

THE NEWEST RELEASE FROM GOLD BEAR PUBLISHING

# 深度 機器學習

MASTERING DEEP LEARNING:  
REVOLUTIONIZING THE FUTURE

張智硯 BEARINGTON

# 主成分分析策略：發現數據的真實故事

---

## 專案前言：

在本次專案中，我們將進行資料探索和主成分分析的任務，分為兩個部分。首先，我們將探索一組關於紅酒的化學成分數據，以及一組有關乳癌患者腫瘤的影像量測資料。這兩個數據集都有其獨特的特徵和挑戰，我們將通過相關性分析、散布圖、主成分分析等方法來深入研究它們。

## 專案目標簡介：

本專案旨在進行資料探索和主成分分析 (PCA) 的兩個部分，分別針對以下兩個數據集：

### 1. 紅酒化學成分數據集

- 繪製相關性圖，檢查變數之間的相關性。
- 繪製盒鬚圖，評估變數的範圍，決定是否需要標準化。
- 執行PCA，繪製特徵值分布圖和Scree Plot，確定主成分數量。
- 繪製紅酒數據的前兩個主成分的散布圖，並根據標籤著色。

### 2. 乳癌患者腫瘤影像量測資料集

- 進行相關性分析，適當視覺化多個變數之間的相關性。
- 選擇適當的數據縮減方法，以簡化複雜的數據集。

通過達成這些目標，我們將深入探索兩個不同的數據集，並利用主成分分析等技術來提取有價值的資訊，以進一步了解數據的結構和關聯性。

---

**資料集（一）：**有一組資料來自義大利某個地區三個紅酒製造商所產的紅酒，資料內容包括的 178 支紅酒的 13 種化學成分。利用這組資料回答下列問題：

導入初步套件

```
In [ ]: import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import numpy as np
from numpy.linalg import svd
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from mpl_toolkits.mplot3d import Axes3D
```

## 資料讀入

```
In [ ]: data = pd.read_excel('/content/drive/MyDrive/淺度機器學習/wine.xlsx')
labels = data['Customer_Segment']
data = data.drop("Customer_Segment", axis = 1)
data.head()
```

```
Out[ ]:   Alcohol  Malic_Acid  Ash  Ash_Alcanity  Magnesium  Total_Phenols  Flavanoids  Nonflava
0      14.23       1.71    2.43          15.6        127         2.80      3.06
1      13.20       1.78    2.14          11.2        100         2.65      2.76
2      13.16       2.36    2.67          18.6        101         2.80      3.24
3      14.37       1.95    2.50          16.8        113         3.85      3.49
4      13.24       2.59    2.87          21.0        118         2.80      2.69
```

```
In [ ]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 178 entries, 0 to 177
Data columns (total 13 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Alcohol          178 non-null    float64
 1   Malic_Acid       178 non-null    float64
 2   Ash               178 non-null    float64
 3   Ash_Alcanity     178 non-null    float64
 4   Magnesium         178 non-null    int64  
 5   Total_Phenols    178 non-null    float64
 6   Flavanoids        178 non-null    float64
 7   Nonflavanoid_Phenols  178 non-null  float64
 8   Proanthocyanins  178 non-null    float64
 9   Color_Intensity  178 non-null    float64
 10  Hue              178 non-null    float64
 11  OD280            178 non-null    float64
 12  Proline           178 non-null    int64  
dtypes: float64(11), int64(2)
memory usage: 18.2 KB
```

```
In [ ]: fontparams = {'font.size': 13, 'font.weight':'bold',
                     'font.family':'arial', 'font.style':'italic'}
plt.rcParams.update(fontparams)
labelparams = {'size': 14, 'weight':'semibold',
               'family':'serif', 'style':'italic'}
```

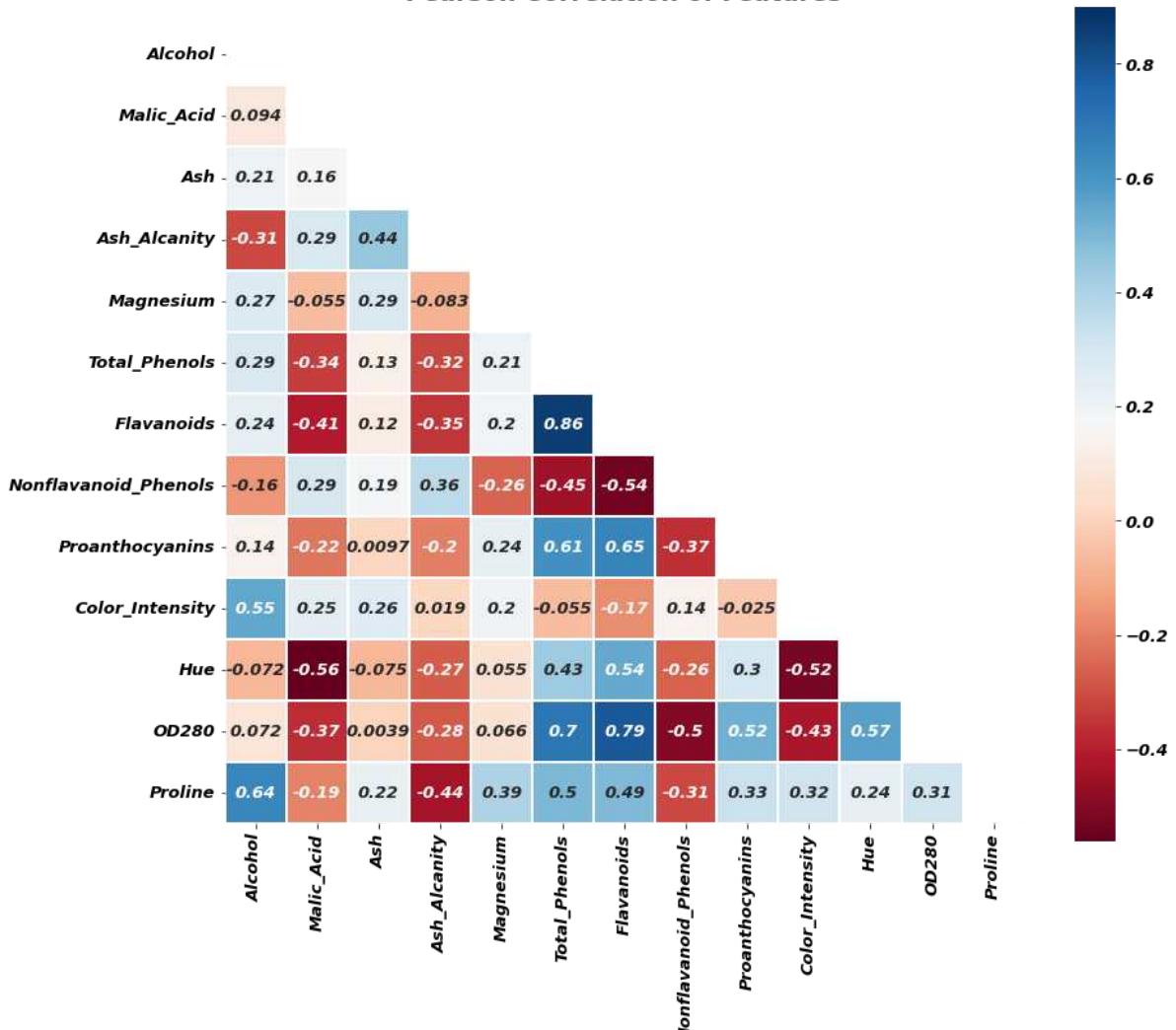
## 1. 繪製變數間的相關係數圖，以觀察變數間是否存在相關性

```
In [ ]: colormap = plt.cm.RdBu
plt.figure(figsize=(14,12))
plt.title('Pearson Correlation of Features', y=1.02, size=19, fontweight='bold')

corr_matrix = data.astype(float).corr() #相關係數矩陣
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))
sns.heatmap(corr_matrix, linewidths=0.1, vmax=0.9,
            square=True, mask=mask, cmap=colormap, linecolor='white', annot=True)
```

```
Out[ ]: <AxesSubplot:title={'center':'Pearson Correlation of Features'}>
```

### Pearson Correlation of Features



以下是觀察自變數間相關性的結論：

- 多數自變數之間存在相關性，且部分相關性很強。
- 最強的相關性為 total\_phenols 和 flavanoids 之間，相關係數為 0.86，其次是 flavanoids 和 proanthocyanins 之間，相關係數為 0.75。
- 酒精濃度 (Alcohol) 和花青素類化合物 (Flavanoids) 、類黃酮化合物 (Proanthocyanins) 、總酚 (Total\_phenols) 之間存在中等程度的相關性，相關係數分別為 0.55、0.55、0.51。

- 繪製一張含每個化學成分（變數）的盒鬚圖（Boxplot），觀察每個變數的 scaling，作為是否標準化的參考

In [ ]: #老師講義的方法繪製Boxplot

```
plt.figure(figsize = (10, 8))
plt.rcParams['font.family'] = ['serif']

ratings = np.array(data)
categories = data.columns

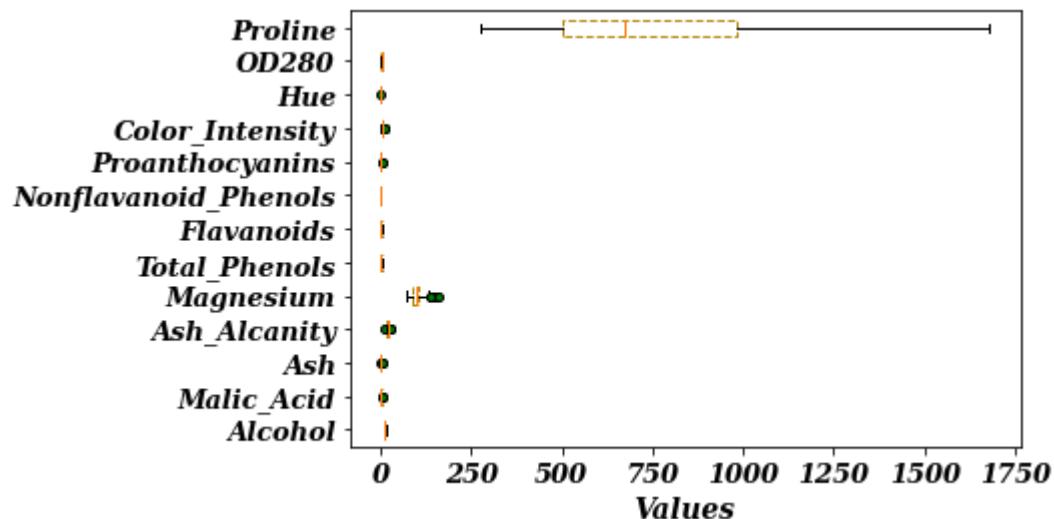
fig, ax = plt.subplots()
boxprops = dict(linestyle = '--', linewidth = 1,
color = 'darkgoldenrod')
flierprops = dict(marker='o', markerfacecolor = 'green',
```

```

    markersize = 4, linestyle = 'none')
ax.boxplot(ratings, boxprops = boxprops, \
flierprops = flierprops, \
labels = categories, vert = False)
ax.set_xlabel('Values', labelparams)
plt.show()

```

<Figure size 720x576 with 0 Axes>

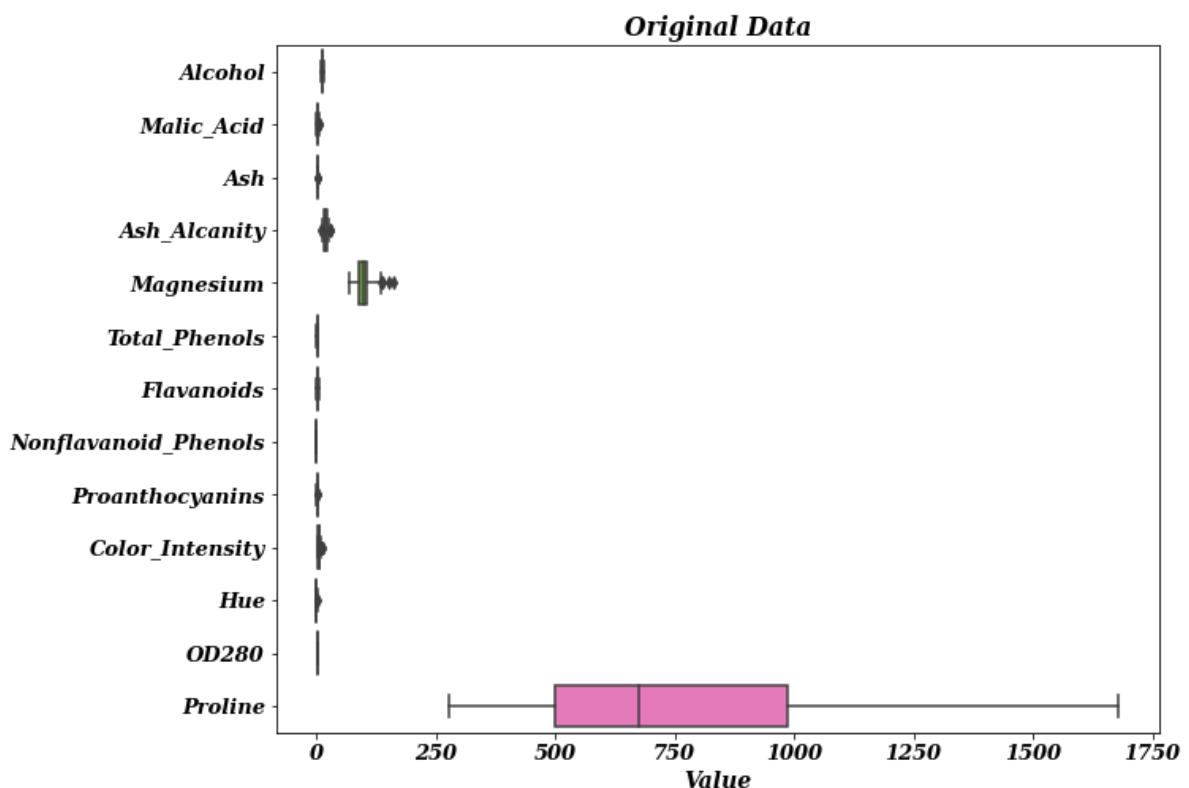


In [ ]: #使用sns繪製Boxplot, 變數排列方式為水平

```

import seaborn as sns
import pandas as pd
plt.figure(figsize = (10, 8))
ax = sns.boxplot(data = data, palette = 'husl', orient = 'h')
ax.set_title('Original Data', fontweight = 'bold')
ax.set_xlabel('Value', labelparams)
plt.show()

