Exploring and Visualizing Data

Shan-Hung Wu & DataLab
Fall 2023

In [1]:
```
# inline plotting instead of popping out
%matplotlib inline

import os
import numpy as np # numpy 1.26.0
import pandas as pd # pandas 2.1.1
import matplotlib.pyplot as plt # matplotlib 3.8.0
import seaborn as sns # seaborn 0.13.0
from sklearn.preprocessing import StandardScaler # scikit-learn 1.3.1
```

Exploratory Data Analysis

Exploratory Data Analysis (EDA) is an important and recommended first step of Machine Learning (prior to the training of a machine learning model that are more commonly seen in research papers). EDA performs the **exploration** and **exploitation** steps iteratively. In the exploration step, you "explore" the data, usually by visualizing them in different ways, to discover some characteristics of data. Then, in the exploitation step, you use the identified characteristics to figure out the next things to explore. You then repeat the above two steps until you are satisfied with what you have learned from the data. Data visualization plays an important role in EDA. Next, we use the Wine dataset from the UCI machine learning repository as an example dataset and show some common and useful plots.

Visualizing the Important Characteristics of a Dataset

Let's download the Wine dataset using Pandas first:

In [2]:
```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data',
    header = None)

df.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 = df.drop(labels='Class label', axis=1)
y = df['Class label']

df.head()
```

Out[2]:

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proantl
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	

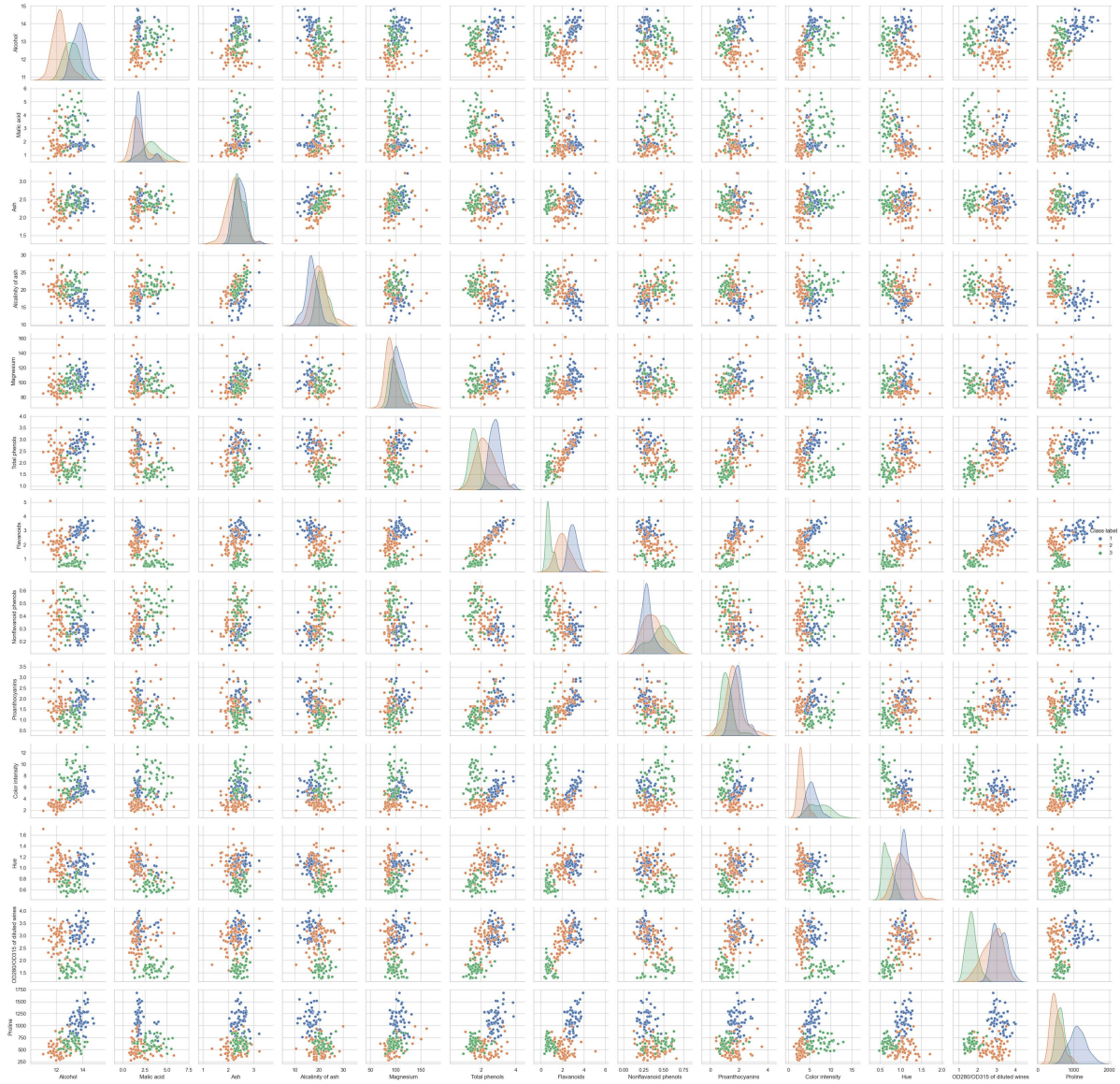
As we can see, showing data row-by-row with their column names does not help us get the "big picture" and characteristics of data.

