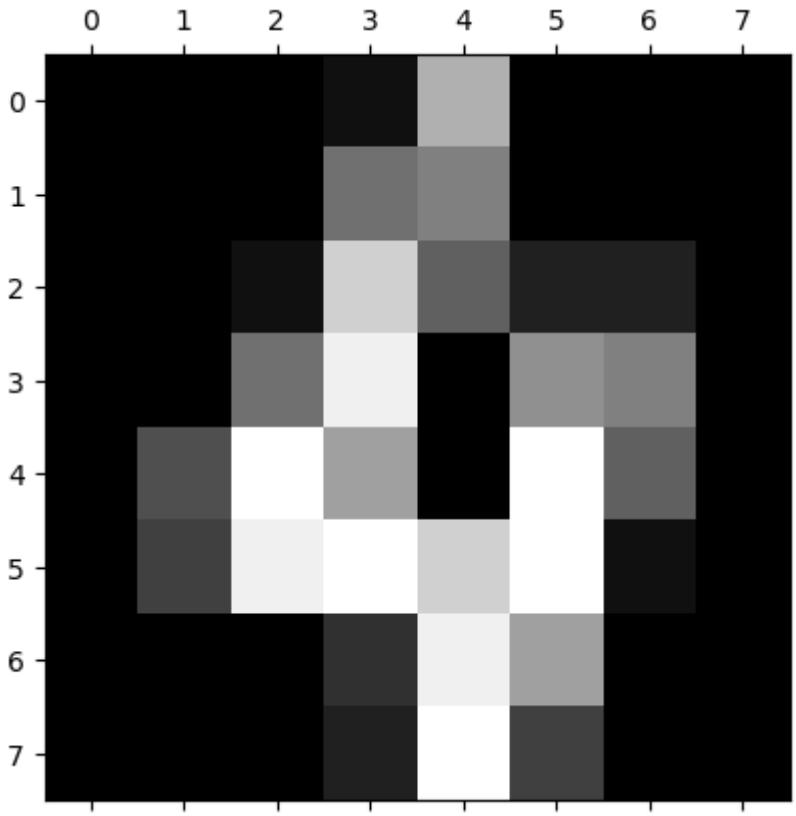
In [8]:
```
dataset.data[0].reshape(8,8)
```

Out[8]:
```
array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.],
       [ 0.,  0., 13., 15., 10., 15.,  5.,  0.],
       [ 0.,  3., 15.,  2.,  0., 11.,  8.,  0.],
       [ 0.,  4., 12.,  0.,  0.,  8.,  8.,  0.],
       [ 0.,  5.,  8.,  0.,  0.,  9.,  8.,  0.],
       [ 0.,  4., 11.,  0.,  1., 12.,  7.,  0.],
       [ 0.,  2., 14.,  5., 10., 12.,  0.,  0.],
       [ 0.,  0.,  6., 13., 10.,  0.,  0.,  0.]])
```

In [9]:
```
print("Target is :" , dataset.target[4])
from matplotlib import pyplot as plt
%matplotlib inline
plt.gray()
plt.matshow(dataset.data[4].reshape(8,8))
plt.show()
```

```
Target is : 4

<Figure size 640x480 with 0 Axes>
```



In [10]:
```
x = df                          #stored dataset in variable x and target colu
y = dataset.target
```