```



In [ ]: #使用sns繪製Boxplot, 變數排列方式為垂直

```

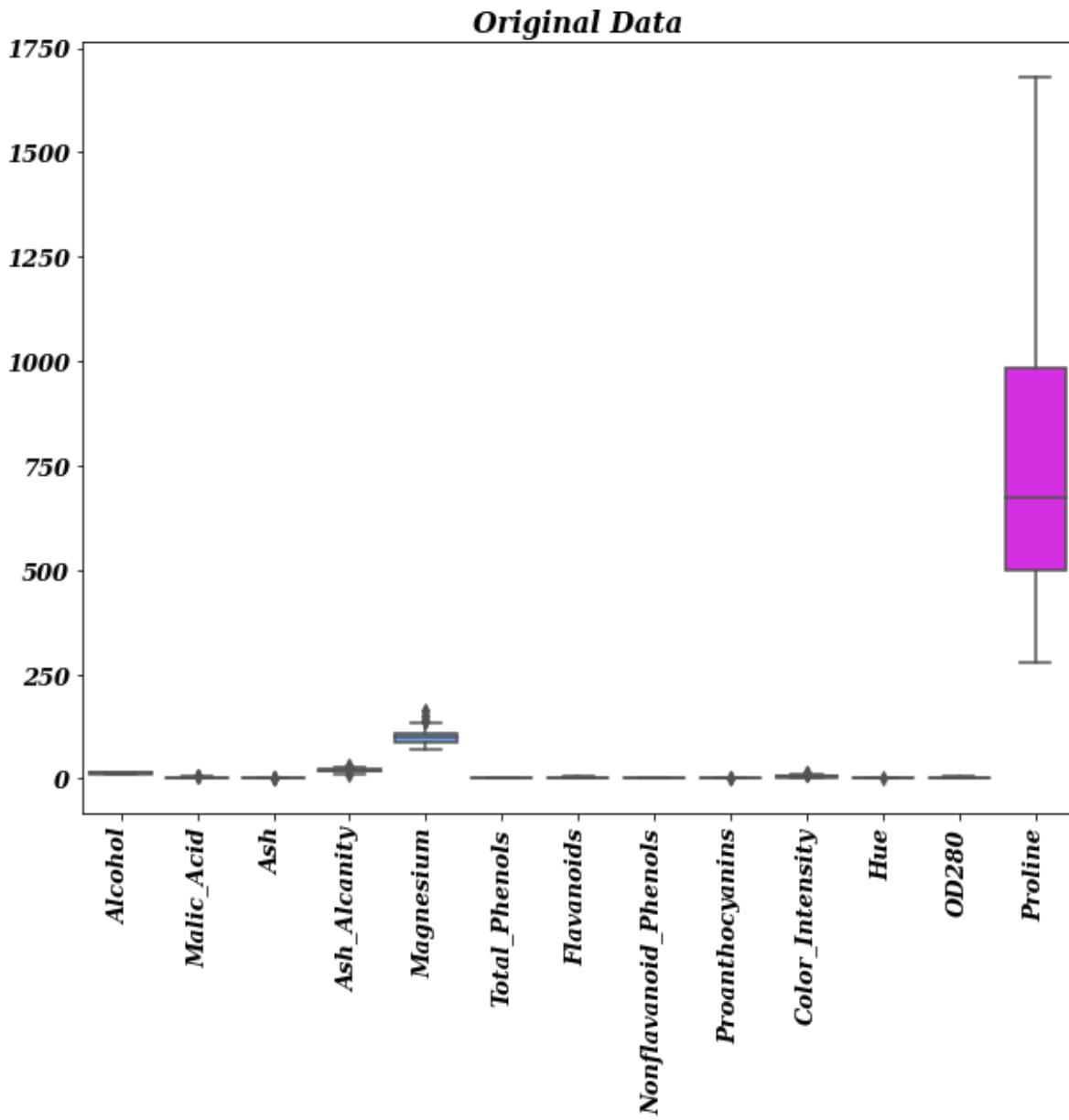
import seaborn as sns

```

```

import pandas as pd
plt.figure(figsize = (10, 8))
ax = sns.boxplot(data = data, palette = 'cool')
ax.set_title('Original Data', fontweight = 'bold' )
plt.xticks(rotation = 90, ha = 'center')
plt.show()

```



因為數據集中的變數'Proline'與其他變數的數據散佈差異程度大，造成箱型圖的可比較能力下降。故對數據集進行標準化，將變數放在相同的比例上觀察。

```

In [ ]: #使用sns繪製標準化Boxplot, 變數排列方式為水平

# 初始化 StandardScaler 對象
scaler = StandardScaler()

# 應用標準化到數據上
data_scaled = scaler.fit_transform(data)

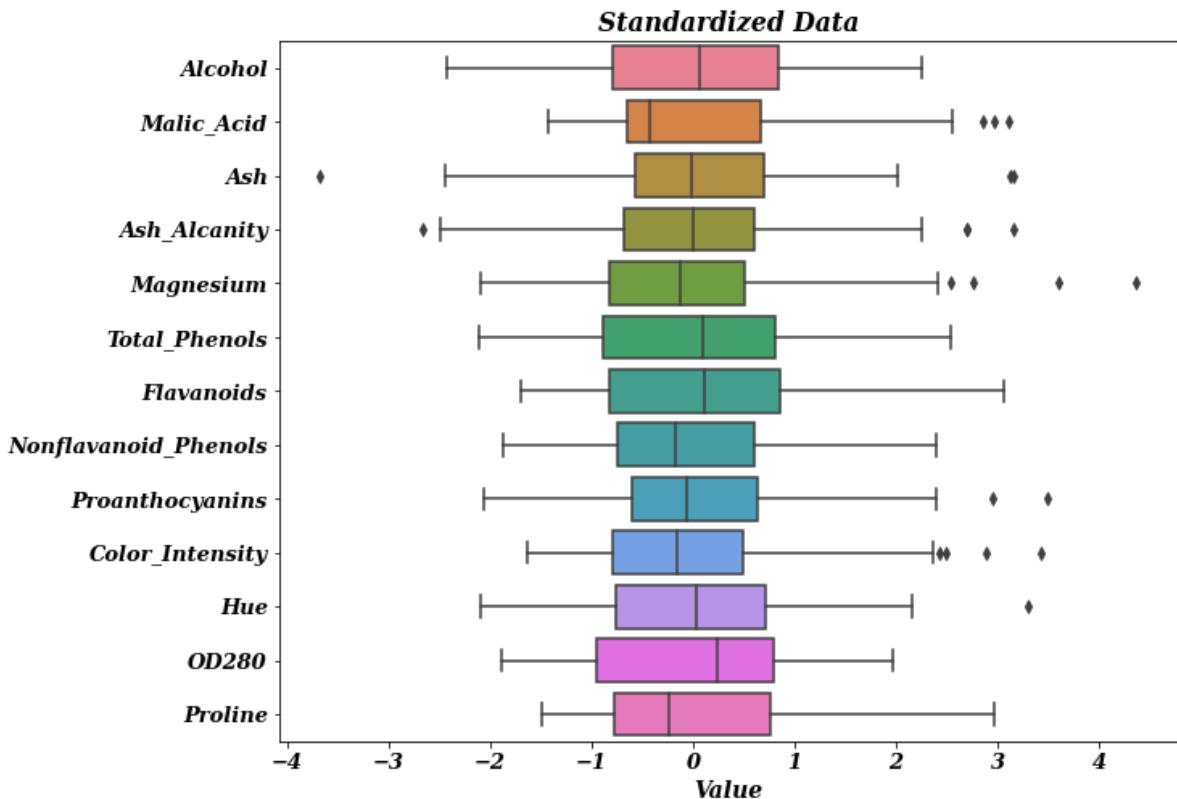
# 將標準化後的數據轉換成 Pandas DataFrame
data_scaled = pd.DataFrame(data_scaled, columns=data.columns)

# 繪製標準化後的Boxplot
plt.figure(figsize=(10, 8))
ax = sns.boxplot(data=data_scaled, palette='husl', orient='h')

```

```
ax.set_title('Standardized Data', fontweight='bold')
ax.set_xlabel('Value', labelparams)
```

Out[ ]: Text(0.5, 0, 'Value')



標準化後的資料所呈現的箱型圖能夠提供更好的比較能力，相較於標準化前的資料呈現方式。

3. 進行主成分分析，繪製特徵值由大而小的分布與 scree plot。

```
In [ ]: #創建pca_raw對象，不進行標準化
pca_raw = PCA()

# 擬合數據並進行轉換
pca_raw.fit(data)
transformed_raw = pca_raw.transform(data)

# 計算特徵值和特徵向量
eigenvalues = pca_raw.explained_variance_
eigenvectors = pca_raw.components_

# 繪製特徵值分佈圖和scree plot
fig, ax1 = plt.subplots(figsize=(12, 6))

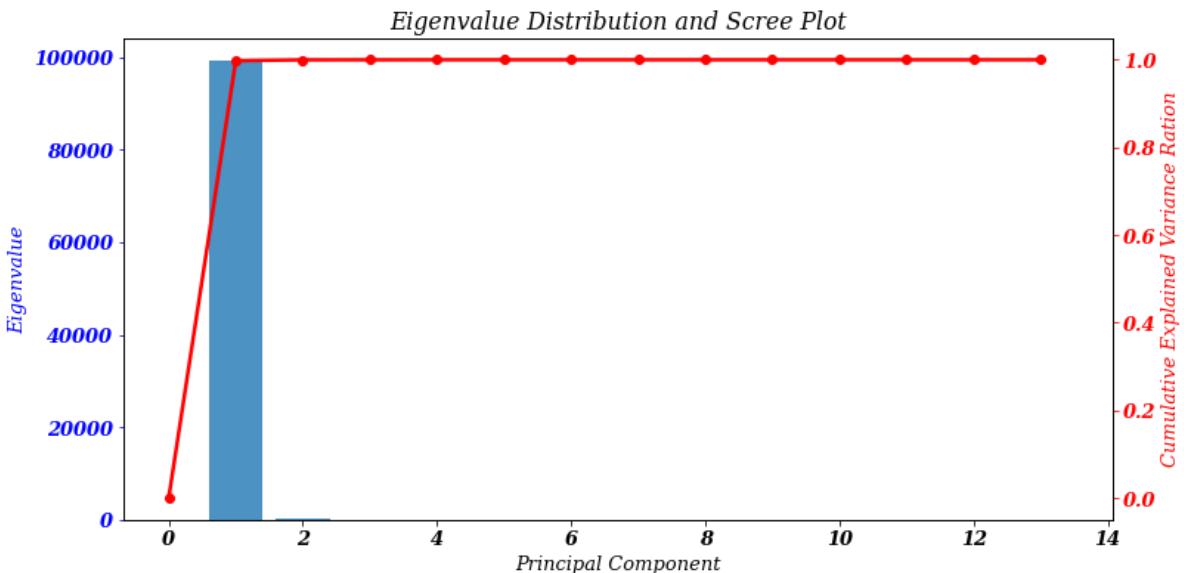
# 繪製特徵值條形圖
ax1.bar(range(1, len(eigenvalues)+1), eigenvalues, alpha=0.8)
ax1.set_xlabel('Principal Component')
ax1.set_ylabel('Eigenvalue', color='b')
ax1.tick_params('y', colors='b')
ax1.set_title('Eigenvalue Distribution and Scree Plot')

# 繪製累積解釋方差曲線
ax2 = ax1.twinx()
ax2.plot(np.insert(np.cumsum(pca_raw.explained_variance_ratio_), 0, 0), 'r')
ax2.set_ylabel('Cumulative Explained Variance Ration', color='r')
```

```

ax2.tick_params('y', colors='r')
plt.show()

```



根據主成份分析的特徵值分佈圖和累積解釋變異曲線，可以觀察到未經過標準化的數據在進行主成份分析後，第一主成份的特徵值非常大，幾乎解釋了所有的變異。這表明數據集中的“Proline”變量比其他變量的值大得多，導致特徵值之間的比較變得不準確。因此，可以推斷出需要對數據進行標準化，以避免單個變量對主成份分析結果的影響過大。

```

In [ ]: # 創建PCA對象，進行標準化
pca_normalized = PCA()
scaler = StandardScaler()

# 對數據進行標準化和擬合
data_standardized = scaler.fit_transform(data)
pca_normalized.fit(data_standardized)
transformed_normalized = pca_normalized.transform(data_standardized)

# 計算特徵值和特徵向量
eigenvalues = pca_normalized.explained_variance_
eigenvectors = pca_normalized.components_

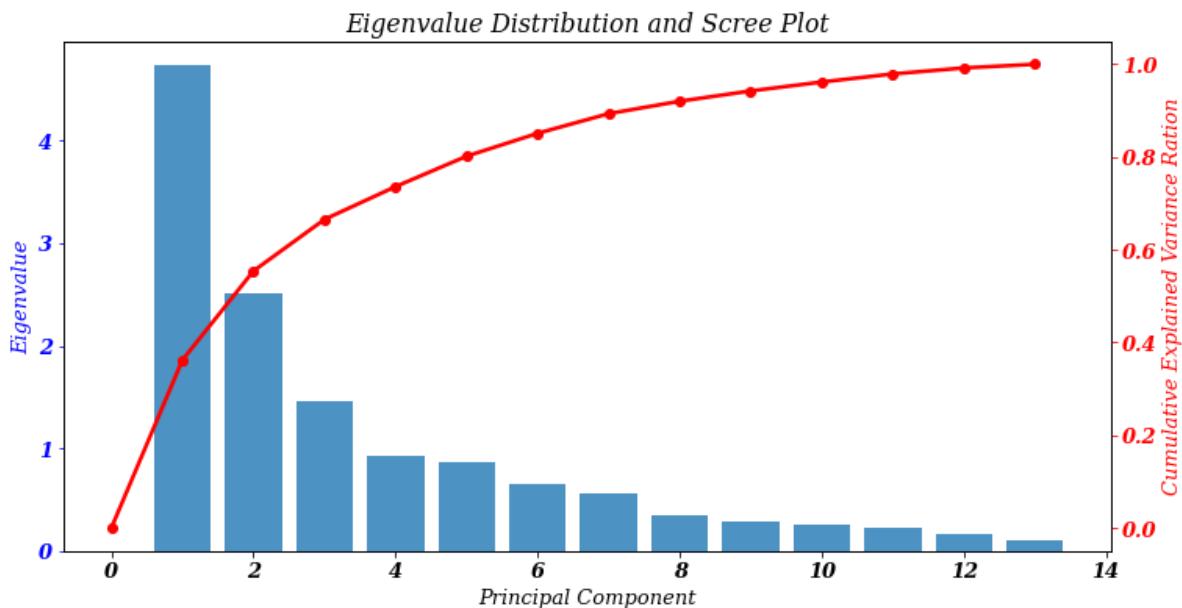
# 繪製特徵值分佈圖和scree plot
fig, ax1 = plt.subplots(figsize=(12, 6))

# 繪製特徵值條形圖
ax1.bar(range(1, len(eigenvalues)+1), eigenvalues, alpha=0.8)
ax1.set_xlabel('Principal Component')
ax1.set_ylabel('Eigenvalue', color='b')
ax1.tick_params('y', colors='b')
ax1.set_title('Eigenvalue Distribution and Scree Plot')

# 繪製累積解釋方差曲線
ax2 = ax1.twinx()
ax2.plot(np.insert(np.cumsum(pca_normalized.explained_variance_ratio_), 0, 0))
ax2.set_ylabel('Cumulative Explained Variance Ration', color='r')
ax2.tick_params('y', colors='r')

plt.show()