NOTE: `pd.read_csv()` function returns a pandas.DataFrame object. Pandas Dataframe is an useful "two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes".

Pairwise Joint Distributions

We can instead see the joint distribution of any pair of columns/attributes/variables/features by using the pairplot function offered by Seaborn, which is based on Matplotlib:

In [3]:
```
sns.set(style='whitegrid', context='notebook')

sns.pairplot(df, vars=df.columns[1:], hue="Class label", height=2.5, palette=sns.co
plt.tight_layout()

if not os.path.exists('./output'):
  os.makedirs('./output')
plt.savefig('./output/fig-wine-scatter.png', dpi=300)
plt.show()
```



From the above figures, we can easily see that there is a linear relationship between the "Total phenols" and "Flavanoids" variables. Furthermore, the class label cannot be easily predicted by a single variable. You may have noticed that the figures along the diagonal look different. They are histograms of values of individual variables. We can see that the "Ash" and "Alcalinity of ash" variables are roughly normally distributed.

NOTE: importing the Seaborn library modifies the default aesthetics of Matplotlib for the current Python session. If you do not want to use Seaborn's style settings, you can reset the Matplotlib settings by calling:

```
>>> sns.reset_orig()
```

Correlation Matrix

Showing the pairwise join distributions may still be overwhelming when we have a lot of variables in the dataset. Sometimes, we can just plot the correlation matrix to quantify the linear relationship between variables. The **correlation coefficient** between two random variables a and b are defined as:

$$\frac{\mathrm{Cov}(a, b)}{\sqrt{\mathrm{Var}(a)\mathrm{Var}(b)}}.$$

Basically, it is the "normalized" variance that captures the **linear** relationship of the two random variables, and the values are bounded to the range $[-1, 1]$. The correlation matrix $\boldsymbol{R} \in \mathbb{R}^{D \times D}$ of a random vector $\mathbf{x}$ is a square matrix whose each element $R_{i,j}$ denotes the correlation between the attributes $\mathbf{x}_i$ and $\mathbf{x}_j$. If we regard data points as the i.i.d. samples of $\mathbf{x}$, then we can have an estimate $\hat{\boldsymbol{R}}$ whose each element