In [11]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(x)  #scaled only X which contains our datafi
```

In [12]:
```
pd.DataFrame(X_scaled)
```

Out[12]:

	0	1	2	3	4	5	6	7	
0	0.0	-0.335016	-0.043081	0.274072	-0.664478	-0.844129	-0.409724	-0.125023	-0.05907
1	0.0	-0.335016	-1.094937	0.038648	0.268751	-0.138020	-0.409724	-0.125023	-0.05907
2	0.0	-0.335016	-1.094937	-1.844742	0.735366	1.097673	-0.409724	-0.125023	-0.05907
3	0.0	-0.335016	0.377661	0.744919	0.268751	-0.844129	-0.409724	-0.125023	-0.05907
4	0.0	-0.335016	-1.094937	-2.551014	-0.197863	-1.020657	-0.409724	-0.125023	-0.05907
...	...	...	...	...	...	...	...	...	
1792	0.0	-0.335016	-0.253452	-0.432200	0.268751	0.038508	-0.409724	-0.125023	-0.05907
1793	0.0	-0.335016	0.167290	0.980343	0.268751	0.921145	-0.108958	-0.125023	-0.05907
1794	0.0	-0.335016	-0.884566	-0.196776	0.735366	-0.844129	-0.409724	-0.125023	-0.05907
1795	0.0	-0.335016	-0.674195	-0.432200	-1.131092	-1.020657	-0.409724	-0.125023	-0.05907
1796	0.0	-0.335016	1.008775	0.509495	-0.897785	-0.844129	-0.409724	-0.125023	-0.05907

1797 rows × 65 columns

In PCA (Principal Component Analysis), the covariance matrix plays a crucial role in determining the principal components of the dataset. The covariance matrix provides information about the relationships between pairs of features (variables) in the dataset.

Here's what the covariance matrix tells us in PCA:

Diagonal Elements: The diagonal elements of the covariance matrix represent the variances of individual features. These variances indicate the spread or dispersion of each feature in the dataset.

Off-diagonal Elements: The off-diagonal elements represent the covariances between pairs of features. Covariance measures how much two variables change together. A positive covariance indicates that the two variables tend to increase or decrease together, while a negative covariance indicates that they tend to change in opposite directions.

Eigenvalues and Eigenvectors: In PCA, we compute the eigenvalues and eigenvectors of the covariance matrix. The eigenvalues represent the variance explained by each principal component, while the eigenvectors represent the direction (or axis) of the principal components. The eigenvectors with the largest eigenvalues (i.e., the principal components with the highest variance) capture the most important information in the dataset.

Overall, the covariance matrix provides valuable information about the relationships between features in the dataset, which is used to compute the principal components in PCA.

In [13]:
```
# Calculate the covariance matrix
cov_mat = np.cov(X_scaled.T)
# Convert the covariance matrix into a DataFrame
pd.DataFrame(cov_mat)
```

Out[13]:

	0	1	2	3	4	5	6	7	8
0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.0	1.000557	0.556928	0.207929	-0.018771	0.060810	0.048415	-0.038948	0.032338
2	0.0	0.556928	1.000557	0.560492	-0.084282	0.043593	0.002842	-0.062313	0.022324
3	0.0	0.207929	0.560492	1.000557	0.023952	-0.171473	-0.115796	-0.040162	0.035683
4	0.0	-0.018771	-0.084282	0.023952	1.000557	0.508014	0.127835	0.010070	0.042089
...	...	...	...	...	...	...	...	...	...
60	0.0	-0.102406	-0.134829	-0.065993	-0.082171	-0.351342	-0.383735	-0.178343	0.049024
61	0.0	-0.029887	-0.041206	-0.054966	-0.215929	-0.268967	-0.304281	-0.141253	0.033428
62	0.0	0.026562	0.072639	0.053467	-0.250838	-0.267808	-0.179044	-0.063256	0.020700
63	0.0	-0.043913	0.082569	0.082016	-0.215469	-0.167821	-0.080354	-0.024519	-0.005229
64	0.0	-0.051863	-0.011843	-0.011496	0.100857	0.193469	0.197453	0.101141	0.020825

65 rows × 65 columns

In [14]:
```
# Extract only features and scale the data using StandardScaler() , compute
np.min(cov_mat)
```

Out[14]:
```
-0.5705136332381068
```

```
The NumPy function np.linalg.eig() calculates the eigenvalues and
eigenvectors of a square matrix. In the context of PCA (Principal
Component Analysis), this function is commonly used to compute the
eigenvalues and eigenvectors of the covariance matrix, where each
eigenvector represents a principal component and its corresponding
eigenvalue represents the amount of variance explained by that principal
component.
```

In [15]:
```
eig_vals, eig_vecs = np.linalg.eig(cov_mat)
```

```
To check the maximum components to be selected"