```



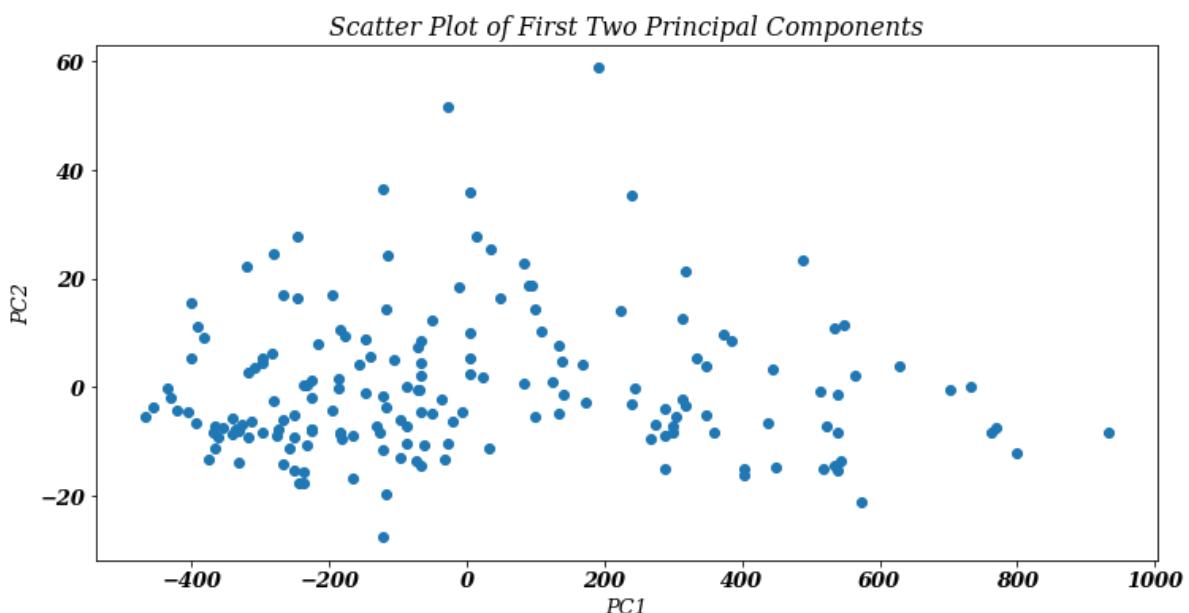
根據主成份分析的特徵值分佈圖和累積解釋變異曲線，經過數據標準化後重新進行主成份分析，可以看到各個特徵值之間的比較更加準確。不再存在像未標準化數據集中出現的單個變量“Proline”對主成份分析結果影響過大的情況。因此，通過數據標準化可以使得主成份分析結果更加可靠和穩定。

---

1. 利用主成分分析取得前兩項成分，並繪製其散佈圖。

```
In [ ]: pc1 = transformed_raw[:, 0]
          pc2 = transformed_raw[:, 1]

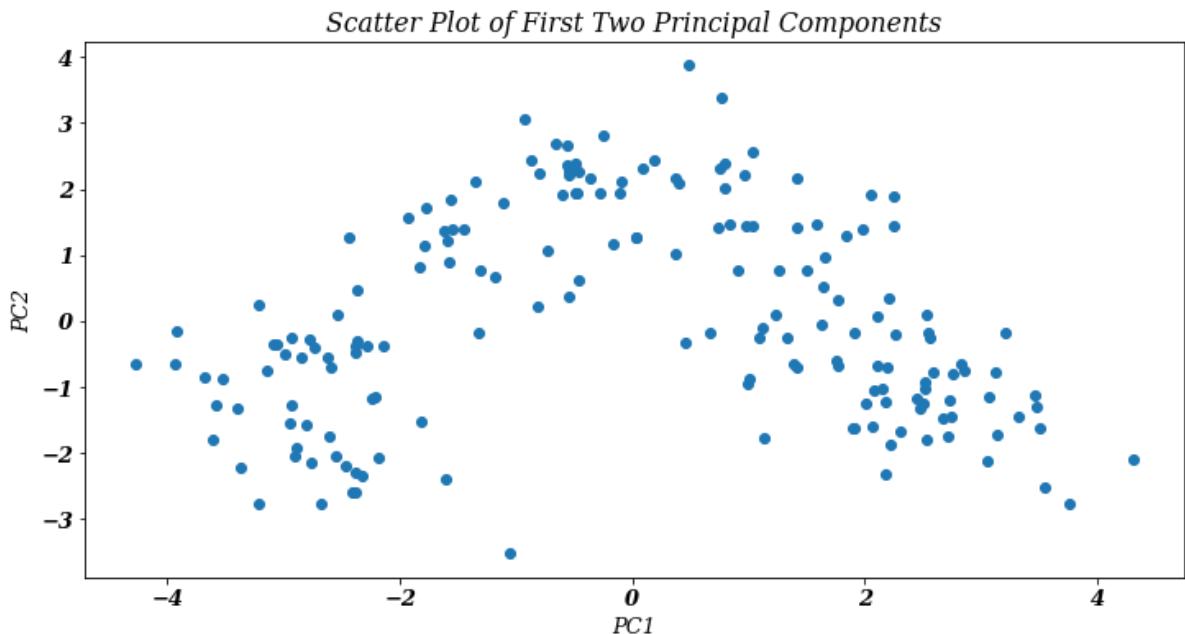
          plt.figure(figsize=(12, 6))
          plt.scatter(pc1, pc2)
          plt.xlabel('PC1')
          plt.ylabel('PC2')
          plt.title('Scatter Plot of First Two Principal Components')
          plt.show()
```



在標準化前的主成分散布圖中看不出三個群組，這可能是因為數據中存在不同變量之間的數值差異，導致了不同變量之間的權重不均衡。

```
In [ ]: pc1 = transformed_normalized[:, 0]
pc2 = transformed_normalized[:, 1]

plt.figure(figsize=(12, 6))
plt.scatter(pc1, pc2)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('Scatter Plot of First Two Principal Components')
plt.show()
```



在標準化後的主成分散布圖中能夠看出三個群組，原因可能為標準化後的主成分分析使得數據更平等，更能準確反映每個變量對群組的貢獻。

---

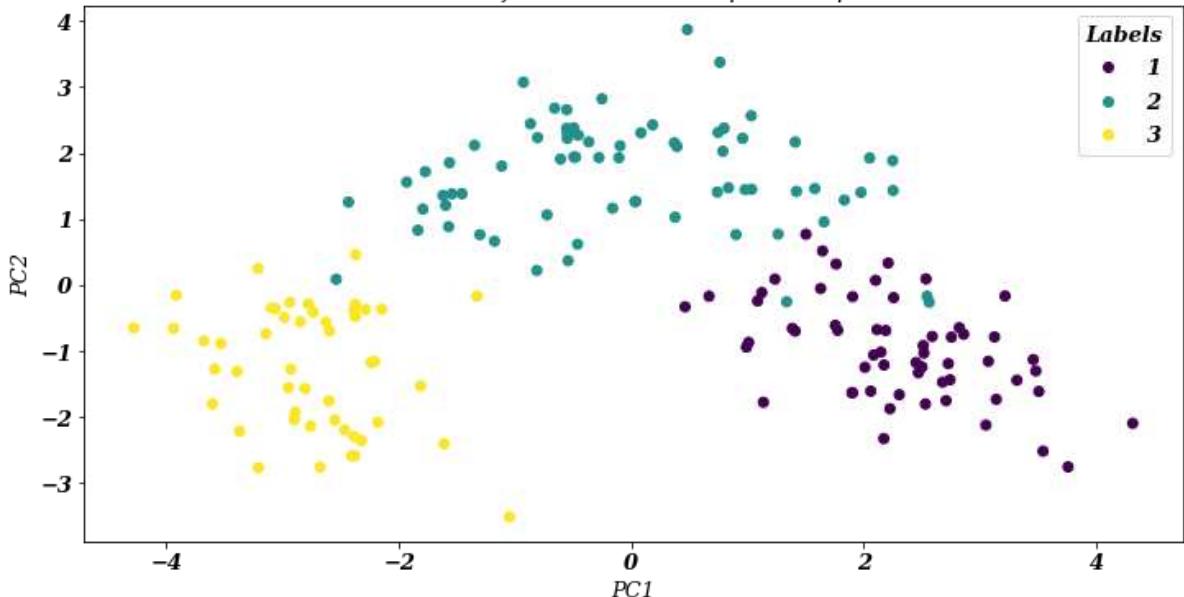
1. 再依據每個資料的標籤，為每個在散布圖上的資料點塗上顏色

```
In [ ]: import matplotlib.pyplot as plt

pc1 = transformed_normalized[:, 0]
pc2 = transformed_normalized[:, 1]

plt.figure(figsize=(12, 6))
scatter = plt.scatter(pc1, pc2, c=labels)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('Scatter Plot of First Two Principal Components')
plt.legend(*scatter.legend_elements(), title="Labels")
plt.show()
```

Scatter Plot of First Two Principal Components



在加入labels之後，可以發現前兩個主成分已經可以很好地將數據分成不同的類別，只有第一類和第二類有些微交錯。這表明前兩個主成分已經捕捉到了數據中大部分大部分的差異。

---

- 採取前三個主成分，並繪製三維散點圖，觀察其在群組分辨能力上是否比前兩個主成分更優秀。

```
In [ ]: import plotly.express as px

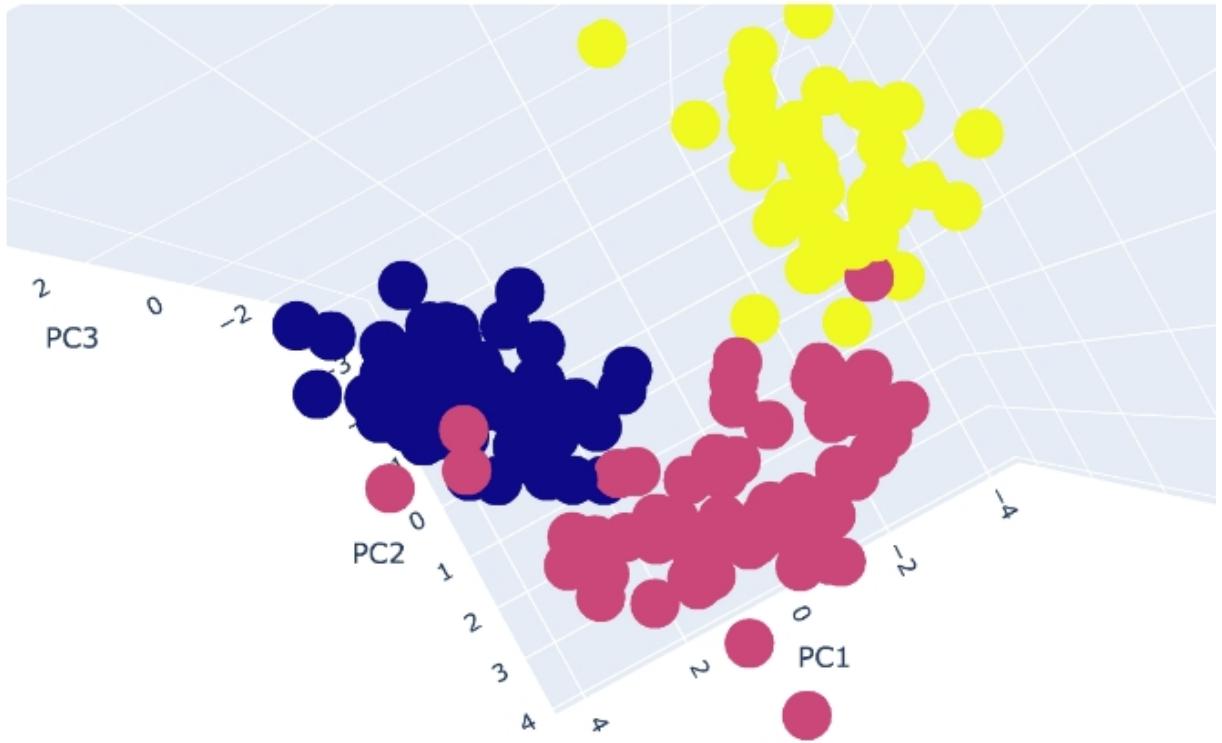
# 提取前三個主成分
pc1 = transformed_normalized[:, 0]
pc2 = transformed_normalized[:, 1]
pc3 = transformed_normalized[:, 2]

# 將每個點分配到不同的群組
df = pd.DataFrame({'x': pc1, 'y': pc2, 'z': pc3, 'labels': labels})

# 創建三維散點圖(可轉動)
fig = px.scatter_3d(df, x='x', y='y', z='z', color='labels')

# 添加圖例和標籤
fig.update_layout(scene=dict(xaxis_title='PC1', yaxis_title='PC2', zaxis_title='PC3',
                             showlegend=True, coloraxis_colorbar_tickvals=[1, 2, 3]))

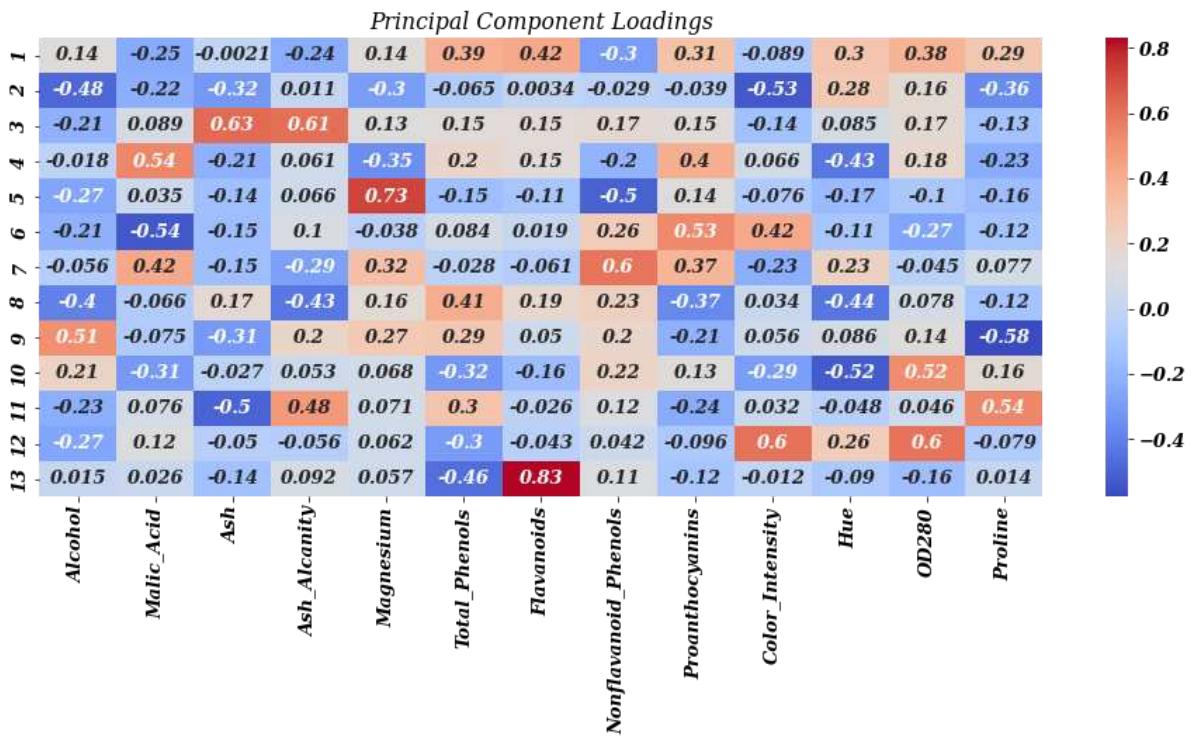
fig.show()
```