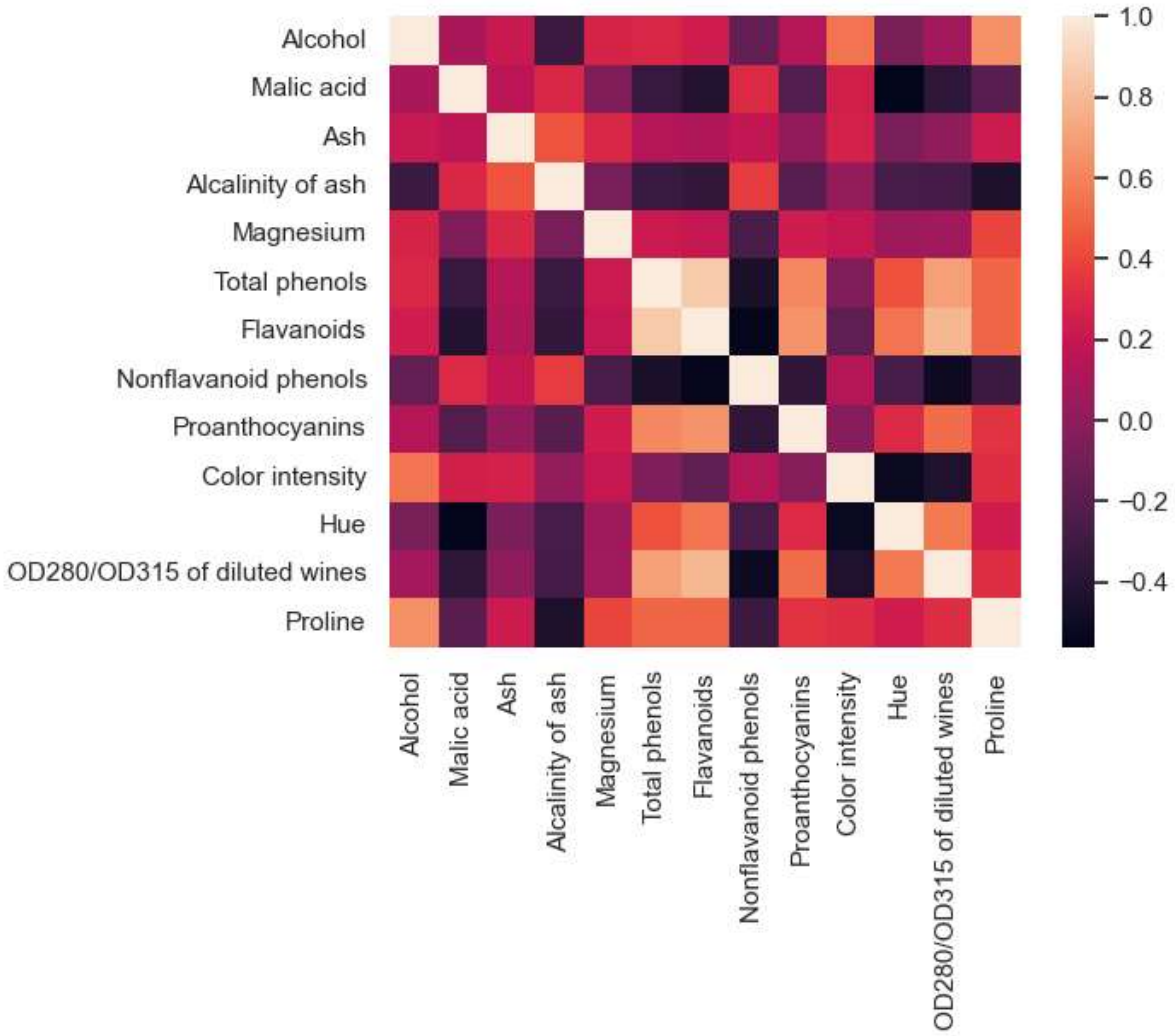
$$\hat{R}_{i,j} = \frac{\Sigma_{s=1}^{N}(x_i^{(s)} - \hat{\mu}_{\mathrm{x}_i})(x_j^{(s)} - \hat{\mu}_{\mathrm{x}_j})}{\sqrt{\Sigma_{s=1}^{N}(x_i^{(s)} - \hat{\mu}_{\mathrm{x}_i})^2}\sqrt{\Sigma_{s=1}^{N}(x_j^{(s)} - \hat{\mu}_{\mathrm{x}_j})^2}} = \frac{\hat{\sigma}_{\mathrm{x}i,\mathrm{x}j}}{\hat{\sigma}_{\mathrm{x}_i}\hat{\sigma}_{\mathrm{x}_j}}$$

is an estimate of the correlation (usually called the Pearson's r) between attribute $\mathbf{x}_i$ and $\mathbf{x}_j$. Note that if we **z-normalize** each data point such that

$$z_i^{(s)} = \frac{x_i^{(s)} - \hat{\mu}_{\mathrm{x}i}}{\hat{\sigma}_{\mathrm{x}i}}$$

for all i. Then we simply have $\hat{\boldsymbol{R}} = \frac{1}{N}\boldsymbol{Z}^\top\boldsymbol{Z}$, where $\boldsymbol{Z}$ is the design matrix of the normalized data points. We can plot $\hat{\boldsymbol{R}}$ as follows:

In [4]:
```
# Z-normalize data
sc = StandardScaler()
Z = sc.fit_transform(X)
# Estimate the correlation matrix
R = np.dot(Z.T, Z) / df.shape[0]

sns.set(font_scale=1.0)

ticklabels = [s for s in X.columns]

hm = sns.heatmap(
    R,
    cbar=True,
    square=True,
    yticklabels=ticklabels,
    xticklabels=ticklabels
)

plt.figure(figsize=(10, 8))
plt.tight_layout()
plt.savefig('./output/fig-wine-corr.png', dpi = 300)
plt.show()

sns.reset_orig()
```




```
<Figure size 1000x800 with 0 Axes>
```

The correlation matrix gives a more concise view of the relationship between variables. Some models, such the linear regression, assume that the explanatory variables are linearly correlated to the target variable. A heatmap of correlations can help us select variables supporting this assumption. For example, if we want to predict the "Hue" variable using the linear regression, we may pick the "Malic acid" and "Flavanoids" as the explanatory variables.

NOTE: we could have simply used the NumPy function

```
>>>R = np.corrcoef(df.values.T)
```

to get the estimate $\hat{\boldsymbol{R}}$ of the correlation matrix. We calculate $\hat{\boldsymbol{R}}$ by ourselves using the normalized design matrix $\boldsymbol{Z}$ because we will reuse $\boldsymbol{Z}$ later.

PCA for Visualization

PCA reduces the dimension of data points, and has been widely used across different machine learning tasks. One application of PCA is to help the visualization of high-dimensional data, as discussed next.

Principal component analysis finds a projection matrix $\boldsymbol{W} = [w^{(1)}, \ldots, w^{(k)}] \in \mathbb{R}^{D \times K}$,where w^i are orthonormal vectors, such that each attribute $z_j^{pca} = w^{(j)T} z$ has the

maximum variance Var(z_j^{pca}).

This problem can be reduced to solve

$$arg \max_{w^i \in \mathbb{R}^D} w^{(i)T} Z^T Z w^{(i)}, \ for\ i \in [1,\ K]$$

by Rayleigh's Quotient, the optimal $w^{(i)}$ is given by the eigenvector of $Z^T Z$ (or $\hat{\boldsymbol{R}}$) corresponding to the ith largest eigenvalue.

Let's summarize PCA in a few simple steps:

1. Standardize the D -dimensional dataset $\boldsymbol{X}$, e.g., via the z -normalization, and get $\boldsymbol{Z}$;
2. Estimate the covariance matrix $\hat{\boldsymbol{R}}$;
3. Decompose $\hat{\boldsymbol{R}}$ into its eigenvectors and eigenvalues;
4. Select K eigenvectors that correspond to the K largest eigenvalues, where K is the dimensionality of the new feature subspace ($k < d$);
5. Construct a projection matrix $\boldsymbol{W}$ from the top-K eigenvectors;
6. Transform the D-dimensional input dataset $\boldsymbol{Z}$ using the projection matrix $\boldsymbol{W}$.

Eigendecomposition

Since we already have $\boldsymbol{Z}$ and $\hat{\boldsymbol{R}}$ from the above. We can begin from the step 3:

In [5]:
```
eigen_vals, eigen_vecs = np.linalg.eigh(R)

print('\nEigenvalues: \n%s' % eigen_vals)
```

```
Eigenvalues:
[0.10337794 0.16877023 0.22578864 0.25090248 0.28887994 0.34849736
 0.55102831 0.64165703 0.85322818 0.91897392 1.44607197 2.49697373
 4.70585025]
```

NOTE: there is an `np.linalg.eig()` function in NumPy that also eigendecomposes matrices. The difference is that `np.linalg.eigh()` is optimized for symmetric matrices whose eigenvalues are always real numbers. The numerically less stable `np.linalg.eig()` can decompose non-symmetric square matrices and returns complex eigenvalues.