The cumulative explained variance helps in understanding how much of the
total variance in the dataset is captured by including a certain number of
principal components. By examining the cumulative explained variance plot,
one can determine the number of principal components needed to retain a
certain percentage of the total variance. This is crucial for
dimensionality reduction because it helps in deciding the appropriate
number of principal components to retain while reducing the dimensionality
of the dataset. Therefore, the cumulative explained variance is essential
for determining the maximum number of components to be selected in PCA.
```

```
eig_vals.count()When you extract only features and scale the data using
StandardScaler, the number of eigenvalues calculated will be equal to the
number of features in your dataset. Each eigenvalue corresponds to a
principal component, and each principal component represents a linear
combination of the original features.

So, if your dataset has n features, you will calculate n eigenvalues when
performing Principal Component Analysis (PCA) on the scaled features. Each
eigenvalue represents the amount of variance explained by the
corresponding principal component.

Here's a summary:

Number of features in the dataset: n
Number of eigenvalues calculated in PCA: n
Each eigenvalue will give you information about the amount of variance
captured by each principal component in the dataset.
```

In [16]:
```
pd.DataFrame(eig_vals)
```

Out[16]:

	0
0	7.344787
1	5.899764
2	5.163824
3	3.973668
4	2.987264
...	...
60	0.123930
61	0.130583
62	0.000000
63	0.000000
64	0.000000

65 rows × 1 columns

In [17]:
```
pd.DataFrame(eig_vecs)
```

Out[17]:

	0	1	2	3	4	5	6	7	
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
1	0.182237	-0.042855	-0.027187	-0.180053	-0.039724	-0.198227	-0.070058	0.172512	0.2
2	0.285850	-0.058624	0.052941	-0.158471	-0.022230	-0.110539	0.008109	0.038947	0.1
3	0.220377	0.019781	0.040743	-0.166430	0.052872	0.096353	0.215186	-0.098451	-0.1
4	-0.025253	-0.156829	0.060881	-0.003162	-0.134397	0.101930	-0.042911	0.014178	-0.0
...	...	...	...	...	...	...	...	...	
60	0.017902	0.249446	0.071507	0.154802	-0.016465	-0.123606	-0.164511	0.022629	-0.1
61	0.103292	0.237106	0.033417	0.256280	0.092863	-0.054465	-0.021724	0.051948	-0.0
62	0.119899	0.166247	-0.093141	0.163778	0.230682	-0.082454	0.055675	-0.149913	0.1
63	0.071568	0.076852	-0.090868	0.116709	0.202854	-0.119442	-0.020626	-0.304548	0.1
64	-0.001326	-0.116667	0.047425	0.051352	0.118815	0.224295	0.072747	0.344642	-0.0

65 rows × 65 columns

In [18]:
```
tot = sum(eig_vals)
var_exp = [(i/tot)*100 for i in sorted(eig_vals, reverse=True)] # Individua
var_exp
```

Out[18]: [11.839837653617856,
 9.510453831516248,
 8.324113357744103,
 6.405576010100892,
 4.815485963264094,
 4.2389143965875915,
 3.884052156077588,
 3.7345684726791153,
 2.9520170550496685,
 2.88471239490168,
 2.7704127880432554,
 2.5872515136652807,
 2.238715116272282,
 2.2282714480796644,
 2.1672535234911363,
 1.8936987942536048,
 1.75074807751154,
 1.7356070643735848,
 1.593396275325301,
 1.4656521833440017,
 1.3293589523407658,
 1.2823827528596727,
 1.1623017804875144,
 1.0516118037991222,
 1.0034524750729583,
 0.9383687526395528,
 0.8959589510973901,
 0.836112932939212,
 0.8004309072520539,
 0.7415017502003909,
 0.7141326686200253,
 0.6818199929147849,
 0.6433639778800367,
 0.6316994003932546,
 0.592525821486293,
 0.5633911941939794,
 0.5209818651995245,
 0.5057722914850954,
 0.4714997267285587,
 0.4389094070568951,
 0.4159919205433348,
 0.39069402135371767,
 0.3633234377582908,
 0.3505776373551296,
 0.333900166348013,
 0.3092718154424072,
 0.2957916990950438,
 0.2802858344904612,
 0.27087345882441594,
 0.24831780433641043,
 0.2306992699251755,
 0.21050067487937107,
 0.19977573515037292,
 0.1880474413221245,
 0.18012078134607865,
 0.16504132744123493,
 0.15858957127295314,
 0.1444994690882278,
 0.13256461226429694,
 0.12294586545170035,
 0.10075345767382557,