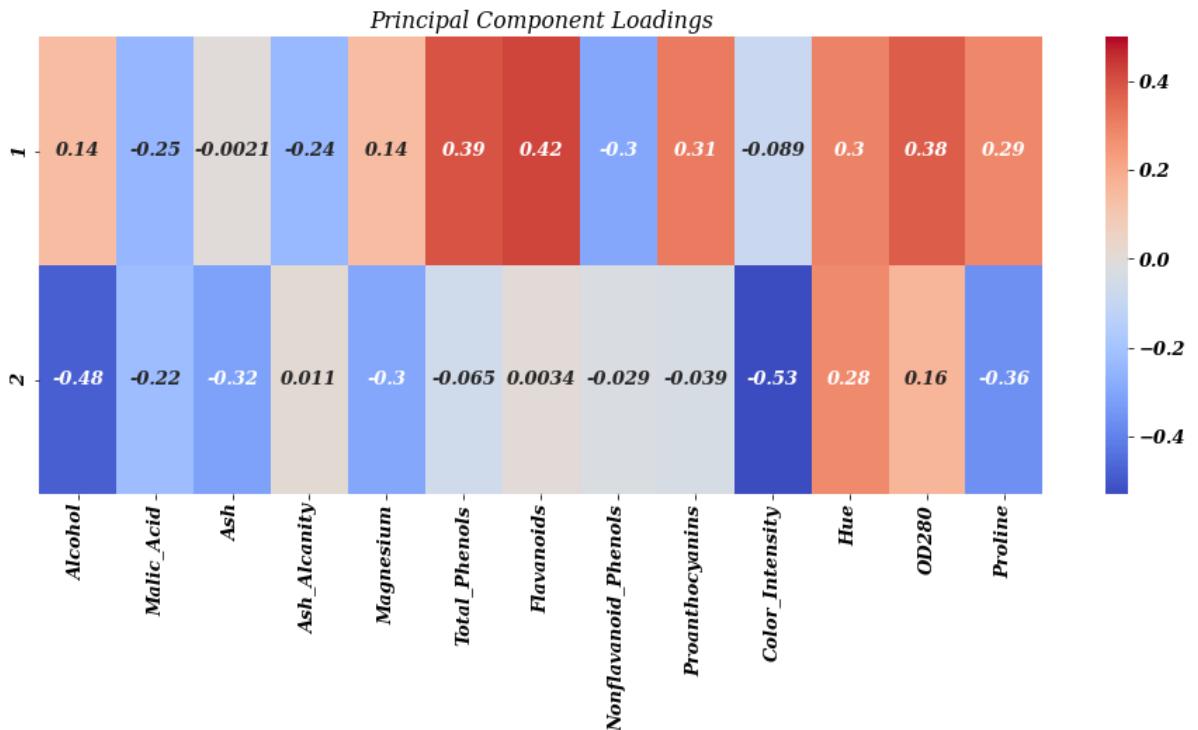
觀察三維散點圖後發現，加入前三個主成份對於區分群組的能力略有提高，但是並不明顯。這可能是因為前兩個主成份已經足以有效地區分群組，第三個主成份的加入只會對結果產生微小的影響，並不會明顯提高群組分辨能力。

- 
1.  $Z_1$  與  $Z_2$  都是從原變數組合而成的新變數，可否從  $Z_1$  與  $Z_2$  的組成係數，如式(10)，看出原變數哪個比較重要？哪個比較不重要？若再與原變數間的相關係數圖對照，是否透露相同的訊息。請提出你的觀察心得。

```
In [ ]: # 繪製主成分負載圖
plt.figure(figsize=(16, 6))
sns.heatmap(pca_normalized.components_, cmap='coolwarm', annot=True, xticklabels=True)
plt.title('Principal Component Loadings')
plt.show()
```



```
In [ ]: #僅切割出前兩個主成份的組成係數
plt.figure(figsize=(16, 6))
sns.heatmap(pca_normalized.components_[:2, :], cmap='coolwarm', annot=True,
            xticklabels=data.columns, yticklabels=np.arange(1, 3), vmax=0.5)
plt.title('Principal Component Loadings')
plt.show()
```



從組成係數熱圖可以看出，對於第一個主成分，'Flavanoids'、'total\_phenols'和'OD280'這三個變數對第一主成分的貢獻最大，因此可以認為這三個變數比較重要。

而'Ash'和'Color\_Intensity'這兩個變數對第一主成分的貢獻相對較小，可以認為它們比較不重要。

從第二主成分的組成係數來看，'Alcohol'和'Color\_Intensity'這兩個變數對第一主成分的貢獻最大，因此可以認為這兩個變數比較重要。而'Flavanoids'這個變數的係數則為最接近零，表

示這個變數對第二主成分的貢獻相對較小，可以認為它比較不重要。

第一主成分中的三個變數'Flavanoids'、'total\_phenols'和'OD280' 的相關係數如下：

- 'Flavanoids' 和 'total\_phenols' 的相關係數為 0.86
- 'total\_phenols' 和 'OD280' 的相關係數為 0.7
- 'Flavanoids' 和 'OD280' 的相關係數為 0.79

可以看出，第一個主成分主要由 'Flavanoids'、'total\_phenols' 和 'OD280' 三個變量組成，且它們之間的組成係數相當接近。這與它們之間的相關性較高是一致的

對於第二個主成分：

- 'Alcohol'和'Color\_Intensity'的相關係數為0.55

可以看出，第二個主成分主要由'Alcohol'和'Color\_Intensity' 這兩個變量組成，且它們之間的組成係數相當接近。這與它們之間的相關性較高是一致的

---

**資料集（二）**：資料則是同樣來自 `sklearn.datasets` 的一組關於乳癌患者腫瘤的影像量測資料。量測變數 30 個，樣本數 569 位患者，區分為兩個群組，分別是 Malignant（惡性腫瘤）與 Benign（良性腫瘤）。請注意，由於變數多，因此如前一練習的相關性圖，必須做些改變。

導入初步套件

```
In [ ]: import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from mpl_toolkits.mplot3d import Axes3D
```

```
In [ ]: fontparams = {'font.size': 13.5, 'font.weight':'bold',
                  'font.family':'arial', 'font.style':'italic'}
plt.rcParams.update(fontparams)
labelparams = {'size': 14.5, 'weight':'semibold',
               'family':'serif', 'style':'italic'}
```

資料讀入

```
In [ ]: # 載入乳癌資料集
cancer = load_breast_cancer()
labels = pd.Series(cancer.target, name='label')
# 轉換為 pandas DataFrame
data = pd.DataFrame(cancer.data, columns=cancer.feature_names)
data.head()
```

Out[ ]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symm
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.

5 rows × 30 columns

In [ ]: `data.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 30 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   mean radius      569 non-null   float64
 1   mean texture     569 non-null   float64
 2   mean perimeter   569 non-null   float64
 3   mean area        569 non-null   float64
 4   mean smoothness  569 non-null   float64
 5   mean compactness 569 non-null   float64
 6   mean concavity   569 non-null   float64
 7   mean concave points 569 non-null   float64
 8   mean symmetry    569 non-null   float64
 9   mean fractal dimension 569 non-null   float64
 10  radius error     569 non-null   float64
 11  texture error    569 non-null   float64
 12  perimeter error  569 non-null   float64
 13  area error       569 non-null   float64
 14  smoothness error 569 non-null   float64
 15  compactness error 569 non-null   float64
 16  concavity error  569 non-null   float64
 17  concave points error 569 non-null   float64
 18  symmetry error   569 non-null   float64
 19  fractal dimension error 569 non-null   float64
 20  worst radius     569 non-null   float64
 21  worst texture    569 non-null   float64
 22  worst perimeter   569 non-null   float64
 23  worst area        569 non-null   float64
 24  worst smoothness  569 non-null   float64
 25  worst compactness 569 non-null   float64
 26  worst concavity   569 non-null   float64
 27  worst concave points 569 non-null   float64
 28  worst symmetry    569 non-null   float64
 29  worst fractal dimension 569 non-null   float64
dtypes: float64(30)
memory usage: 133.5 KB
```

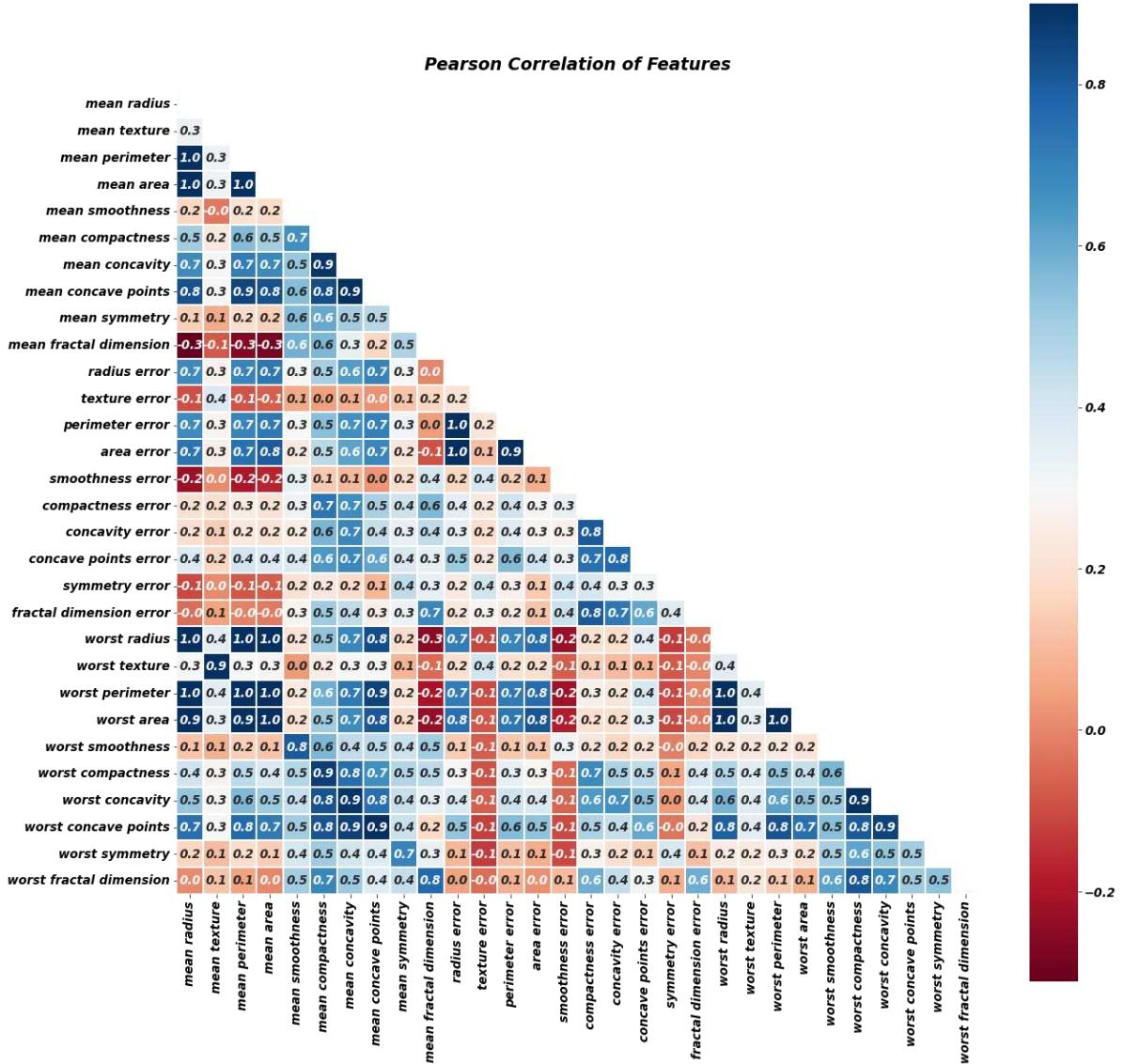
這個資料集中所有的特徵變數皆完整無缺失值，且其資料型態為數值型，方便進行進一步的分析。

1. 繪製變數間的相關係數圖，以觀察變數間是否存在相關性

```
In [ ]: colormap = plt.cm.RdBu
plt.figure(figsize=(20,20))
plt.title('Pearson Correlation of Features', y=1.02, size=19, fontweight='bold')

corr_matrix = data.astype(float).corr() #相關係數矩陣
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))
sns.heatmap(corr_matrix, fmt='.1f', linewidths=0.1, vmax=0.9,
            square=True, mask=mask, cmap=colormap, linecolor='white', annot=True)
```

Out[ ]: <AxesSubplot:title={'center':'Pearson Correlation of Features'}>



由於相關係數矩陣過大，我選擇將其進行簡化處理。

以下為進行相關係數矩陣簡化的三種方式：

- 拆分相關係數矩陣成兩半，分別繪製兩個子矩陣的相關係數圖，減少圖形中變量數量。
- 選擇相同特徵的變量，繪製它們之間的相關係數圖，更好地理解特定特徵之間的關係。
- 篩選出與目標變數高度相關的變量，再繪製相關係數圖，更好地理解目標變數與其他變量之間的關係，提高繪圖效率。

方式1:拆分相關係數矩陣

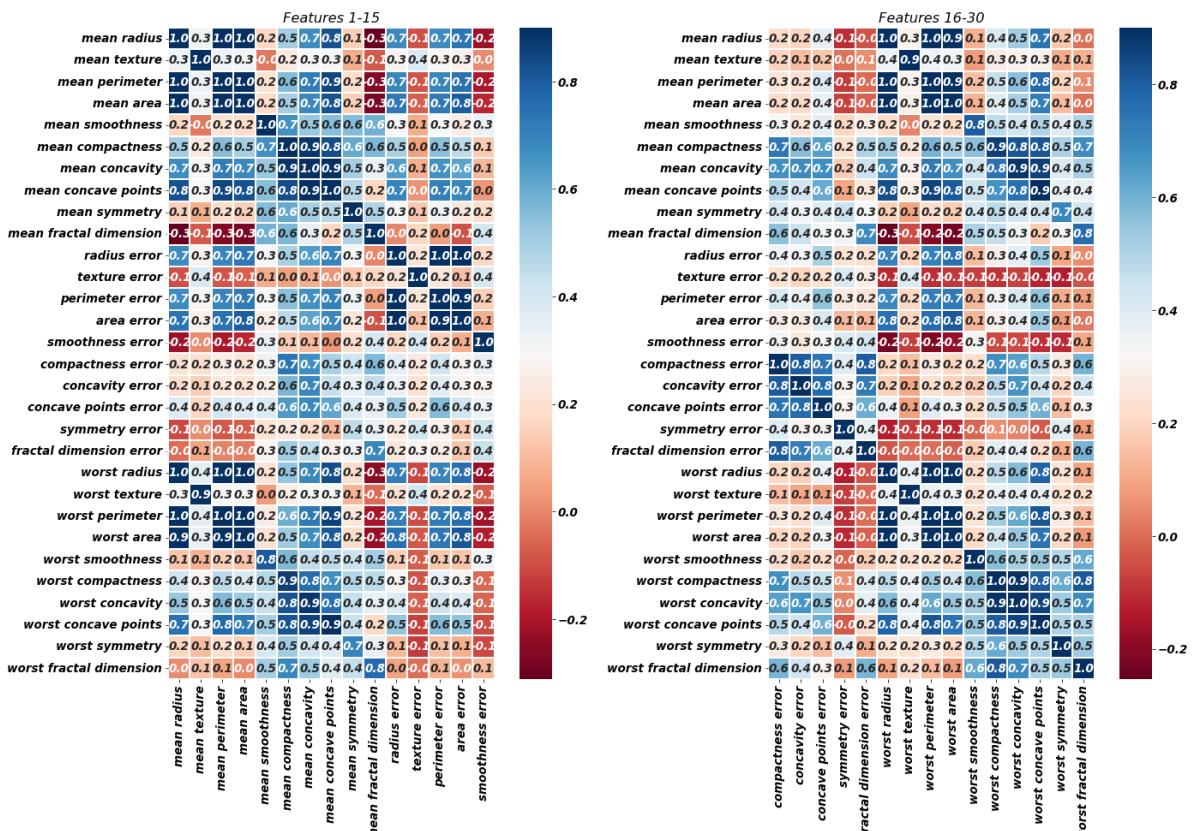
```
In [ ]: colormap = plt.cm.RdBu
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(23, 14))
fig.suptitle('Pearson Correlation of Features',
             y=1.02, size=19, fontweight='bold')

corr_matrix = data.astype(float).corr() # 相關係數矩陣

sns.heatmap(corr_matrix.iloc[:, :15], fmt='.1f',
            linewidths=0.1, vmax=0.9,
            square=True, cmap=colormap, linecolor='white', annot=True, ax=axs[0])
axs[0].set_title('Features 1-15')

sns.heatmap(corr_matrix.iloc[:, 15:], fmt='.1f',
            linewidths=0.1, vmax=0.9,
            square=True, cmap=colormap, linecolor='white', annot=True, ax=axs[1])
axs[1].set_title('Features 16-30')
plt.show()
```

Pearson Correlation of Features



## 方式2:相同特徵下的相關係數矩陣