Eigenvector Selection

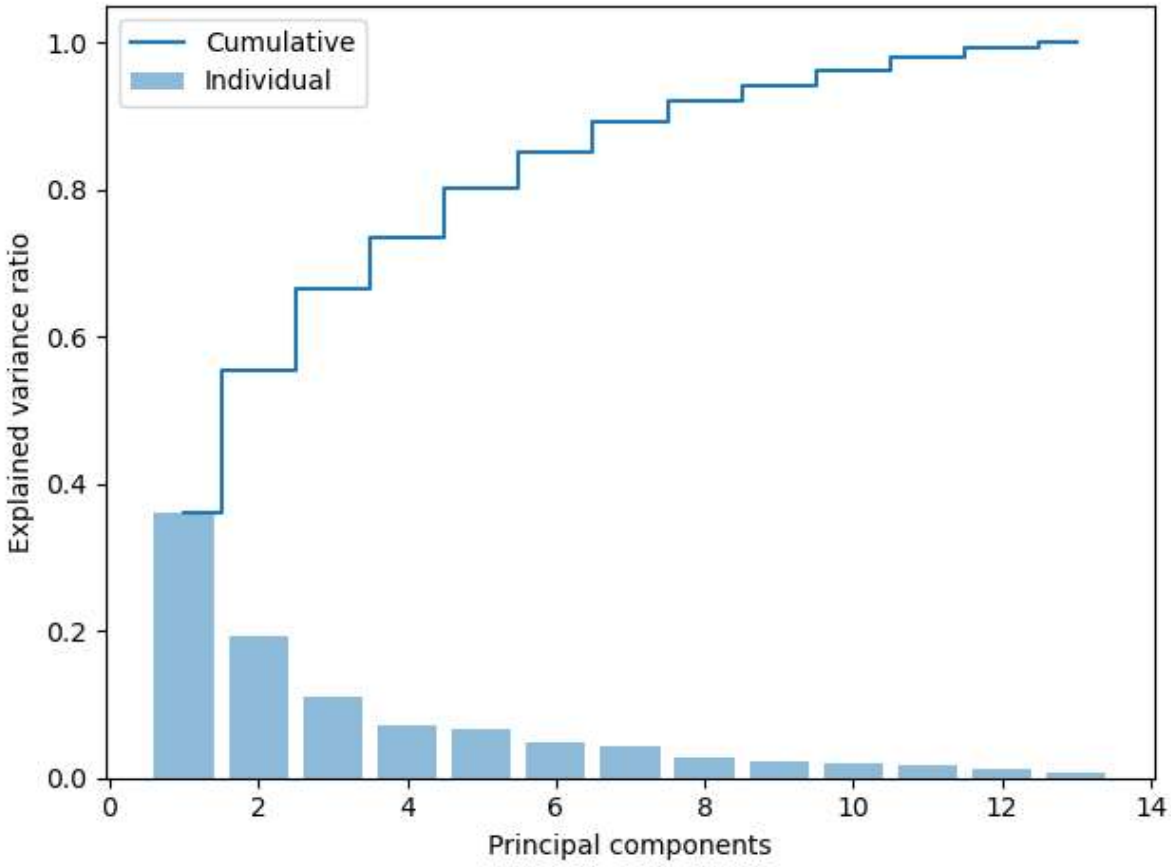
In step 4, we need to decide the value of K. We can plot the **variance explained ratio** of each eigenvalue:

$$\frac{|\lambda_j|}{\Sigma_{j=1}^{D} |\lambda_j|}$$

in the descending order to help us decide how many eigenvectors to keep.