```
 0.0811185180918737,
 0.0,
 0.0,
 0.0]
```

cum_var_exp = np.cumsum(var_exp): This line calculates the cumulative sum of the explained variance for each principal component. It uses NumPy's cumsum() function to compute the cumulative sum of the var_exp list, which contains the percentage of variance explained by each principal component.

pd.DataFrame(cum_var_exp): This line creates a DataFrame from the cumulative explained variance (cum_var_exp). It converts the numpy array cum_var_exp into a DataFrame using Pandas.

In [19]:
```
# next we will find the cumulative explained variance ----------------------
cum_var_exp = np.cumsum(var_exp)
pd.DataFrame(cum_var_exp)
```

Out[19]:

	0
0	11.839838
1	21.350291
2	29.674405
3	36.079981
4	40.895467
...	...
60	99.918881
61	100.000000
62	100.000000
63	100.000000
64	100.000000

65 rows × 1 columns

Individual Explained Variance (Bar Plot):

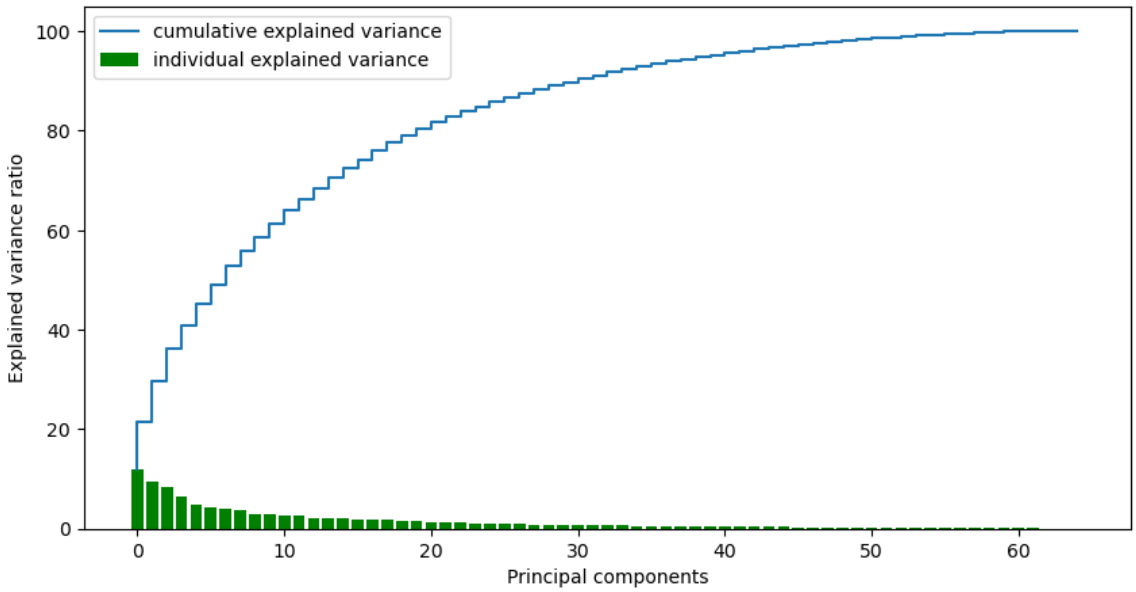
Each bar in the plot represents the percentage of variance explained by a single principal component.
The bars are sorted from left to right in decreasing order of explained variance.
This part of the plot helps you understand how much each principal component contributes to explaining the total variance in the dataset.
A higher bar indicates that the corresponding principal component captures more variance in the data.
Cumulative Explained Variance (Step Plot):

The line plot (step plot) shows the cumulative explained variance as you move from left to right along the x-axis (from the first to the last principal component).
Each point on the line represents the total cumulative explained variance up to that principal component.
This part of the plot helps you understand how much total variance in the dataset is explained by including a certain number of principal components.

Typically, you want to include principal components until you reach a point where adding more components does not significantly increase the cumulative explained variance (elbow point).
Inference:

By looking at the bar plot, you can identify the principal components that capture the most variance in the data. These are the components with the highest bars.
By examining the step plot, you can determine how many principal components are needed to capture a certain percentage of the total variance in the dataset. This helps in deciding the appropriate dimensionality reduction for your PCA.

In [20]:
```
plt.figure(figsize=(10, 5))
plt.bar(range(len(var_exp)), var_exp, label='individual explained variance'
plt.step(range(len(cum_var_exp)), cum_var_exp,label='cumulative explained va
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal components')
plt.legend()
plt.show()
```



In [21]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=
```