```
In [ ]: #特徵篩選
data.columns
features_mean = data.columns[0:10]
features_error = data.columns[10:20]
features_worst = data.columns[20:]
```

```
In [ ]: #繪製mean群組的相關熱力圖
corr_matrix_mean = data[features_mean].corr()

plt.figure(figsize=(11,11))
```

```

plt.title('Pearson Correlation of Features for Mean Group',
          y=1.05, size=19, fontweight = 'bold' )
mask = np.triu(np.ones_like(corr_matrix_mean, dtype=bool))
g = sns.heatmap(corr_matrix_mean, cbar = True, annot=True,
                 annot_kws={'size': 15}, fmt= '.2f',
                 square = True, mask = mask, cmap = 'coolwarm' )
g.set_xticklabels(rotation=90, labels = features_mean, size = 15)
g.set_yticklabels(rotation=0, labels = features_mean, size = 15)

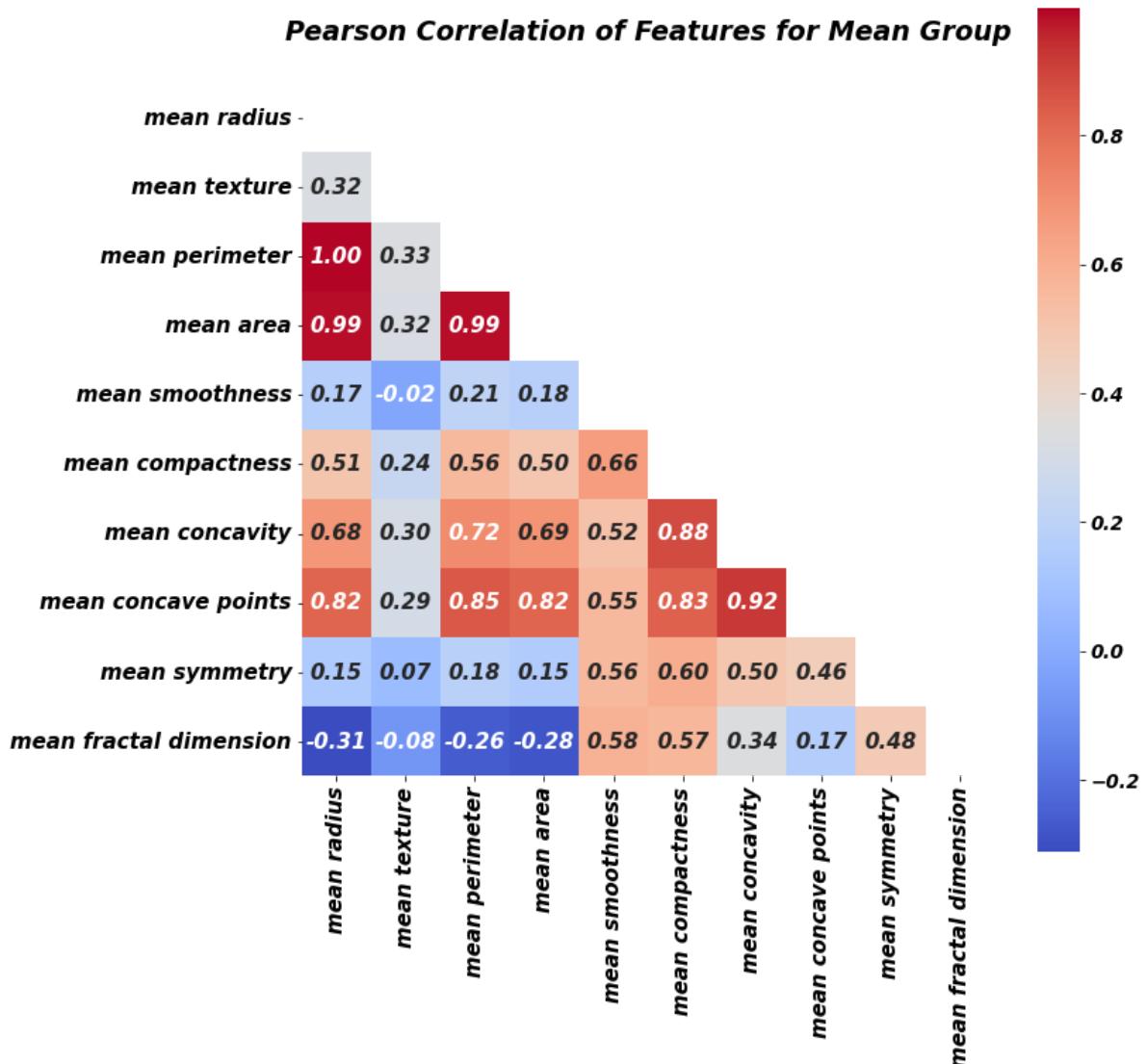
```

Out[ ]:

```

[Text(0, 0.5, 'mean radius'),
 Text(0, 1.5, 'mean texture'),
 Text(0, 2.5, 'mean perimeter'),
 Text(0, 3.5, 'mean area'),
 Text(0, 4.5, 'mean smoothness'),
 Text(0, 5.5, 'mean compactness'),
 Text(0, 6.5, 'mean concavity'),
 Text(0, 7.5, 'mean concave points'),
 Text(0, 8.5, 'mean symmetry'),
 Text(0, 9.5, 'mean fractal dimension')]

```



In [ ]:

```

#繪製error群組的相關熱力圖
corr_matrix_error = data[features_error].corr()

plt.figure(figsize=(11,11))
plt.title('Pearson Correlation of Features for Error Group',
          y=1.05, size=19, fontweight = 'bold' )
mask = np.triu(np.ones_like(corr_matrix_error, dtype=bool))
g = sns.heatmap(corr_matrix_error, cbar = True, annot=True,
                 annot_kws={'size': 15}, fmt= '.2f', square = True,
                 mask = mask, cmap = 'coolwarm' )

```

```
g.set_xticklabels(rotation=90, labels = features_error, size = 15)
g.set_yticklabels(rotation=0, labels = features_error, size = 15)
```

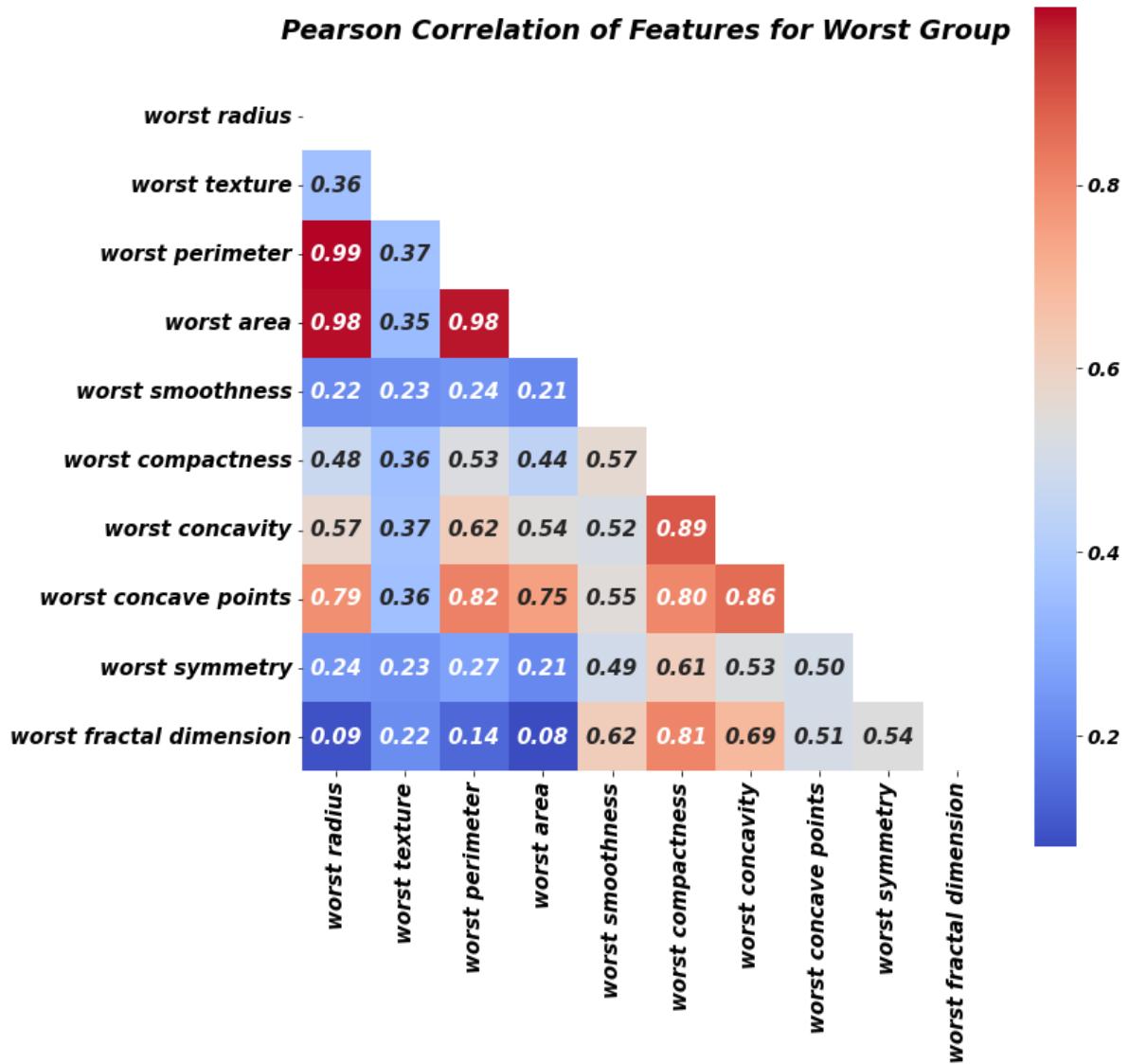
```
Out[ ]: [Text(0, 0.5, 'radius error'),
Text(0, 1.5, 'texture error'),
Text(0, 2.5, 'perimeter error'),
Text(0, 3.5, 'area error'),
Text(0, 4.5, 'smoothness error'),
Text(0, 5.5, 'compactness error'),
Text(0, 6.5, 'concavity error'),
Text(0, 7.5, 'concave points error'),
Text(0, 8.5, 'symmetry error'),
Text(0, 9.5, 'fractal dimension error')]
```



```
#繪製worst群組的相關熱力圖
corr_matrix_worst = data[features_worst].corr()

plt.figure(figsize=(11,11))
plt.title('Pearson Correlation of Features for Worst Group',
          y=1.05, size=19, fontweight = 'bold' )
mask = np.triu(np.ones_like(corr_matrix_worst, dtype=bool))
g = sns.heatmap(corr_matrix_worst, cbar = True, annot=True,
                 annot_kws={'size': 15}, fmt= '.2f', square = True,
                 mask = mask, cmap = 'coolwarm' )
g.set_xticklabels(rotation=90, labels = features_worst, size = 15)
g.set_yticklabels(rotation=0, labels = features_worst, size = 15)
```

```
Out[ ]: [Text(0, 0.5, 'worst radius'),
Text(0, 1.5, 'worst texture'),
Text(0, 2.5, 'worst perimeter'),
Text(0, 3.5, 'worst area'),
Text(0, 4.5, 'worst smoothness'),
Text(0, 5.5, 'worst compactness'),
Text(0, 6.5, 'worst concavity'),
Text(0, 7.5, 'worst concave points'),
Text(0, 8.5, 'worst symmetry'),
Text(0, 9.5, 'worst fractal dimension')]
```



方式3:篩選出與目標變數高度相關的變量再繪製相關係數矩陣

```
In [ ]: #將原始data併入目標變數label
All_data = pd.concat([data, labels], axis=1)
```

```
In [ ]: # 計算所有變數之間的相關係數矩陣
corr_matrix = All_data.corr()

# 設定相關係數閾值
threshold = 0.6

# 找出與目標變數高度相關的變數，建立篩選器
filter = np.abs(corr_matrix["label"]) > threshold

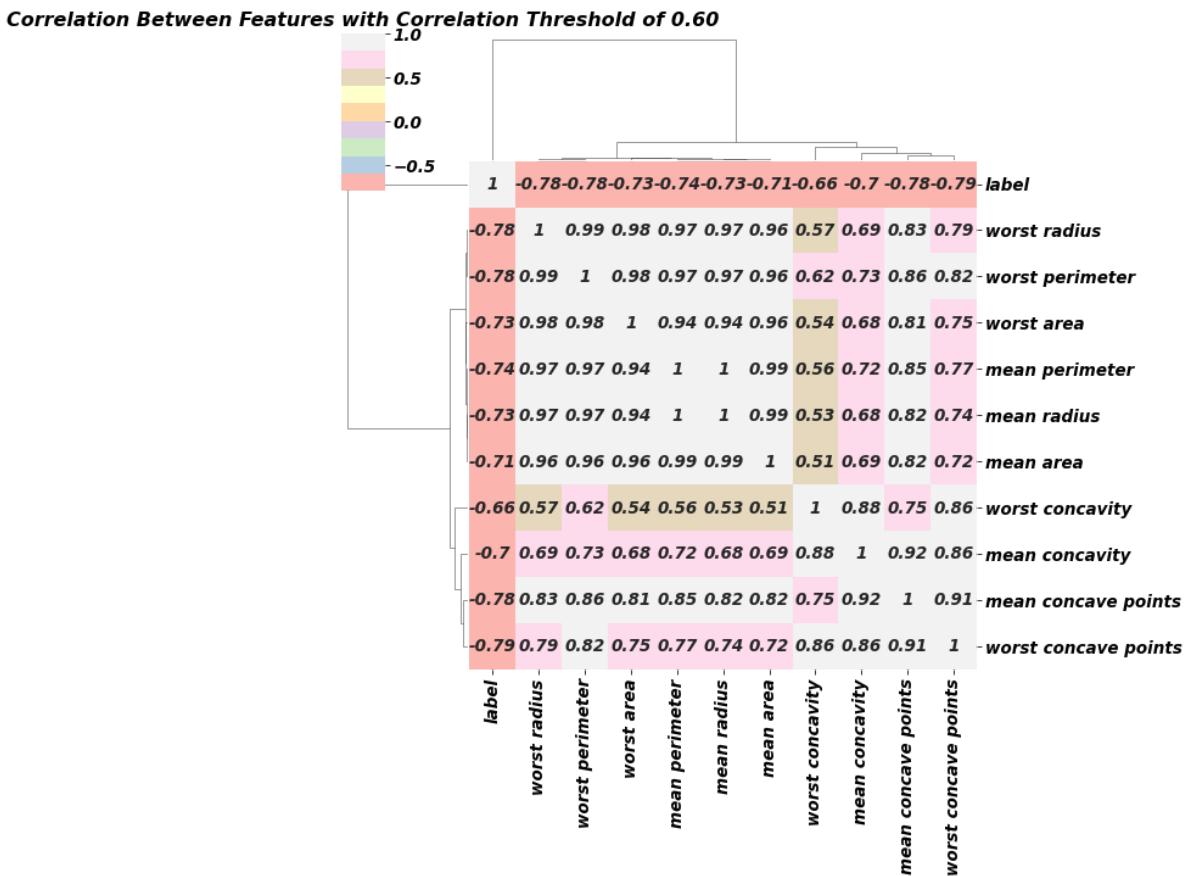
# 將篩選器應用到所有變數中，找出高度相關的變數名稱
```

```

corr_features = corr_matrix.columns[filter].tolist()