In [6]:
```
tot = sum(np.abs(eigen_vals))
var_exp = [(i / tot) for i in sorted(np.abs(eigen_vals), reverse=True)]
cum_var_exp = np.cumsum(var_exp)

plt.bar(range(1, eigen_vals.size + 1), var_exp, alpha=0.5, align='center',
        label='Individual')
plt.step(range(1, eigen_vals.size + 1), cum_var_exp, where='mid',
```

```
         label='Cumulative')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal components')
plt.legend(loc='best')
plt.tight_layout()
plt.savefig('./output/fig-pca-var-exp.png', dpi=300)
plt.show()
```



The resulting plot indicates that the first principal component alone accounts for 40 percent of the variance. Also, we can see that the first two principal components combined explain almost 60 percent of the variance in the data. Next, we collect the two eigenvectors that correspond to the two largest values to capture about 60 percent of the variance in this dataset. Note that we only chose two eigenvectors for the purpose of illustration, since we are going to plot the data via a two-dimensional scatter plot later in this subsection. In practice, the number of principal components can be determined by other reasons, such as the trade-off between computational efficiency and performance.

Feature Transformation

Let's now proceed with the last three steps to project the standardized Wine dataset onto the new principal component axes. We start by sorting the eigenpairs by decreasing order of the eigenvalues:

In [7]:
```
# Make a list of (eigenvalue, eigenvector) tuples
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i]) for i in range(len(eigen_va

# Sort the (eigenvalue, eigenvector) tuples from high to low
eigen_pairs.sort(reverse=True)
```

Next, we pick the first two eigenvectors and form the project matrix $\boldsymbol{W}$:

In [8]:
```
W = np.hstack((eigen_pairs[0][1][:, np.newaxis],
               eigen_pairs[1][1][:, np.newaxis]))
print('Projection matrix W:\n', W)
```

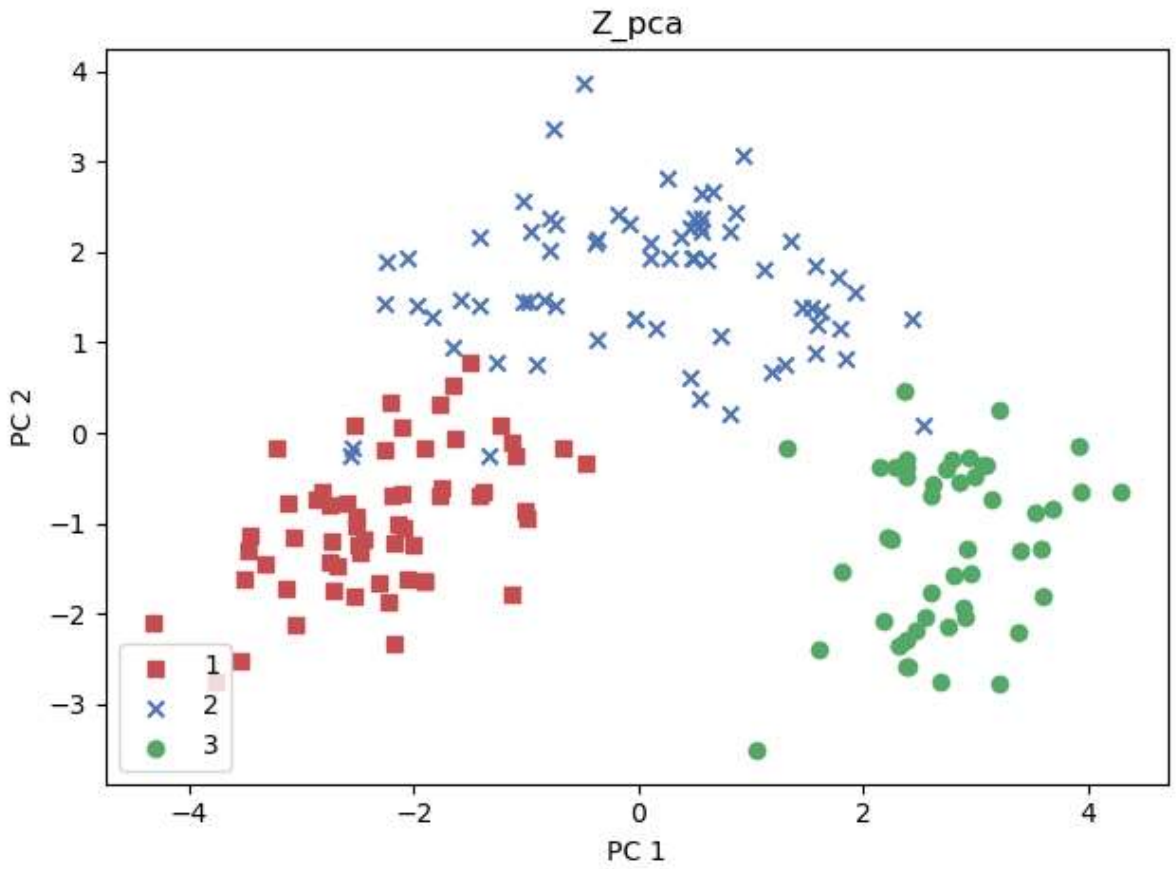
```
Projection matrix W:
 [[-0.1443294  -0.48365155]
 [ 0.24518758 -0.22493093]
 [ 0.00205106 -0.31606881]
 [ 0.23932041  0.0105905 ]
 [-0.14199204 -0.299634  ]
 [-0.39466085 -0.06503951]
 [-0.4229343   0.00335981]
 [ 0.2985331  -0.02877949]
 [-0.31342949 -0.03930172]
 [ 0.0886167  -0.52999567]
 [-0.29671456  0.27923515]
 [-0.37616741  0.16449619]
 [-0.28675223 -0.36490283]]
```