In [22]:
```
from sklearn.decomposition import PCA

pca = PCA(0.95)   #use n components as 95%
X_pca = pca.fit_transform(X_scaled)
X_pca.shape
```

Out[22]: (1797, 41)

In [23]:
```
pca.explained_variance_ratio_
```

Out[23]:
```
array([0.11839838, 0.09510454, 0.08324113, 0.06405576, 0.04815486,
       0.04238914, 0.03884052, 0.03734568, 0.02952017, 0.02884712,
       0.02770413, 0.02587252, 0.02238715, 0.02228271, 0.02167254,
       0.01893699, 0.01750748, 0.01735607, 0.01593396, 0.01465652,
       0.01329359, 0.01282383, 0.01162302, 0.01051612, 0.01003452,
       0.00938369, 0.00895959, 0.00836113, 0.00800431, 0.00741502,
       0.00714133, 0.0068182 , 0.00643364, 0.00631699, 0.00592526,
       0.00563391, 0.00520982, 0.00505772, 0.004715  , 0.00438909,
       0.00415992])
```

In [24]:
```
pca.n_components_
```

Out[24]:
```
41
```

In [25]:
```
pd.DataFrame(X_pca)
```

Out[25]:

	0	1	2	3	4	5	6	7	
0	1.911750	-1.057806	-3.870180	2.160589	-0.593702	1.138338	-1.132136	-1.931097	(
1	0.587193	0.636740	4.013489	-1.712615	-1.173779	-0.185333	2.076300	-0.776074	1
2	1.300136	-0.539011	3.056557	-1.983516	-2.292366	1.023472	-1.216504	-0.962085	(
3	-3.021676	-0.878534	-0.820736	-2.160010	-0.633995	0.581240	0.762927	1.181205	(
4	4.527958	-1.151432	0.942517	-1.406082	-1.787466	1.101794	0.799708	1.293557	-1
...	...	...	...	...	...	...	...	...	
1792	0.107254	0.590324	-3.841174	-1.996313	-0.034401	-0.750891	0.512613	0.156619	1
1793	2.420227	-1.584280	-2.975840	2.765666	-1.155134	0.780240	-0.681631	-2.682015	(
1794	1.023927	-0.074675	2.396417	-0.704527	-0.782070	-0.413713	0.271372	0.442339	2
1795	1.078417	-0.095411	-2.551322	-1.380247	0.412174	-0.723247	1.053729	-0.027058	1
1796	-1.256887	-2.043804	0.124162	-0.240207	-1.313452	1.093777	-1.022462	2.843482	(

1797 rows × 41 columns

Now, we can apply machine learning algorithms to train and predict the data, as dimensionality reduction has been performed

In [26]:
```
#Now again we will check the accuracy
X_train_pca, X_test_pca, y_train, y_test = train_test_split(X_pca, y, test_
```

In [27]:
```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(max_iter=1000)
model.fit(X_train_pca, y_train)
model.score(X_test_pca, y_test)
```

Out[27]:
```
0.9888888888888889
```

In [28]:
```
Y_pred_pca= model.predict(X_test_pca)
```

In [30]:
```
from sklearn.metrics import accuracy_score
accuracy_score(y_test,Y_pred_pca)
```

Out[30]: 0.9888888888888889