# 繪製高度相關變數之間的相關係數熱圖
sns.clustermap(All_data[corr_features].corr(),
                annot=True, cmap="Pastel1", vmax=1)
plt.title("Correlation Between Features with Correlation Threshold of 0.60",
          fontweight="bold", fontsize=16)
plt.show()

```



找出與目標變數高度相關的變數，可以幫助我們縮小分析的範圍，只關注那些對目標變數影響較大的變數。透過繪製這些變數之間的相關係數熱圖，可以讓我們更直觀地了解這些變數之間的相互關係。

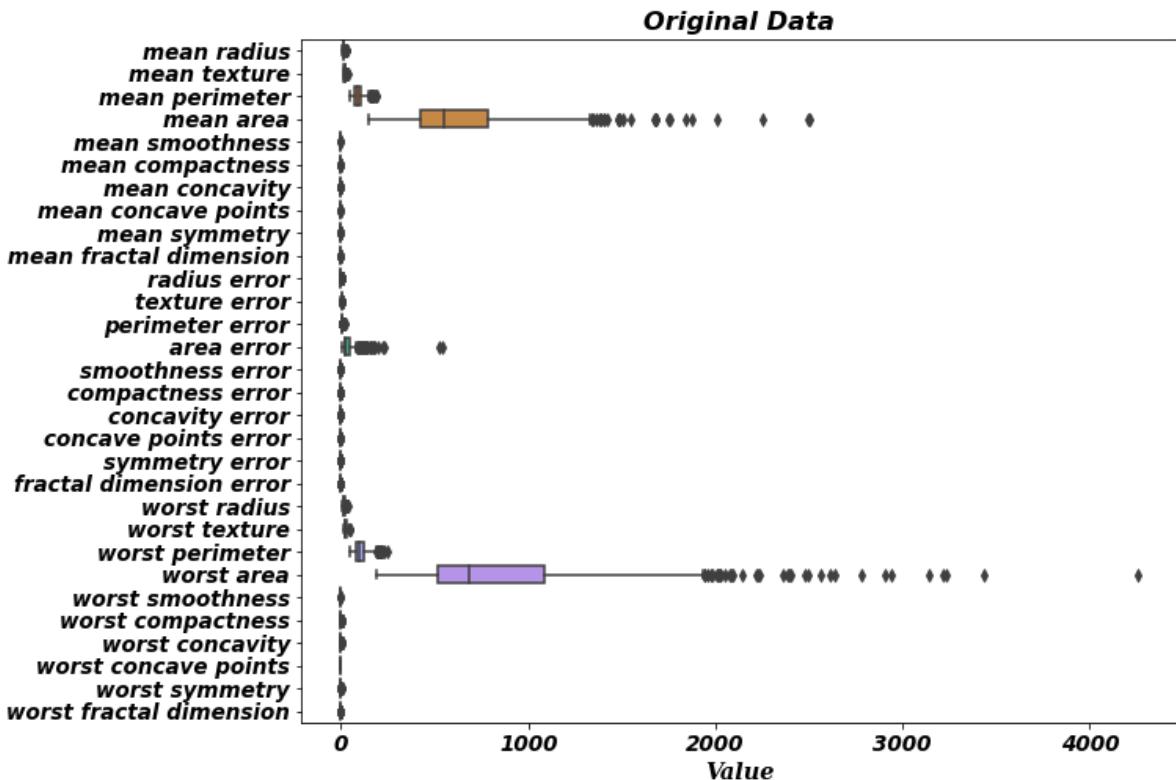
綜合以上三種相關係數分析，可以得出以下結果：

- 腫瘤大小相關的變數（mean radius、mean perimeter、mean area、radius error、perimeter error和area error）之間的相關係數都很高（相關係數大於0.9），這意味著它們之間存在著很強的正相關關係。
- 細胞核特徵相關的變數（mean texture、texture error、worst texture、mean smoothness、smoothness error和worst smoothness）之間的相關係數也較高，但是這些變數之間的相關係數比較不明顯（相關係數約在0.6–0.8之間）
- 腫瘤大小相關的變數（mean radius、mean perimeter、mean area、radius error、perimeter error和area error）和細胞核特徵相關的變數（mean texture、texture error、worst texture、mean smoothness、smoothness error和worst smoothness）之間的相關係數也都比較高，這意味著它們之間存在著一定的關聯性。

1. 繪製一張含每個變數的盒鬚圖（Boxplot），觀察每個變數的scaling，作為是否標準化的參考。

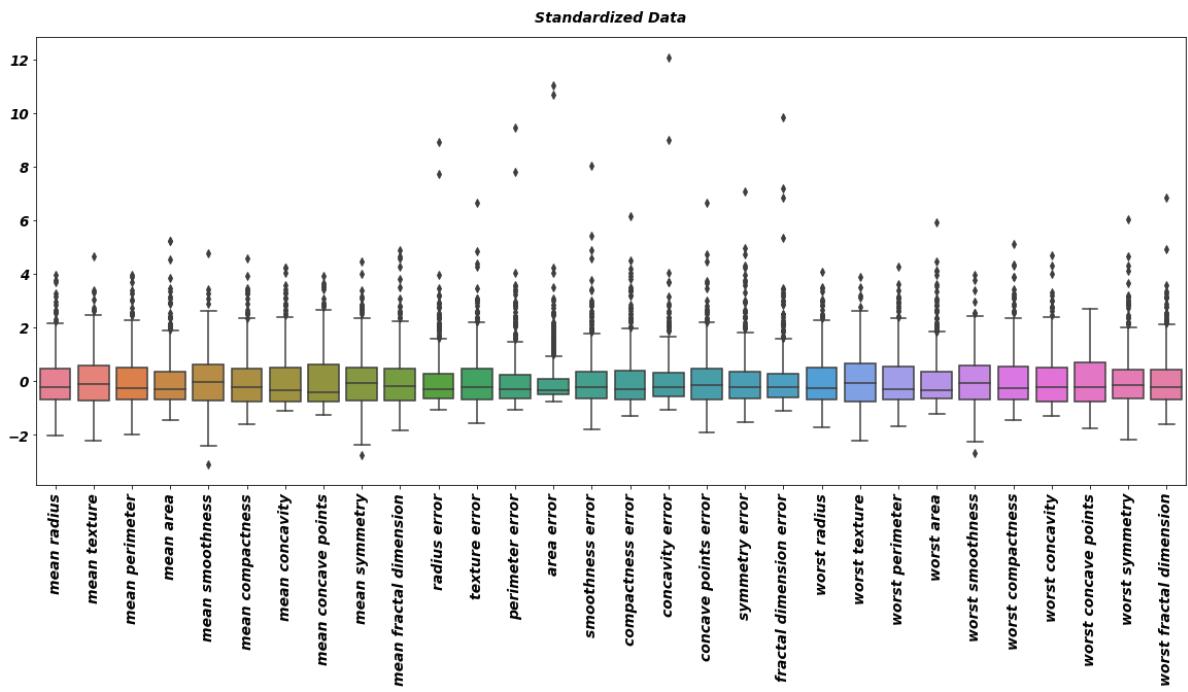
```
In [ ]: # 使用sns繪製Boxplot, 變數排列方式為水平  
plt.figure(figsize = (10, 8))  
ax = sns.boxplot(data = data, palette = 'husl', orient = 'h')  
ax.set_title('Original Data', fontweight = 'bold')  
ax.set_xlabel('Value', labelparams)
```

```
Out[ ]: Text(0.5, 0, 'Value')
```



根據上面的箱型圖，我們可以看到 "mean perimeter"、"mean area"、"mean concavity"、"mean concave points"、"worst radius"、"worst perimeter"、"worst area" 這幾個變數的範圍與其他變數相比差異較大，它們的值偏向於大型值，所以可以對變數採取標準化。

```
In [ ]: # 使用sns繪製標準化Boxplot, 變數排列方式為水平  
# 初始化 StandardScaler 對象  
scaler = StandardScaler()  
  
# 應用標準化到數據上  
data_scaled = scaler.fit_transform(data)  
  
# 將標準化後的數據轉換成 Pandas DataFrame  
data_scaled = pd.DataFrame(data_scaled, columns=data.columns)  
  
# 繪製標準化後的Boxplot  
plt.figure(figsize=(20, 8))  
ax = sns.boxplot(data=data_scaled, palette='husl')  
ax.set_title('Standardized Data', y=1.02, size=14, fontweight='bold')  
plt.xticks(rotation=90)  
ax.tick_params(axis='x', labelsize=14) # 設定x軸標籤字體大小
```



在水平方向繪製盒形圖時，由於存在一些離群值(超出過多的極端值)，這些離群值會被繪製在盒形圖的一側，而使得整個盒形圖整體靠左而顯得不太美觀，所以改成繪製垂直的盒形圖。

1. 進行主成分分析，繪製特徵值由大而小的分布與 scree plot。

```
In [ ]: #創建pca_raw對象，不進行標準化
pca_raw = PCA()

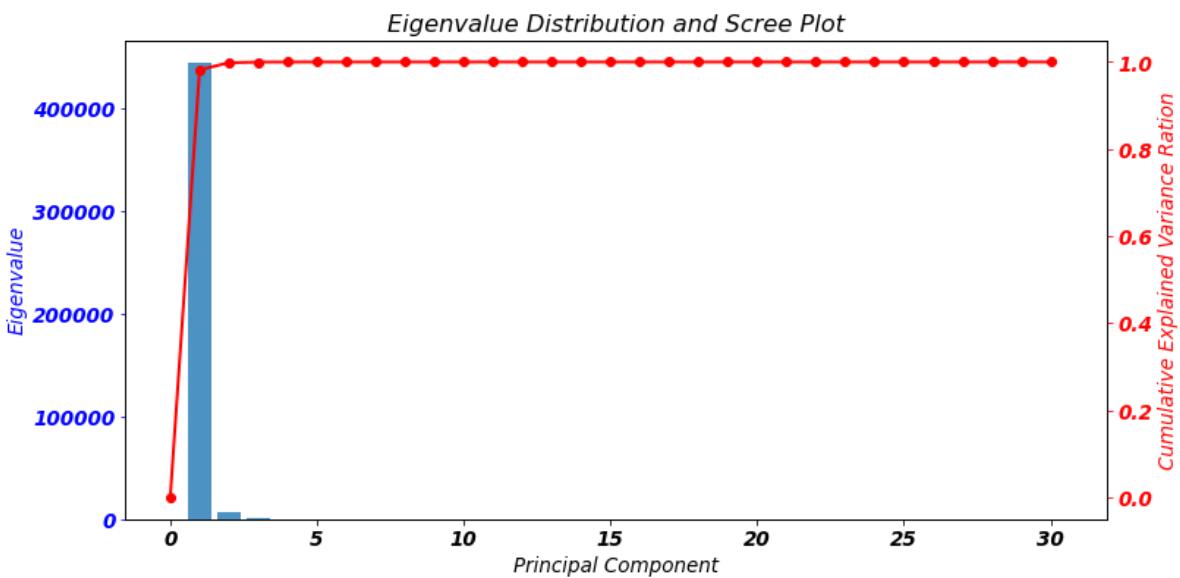
# 擱合數據並進行轉換
pca_raw.fit(data)
transformed_raw = pca_raw.transform(data)

# 計算特徵值和特徵向量
eigenvalues = pca_raw.explained_variance_
eigenvectors = pca_raw.components_

# 繪製特徵值分佈圖和scree plot
fig, ax1 = plt.subplots(figsize=(12, 6))

# 繪製特徵值條形圖
ax1.bar(range(1, len(eigenvalues)+1), eigenvalues, alpha=0.8)
ax1.set_xlabel('Principal Component')
ax1.set_ylabel('Eigenvalue', color='b')
ax1.tick_params('y', colors='b')
ax1.set_title('Eigenvalue Distribution and Scree Plot')

# 繪製累積解釋方差曲線
ax2 = ax1.twinx()
ax2.plot(np.insert(np.cumsum(pca_raw.explained_variance_ratio_), 0, 0), 'r')
ax2.set_ylabel('Cumulative Explained Variance Ration', color='r')
ax2.tick_params('y', colors='r')
plt.show()
ax.tick_params(axis='x', labelsize=14) # 設定x軸標籤字體大小
```



由於未進行標準化，因此在解釋方差方面存在偏差。從特徵值條形圖和累積解釋變異曲線可以看出，第一主成分幾乎解釋了所有的解釋方差，這表明第一主成分可能包含了大部分數據的變異性。

```
In [ ]: # 創建PCA對象，進行標準化
pca_normalized = PCA()
scaler = StandardScaler()

# 對數據進行標準化和擬合
data_standardized = scaler.fit_transform(data)
pca_normalized.fit(data_standardized)
transformed_normalized = pca_normalized.transform(data_standardized)

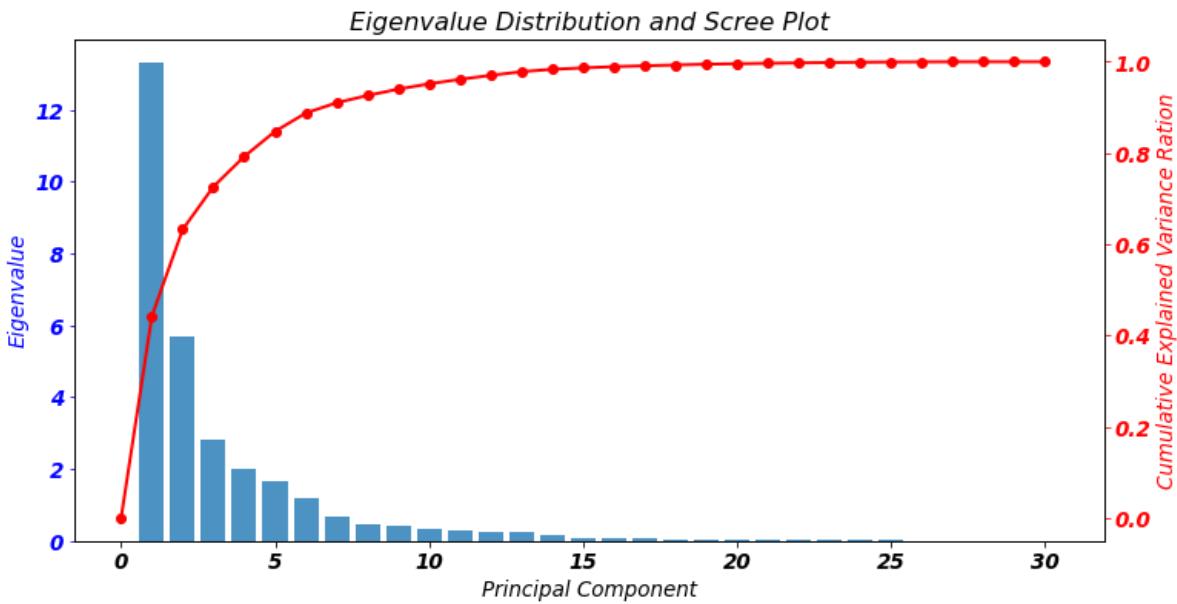
# 計算特徵值和特徵向量
eigenvalues = pca_normalized.explained_variance_
eigenvectors = pca_normalized.components_