Finally, we can obtain the compressed dataset by:

$$\boldsymbol{Z}^{\text{PCA}} = \boldsymbol{Z}\boldsymbol{W}$$

and visualize it using:

In [9]:
```
Z_pca = Z.dot(W)

colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']
for l, c, m in zip(np.unique(y.values), colors, markers):
    plt.scatter(Z_pca[y.values==l, 0],
                Z_pca[y.values==l, 1],
                c=c, label=l, marker=m)

plt.title('Z_pca')
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.savefig('./output/fig-pca-z.png', dpi=300)
plt.show()
```



As we can see, the data is more spread along the x-axis corresponding to the first principal component than the y-axis (second principal component), which is consistent with the explained variance ratio plot that we created in the previous subsection. PCA may save us from examining a lot of pairwise distributions (as shown in the previous section) when the original data dimension D is high. For example, if we see that the data points with different labels can be separated in the space of PCA, then we can simply choose a linear classifier to do the classification.

Finally, let's save the compressed dataset for future use.

In [10]:
```
np.save('./output/Z_pca.npy', Z_pca)
```

Assignment

Here's a generated dataset, with 3 classes and 15 attributes. Your goal is to reduce data dimension to 2 and 3, and then plot 2-D and 3-D visualization on the compressed data, respectively.

In [11]:
```
"""
# import libs, load data
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
"""

df_load = pd.read_csv('https://nthu-datalab.github.io/ml/labs/02_EDA_PCA/gen_datase

X_load = df_load.drop(labels='Class label', axis=1)
```

```
Y_load = df_load['Class label']

df_load.head()
```

Out[11]:

	Class label	a1	a2	a3	a4	a5	a6	a7	a8	
0	2.0	-0.016488	-1.310538	-1.552489	-0.785475	1.548429	0.476687	1.090010	-0.351870	-0.0(
1	0.0	-0.844201	-1.235142	-0.624408	1.502470	-0.079536	1.482053	1.178544	-1.150090	-1.0
2	0.0	-0.181053	0.039422	-0.307827	0.162256	-1.283705	0.541288	0.019113	-0.470718	-1.0
3	2.0	-0.423555	-1.598754	1.597206	-0.239330	1.443564	2.657538	1.824393	-1.809287	1.0
4	2.0	-0.499408	-0.814229	-0.178777	-1.757823	0.678134	3.552825	1.483069	-2.341943	2.1

In [12]:

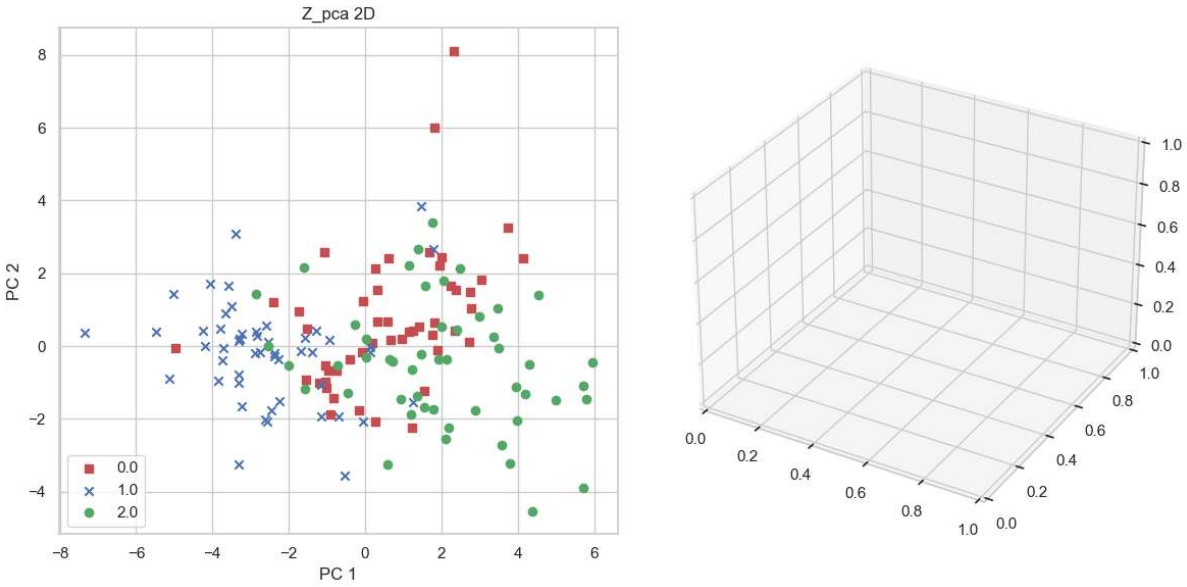
```
import numpy as np
from sklearn.preprocessing import StandardScaler

# Z-normalize data
sc = StandardScaler()
Z = sc.fit_transform(X_load)
# Estimate the correlation matrix
R = np.dot(Z.T, Z) / df_load.shape[0]

# Calculate the eigen values, eigen vectors
eigen_vals, eigen_vecs = np.linalg.eigh(R)

# Make a list of (eigenvalue, eigenvector) tuples
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i]) for i in range(len(eigen_va

# Sort the (eigenvalue, eigenvector) tuples from high to low
eigen_pairs.sort(reverse=True)

# Form the projection matrix
W_2D = np.hstack((eigen_pairs[0][1][:, np.newaxis],
                  eigen_pairs[1][1][:, np.newaxis]))

# You should form a projection matrix which projects from raw-data dimension to 3 d
```

You can see here for information about plotting 3D graph

In [13]:

```
import os
import seaborn as sns
sns.set(style='whitegrid', context='notebook')

# import Axes3D for plottin 3d scatter
from mpl_toolkits.mplot3d import Axes3D

# cacculate z_pca(2d and 3d)
Z_pca2 = Z.dot(W_2D)

# plot settings
colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']
fig = plt.figure(figsize=(12,6))

# plot 2D
plt2 = fig.add_subplot(1,2,1)
for l, c, m in zip(np.unique(Y_load), colors, markers):
    plt2.scatter(Z_pca2[Y_load==l, 0],
```

```
                Z_pca2[Y_load==l, 1],
                c=c, label=l, marker=m)

plt.title('Z_pca 2D')
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()

# plot 3D
plt3 = fig.add_subplot(1,2,2, projection='3d')
# you should plot a 3D scatter using plt3.scatter here (see Axes3D.scatter in matpl

if not os.path.exists('./output'):
    os.makedirs('./output')
plt.savefig('./output/fig-pca-2-3-z.png', dpi=300)
plt.show()
```



From this assignment, you can see the different results between different numbers of principle components chosen.

Requirements:

- Submit to **eeclass** with your code named `Lab02_{student-id}.ipynb` (e.g. `Lab02_109069999.ipynb`).
- **The code file should only contain the Assignment part.**
- Remember to save the file after you rendered the output images in your notebook.
- Deadline: **2024-01-07 (Sun) 23:59**.