# 繪製特徵值分佈圖和scree plot
fig, ax1 = plt.subplots(figsize=(12, 6))

# 繪製特徵值條形圖
ax1.bar(range(1, len(eigenvalues)+1), eigenvalues, alpha=0.8)
ax1.set_xlabel('Principal Component')
ax1.set_ylabel('Eigenvalue', color='b')
ax1.tick_params('y', colors='b')
ax1.set_title('Eigenvalue Distribution and Scree Plot')

# 繪製累積解釋方差曲線
ax2 = ax1.twinx()
ax2.plot(np.insert(np.cumsum(pca_normalized.explained_variance_ratio_), 0, 0), 'ro-', linewidth=2)
ax2.set_ylabel('Cumulative Explained Variance Ration', color='r')
ax2.tick_params('y', colors='r')

plt.show()
```



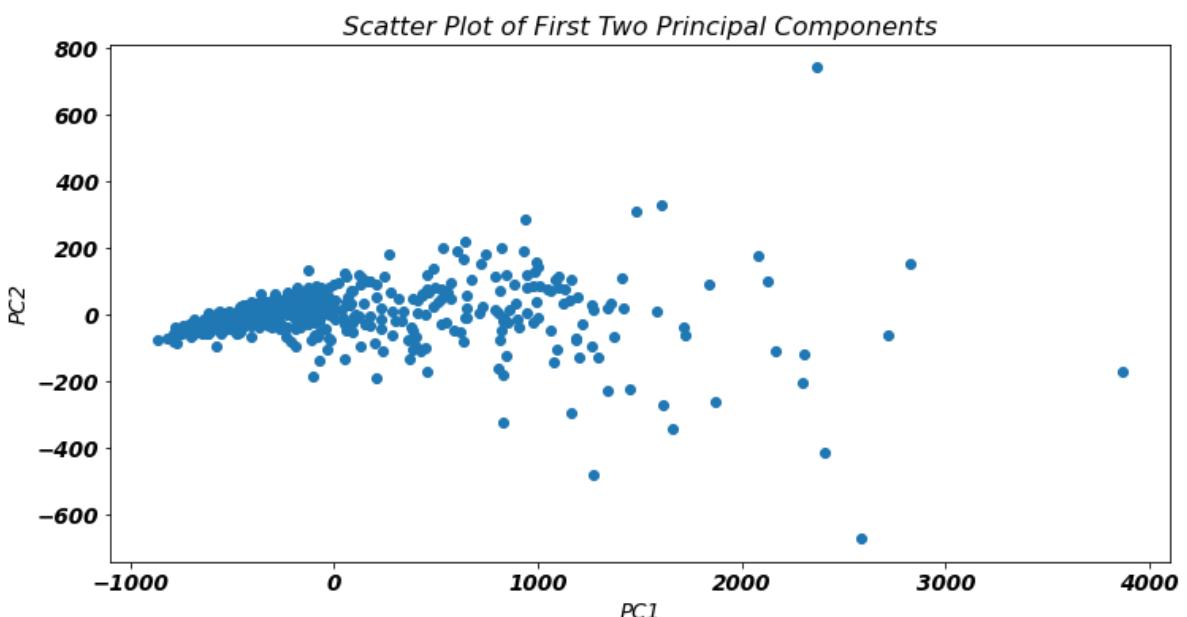
經過標準化後，從特徵值條形圖和累積解釋變異曲線可以看出，前十個主成分能解釋大部分的變異。對於未標準化的情況，這個結果更加可靠。

---

1. 假設先不看這項標籤。利用主成分分析取得前兩項成分，並繪製其散布圖。如是否可以從兩個主成分的散布圖中看出二個群組？

```
In [ ]: #繪製未標準化數據的前兩個主成分散布圖
pc1 = transformed_raw[:, 0]
pc2 = transformed_raw[:, 1]

plt.figure(figsize=(12, 6))
plt.scatter(pc1, pc2)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('Scatter Plot of First Two Principal Components')
plt.show()
```



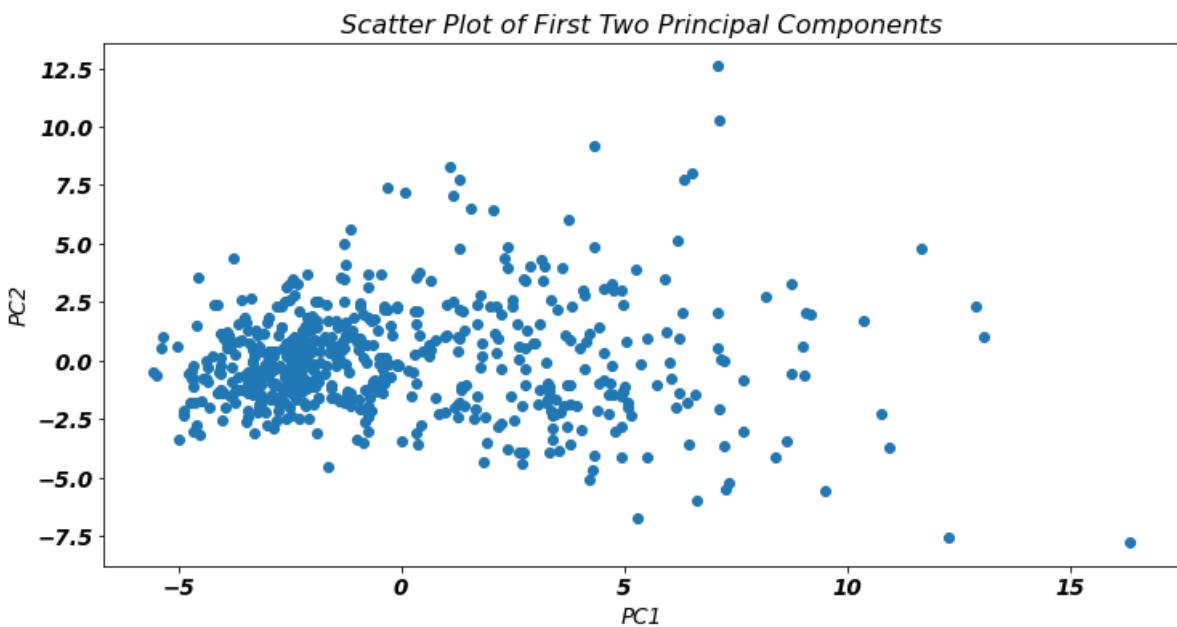
```
In [ ]: #繪製標準化數據的前兩個主成分散布圖
pc1 = transformed_normalized[:, 0]
```

```

pc2 = transformed_normalized[:, 1]

plt.figure(figsize=(12, 6))
plt.scatter(pc1, pc2)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('Scatter Plot of First Two Principal Components')
plt.show()

```



對於乳癌資料集的未標準化和標準化數據的前兩個主成分散布圖，我認為其分群能力相似。因為兩張圖的左側都有密集的點群，右側則較為稀疏，這可能暗示著這兩區域代表不同的群組。

---

1. 再依據每個資料的標籤，為每個在散布圖上的資料點塗上顏色

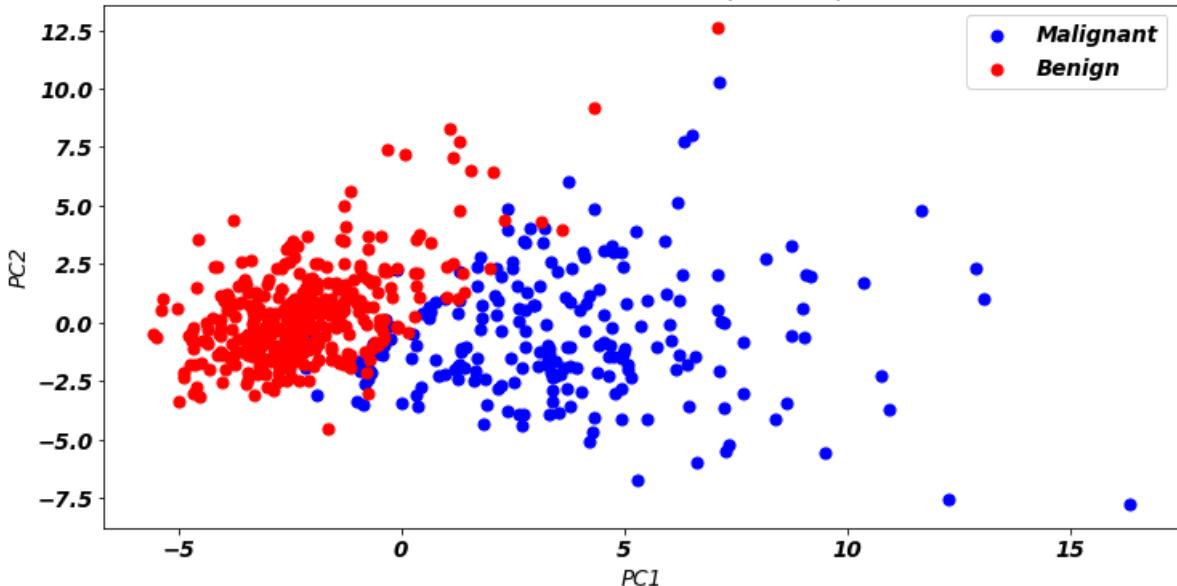
```

In [ ]: pc1 = transformed_normalized[:, 0]
          pc2 = transformed_normalized[:, 1]

          plt.figure(figsize=(12, 6))
          plt.scatter(pc1[labels==0], pc2[labels==0], marker='o',
                      c='Blue', s=50, label='Malignant')
          plt.scatter(pc1[labels==1], pc2[labels==1], marker='o',
                      c='Red', s=50, label='Benign')
          plt.xlabel('PC1')
          plt.ylabel('PC2')
          plt.title('Scatter Plot of First Two Principal Components')
          plt.legend()
          plt.show()

```

Scatter Plot of First Two Principal Components



根據每個資料點的標籤，在散布圖上為它們塗上顏色後，可以觀察到，這與之前的分群猜測相同。

---

1. 如果採三個主成分是否具備更好的群組分辨能力。請嘗試旋轉立體圖的角度以取得最好的辨別視野。

In [ ]: `import plotly.express as px`

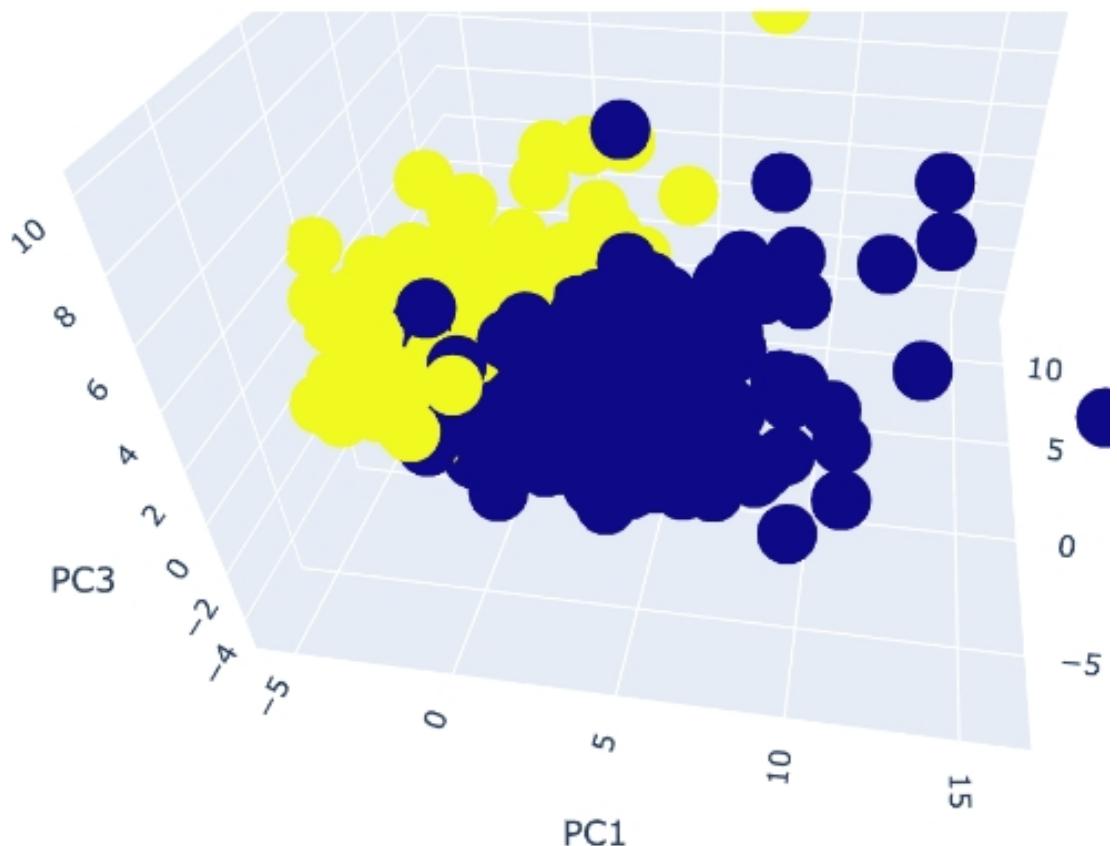
```
# 提取前三個主成分
pc1 = transformed_normalized[:, 0]
pc2 = transformed_normalized[:, 1]
pc3 = transformed_normalized[:, 2]

# 將每個點分配到不同的群組
df = pd.DataFrame({'x': pc1, 'y': pc2, 'z': pc3, 'labels': labels})

# 創建三維散點圖
fig = px.scatter_3d(df, x='x', y='y', z='z', color='labels')

# 設定標題和軸標籤
fig.update_layout(scene=dict(xaxis_title='PC1', yaxis_title='PC2',
                             zaxis_title='PC3'), coloraxis_colorbar_tickvals=[

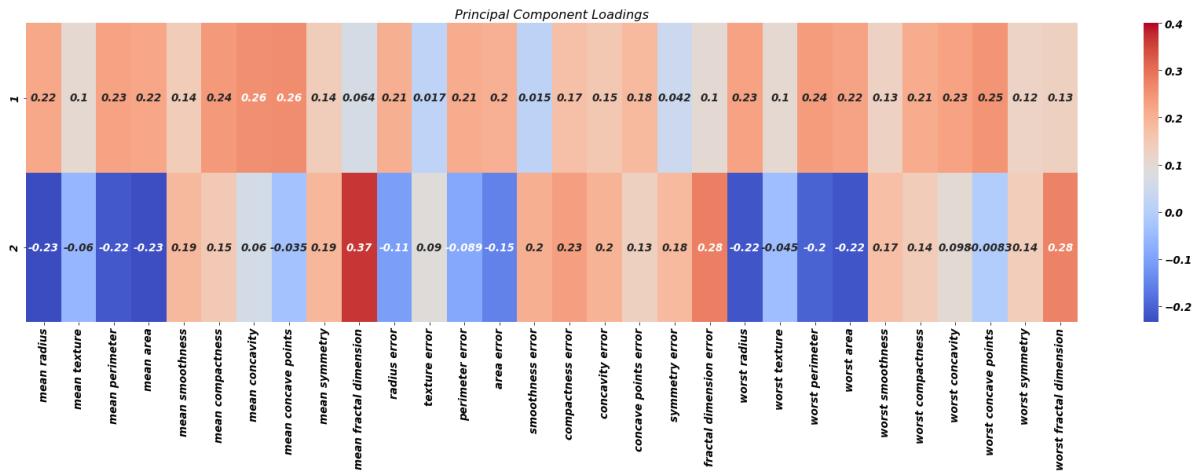
# 顯示圖表
fig.show()
```



兩個主成分的散布圖在兩群交界處有很多重疊，因此其群組分辨能力相對較差。然而，使用三個主成分所建立的立體圖可以透過旋轉得到更好的辨別視野，能夠更清楚地觀察到不同群組之間的區別，因此具有更好的群組分辨能力。換言之，加入第三個主成分可以更好地呈現資料的特徵，進一步增強群組分辨能力。

- 
1.  $Z_1$  與  $Z_2$  都是從原變數組合而成的新變數，可否從  $Z_1$  與  $Z_2$  的組成係數看出原變數哪個比較重要？哪個比較不重要？若再與原變數間的相關係數圖對照，是否透露相同的訊息。請提出你的觀察心得。

```
In [ ]: #僅切割出前兩個主成份的組成係數
plt.figure(figsize=(30, 7))
sns.heatmap(pca_normalized.components_[:2, :], cmap='coolwarm', annot=True,
            xticklabels=data.columns, yticklabels=np.arange(1,3), vmax=0.4)
plt.title('Principal Component Loadings')
plt.show()
```



根據前兩個主成分的組成係數，可以發現第一個主成分與'mean radius', 'mean perimeter', 'mean area', 'mean compactness', 'mean concavity', 'mean concave points', 'worst radius', 'worst perimeter', 'worst area', 'worst compactness', 'worst concavity', 'worst concave points'這些變數具有較高的組成係數，顯示這些變數對於第一主成分貢獻較大，可以視為較重要的變數。但其實組成係數都約在0.2左右，表示這幾個變數對於主成分的解釋力差不多，沒有特別突出的變數。

此外，原變數間的相關係數也提供了額外的信息。可以發現第三種相關係數圖(群集)包含了'mean radius', 'mean perimeter', 'mean area', 'worst radius', 'worst perimeter', 'worst area'等變數，這些變數之間的相關性超過0.5(閥值)，表示它們之間具有較強的相關性。

對於第二主成分'mean fractal dimension', 'worst fractal dimension'和'fractal dimension error'這些變數具有較高的組成係數，顯示這些變數對於第二主成分貢獻較大，可以視為較重要的變數。

此外，在乳癌資料中，'mean fractal dimension'、'worst fractal dimension'和'fractal dimension error'三個變數都代表了腫瘤細胞核的形狀複雜度指標，彼此高度相關。因此，它們在第二主成分中的組成係數也較高是符合預期的。

## 結語

在本次資料探索和主成分分析的專案中，我們深入研究了兩個不同的資料集：紅酒化學成分數據集和乳癌患者腫瘤影像量測資料集。透過相關係數分析，我們努力理解變數之間的相關性以及其對資料集的影響。

在紅酒數據集中，我們發現了一些有趣的相關性模式，這有助於我們更好地理解紅酒化學成分之間的關係。通過主成分分析，我們能夠成功地將多維數據壓縮成更少的主成分，並將數據點在二維空間中可視化，進一步揭示了數據的分佈情況。

在乳癌患者腫瘤影像量測資料集中，考慮到數據集的複雜性，我們運用相關性分析的結果來引導數據縮減的過程。這有助於我們減少了多個變數的數據集複雜性，同時保留了重要的信息。

這個專案強調了數據探索和主成分分析在理解和解釋多變數數據時的重要性。通過採用不同的數據可視化和分析技術，我們可以更深入地了解數據的結構，並發現潛在的模式和關聯性。這對於支持決策、模型建構和進一步的分析都具有重要的啟發作用。

# 透過SVD的目光：圖像壓縮和還原的技術巧思

---

## 專案前言：

圖像壓縮和還原技術在現代數據處理和儲存中起著重要作用。隨著大量數位圖像和視頻的生成和分享，我們需要有效的方法來減少數據的儲存和傳輸成本，同時又能保持圖像品質。奇異值分解（SVD）是一種常用的數學方法，可用於實現圖像壓縮。本專案旨在探討和實施SVD的"Rank q approximation" 技術，以實現圖像壓縮，同時保持足夠的圖像品質。

## 專案目標簡介：

本專案的主要目標是研究和實施圖像壓縮和還原技術，尤其是基於奇異值分解（SVD）的 "Rank q approximation" 方法。以下是我們的專案目標簡介：

### 1. 圖像壓縮和品質比較：

- 通過實施SVD的 "Rank q approximation" 技術，比較不同的q值對圖像的壓縮效果。
- 評估壓縮後的圖像品質，找出最佳的q值。

### 2. 觀察手寫數字影像：

- 使用手寫數字數據集，實現隨機選擇和顯示手寫數字影像的功能。
- 編寫程式碼，使每次執行都能觀看不同的手寫數字影像，以熟悉數據。

### 3. 計算壓縮倍數和顯示壓縮圖像：

- 計算SVD的 "Rank q approximation" 方法對圖像的壓縮倍數。
- 顯示原始圖像和壓縮後的圖像，以直觀比較壓縮效果。

### 4. 圖像加密和解密：

- 選擇不同類型的五張圖像，包括人臉、水果、風景等。
- 使用SVD和 "Rank q approximation" 方法對圖像進行壓縮（加密），然後解密。
- 觀察解密後的圖像效果，並評估人臉和非人臉圖像的解密能力。

通過實現這些目標，我們將深入瞭解SVD在圖像處理中的應用，並研究圖像壓縮和還原技術的實際效能。這項專案將有助於我們更好地理解數據壓縮、圖像處理和圖像安全性方面的核心概念。

---

## 導入初步套件

```
In [ ]: import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

```

from mpl_toolkits.mplot3d import Axes3D
from numpy.linalg import svd
import matplotlib.image as mpimg

In [ ]: # fontparams = {'font.size': 13, 'font.weight':'bold',
# 'font.family':'arial', 'font.style':'italic'}
# plt.rcParams.update(fontparams)
labelparams = {'size': 14, 'weight':'semibold',
               'family':'serif', 'style':'italic'}

```

**習題 1：將一張圖像 X 利用 SVD 的 “Rank q approximation”，能達到壓縮的目的並保持圖像的品質。比較下列幾種對於圖像矩陣 X 的重組安排，並進行 “Rank q approximation”，在同樣的壓縮比之下，觀察還原後的圖像品質哪個最好？能說出理由嗎？**

1. X 不變
2. 將 X 以  $8 \times 8$  小圖 ( patch ) 進行切割，再將每個小圖拉成  $64 \times 1$  的向量，最後重組這些向量並排成新的  $64 \times N$  矩陣。
3. 同上，小圖大小為  $16 \times 16$  /per patch。
4. 同上，但分割成  $32 \times 32$  / per patch。
5. 其他。譬如，隨機挖取 X 裡面的 patch，可重疊，小圖大小自訂、數量隨意。

這道習題要求比較不同的重組安排對於圖像壓縮的影響。我們可以先定義一個函數，將圖像 X 分割成不同大小的小圖，並返回相應的矩陣。然後對每個矩陣進行 SVD，得到其前 q 個奇異值和相應的左右奇異向量。根據這些向量，可以使用下列公式進行 “Rank q approximation”：

$$X_q = U_q \Sigma_q V_q^T$$

其中， $U_q$  是  $m \times q$  的左奇異矩陣， $\Sigma_q$  是  $q \times q$  的奇異值矩陣， $V_q$  是  $n \times q$  的右奇異矩陣。 $X_q$  是重組後的圖像矩陣。

## 1-1. X 不變

```

In [ ]: imgfile = '/content/drive/MyDrive/淺度機器學習/lenna.png'
X = mpimg.imread(imgfile)
if len(X.shape) > 2:
    X = np.mean(X, axis=2) # convert RGB to grayscale
N, p = X.shape

U, E, VT = svd(X, full_matrices = False)
q = np.array([p/4, p/8, p/16]).astype('int')

fig, ax = plt.subplots(1, 3, figsize=(12, 4))
for i, r in enumerate(q):
    Xq = U[:, :r] @ np.diag(E[:r]) @ VT[:r, :]
    ax[i].imshow(Xq, cmap = 'gray')
    ax[i].set_title('Compression ratio: {}'.format(p/r/2))
    ax[i].set_xticks([])
    ax[i].set_yticks([])

plt.tight_layout()
plt.show()

```



當q值設定為p/4時，壓縮比例為2，重建圖片與原圖相比差異不大，仍然能保留許多細節和清晰度。然而，當q值分別設定為p/8和p/16時，壓縮比例分別為4和8，重建圖片的質量明顯下降，圖像細節和清晰度都有所損失。

以下程式碼是一個簡單的圖像處理示例，它的作用是將一張圖片分為多個 64x64 大小的小區域，並以矩形標示出每個小區域的位置和位置信息，最後顯示其中一個小區域座標 (4,4) 的內容。

```
In [ ]: import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import skimage.util as skutil

imgfile = '/content/drive/MyDrive/淺度機器學習/lenna.png'
X = mpimg.imread(imgfile)
if len(X.shape) > 2:
    X = np.mean(X, axis=2)

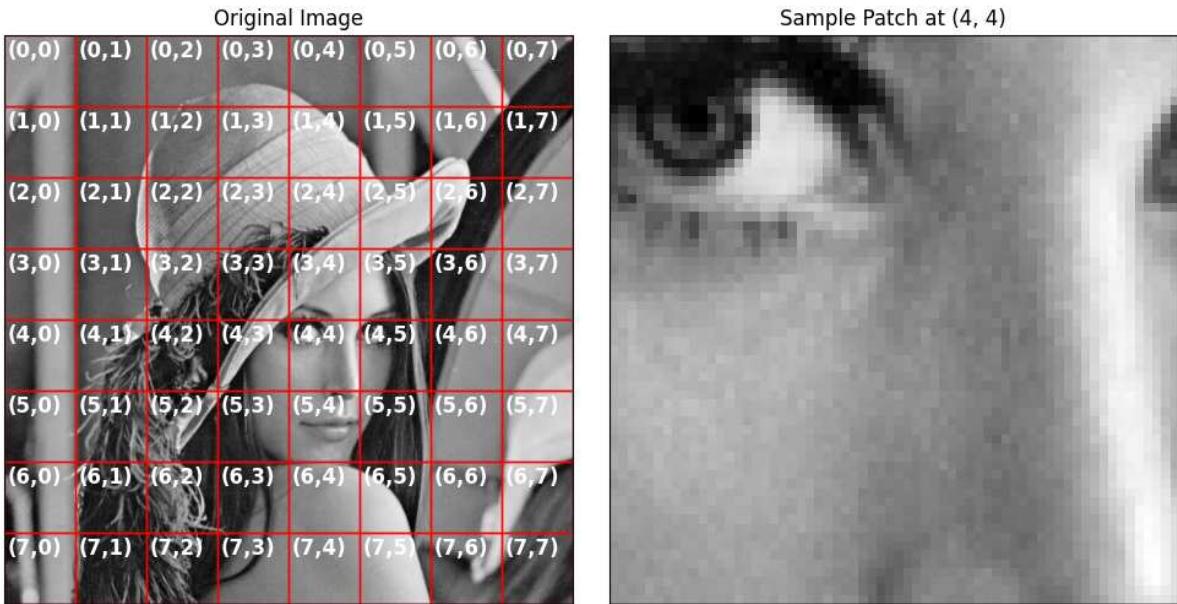
# 將圖像分為 16x16 的小區域
patches = skutil.view_as_windows(X, (64, 64), step=64)

# 顯示原始圖像和切割後的小區域
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].imshow(X, cmap='gray')
ax[0].set_title('Original Image')
ax[0].set_xticks([])
ax[0].set_yticks([])

# 繪製矩形標示出小圖的位置，並標示出位置信息
for i in range(patches.shape[0]):
    for j in range(patches.shape[1]):
        x1 = j * 64
        y1 = i * 64
        rect = plt.Rectangle((x1, y1), 64, 64,
                             edgecolor='r', facecolor='none')
        ax[0].add_patch(rect)
        ax[0].text(x1+2, y1+19, f'({i},{j})',
                   color='w', fontsize=12, fontweight='bold')

# 顯示其中一個小區域
ax[1].imshow(patches[4, 4], cmap='gray')
ax[1].set_title('Sample Patch at (4, 4)')
ax[1].set_xticks([])
ax[1].set_yticks([])

plt.tight_layout()
plt.show()
```



以下是程式碼中值得介紹的部分：

- 第12行：使用 `skutil.view_as_windows()` 函式將圖像分割成大小為 $64 \times 64$ 的小區域。這個函式可以簡化程式碼，減少迴圈的使用，使得程式更加整潔。
- 第22-28行：使用 `plt.Rectangle()` 函式在原圖上繪製紅色的矩形標示出每個小區域的位置。
- 第32-35行：這段程式碼可以透過更改座標來放大查看指定區域。右邊的圖像展示了其中一個小區域，由許多小正方形組成，每個小區域都是一個 $64 \times 64$ 的矩形。

## 1-2. 將 $X$ 以 $8 \times 8$ 小圖 ( **patch** ) 進行切割，再將每個小圖拉成 $64 \times 1$ 的向量，最後重組這些向量並排成新的 $64 \times N$ 矩陣。

```
In [ ]: import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
from numpy.linalg import svd

# 載入圖像並轉換為灰白圖
imgfile = '/content/drive/MyDrive/淺度機器學習/lenna.png'
X = mpimg.imread(imgfile)
if len(X.shape) > 2:
    X = np.mean(X, axis=2)

N, p = X.shape

# 切割圖像成為  $8 \times 8$  的小圖
patches = []
for i in range(64):
    for j in range(64):
        patch = X[i*8:(i+1)*8, j*8:(j+1)*8]
        patches.append(patch)

# 將每個小圖轉換成  $64 \times 1$  的向量，再將它們組成一個  $64 \times 4096$  的矩陣
X_blocks = np.array([patch.flatten() for patch in patches]).T

# 計算 SVD
```

```

U, E, VT = svd(X_blocks, full_matrices=False)

# 定義 q 值
q_vals = np.array([p/8, p/16, p/32]).astype('int')

# 繪製每個 q 值的近似圖像
fig, ax = plt.subplots(1, 3, figsize=(12, 4))
for i, q in enumerate(q_vals):
    # 計算 Rank-q 近似
    Xq = U[:, :q] @ np.diag(E[:q]) @ VT[:q, :]

    # 將近似圖像轉換回 8x8 的小圖
    approx_patches = [Xq[:, i].reshape(8, 8) for i in range(Xq.shape[1])]

    # 將小圖重新組合成圖像(採用 np.vstack 以及 np.hstack 組回原本的圖像大小 512x512)
    approx_image = np.vstack([np.hstack([approx_patches[j*64+i]
                                         for i in range(64)]) for j in range(64)])]

    # 繪製近似圖像
    ax[i].imshow(approx_image, cmap='gray')
    ax[i].set_title('Compression ratio: {}'.format(p/q/2))
    ax[i].set_xticks([])
    ax[i].set_yticks([])

plt.tight_layout()
plt.show()

```



這段程式碼展示將圖像切割成許多  $8 \times 8$  的小圖，並將每個小圖轉換為  $64 \times 1$  的向量，最終組成一個  $64 \times 4096$  的矩陣。接下來，對這個矩陣進行 SVD，得到左奇異矩陣  $U$ ，奇異值矩陣  $E$  和右奇異矩陣  $VT$ 。

對於每個  $q$  值，計算其對應的 Rank- $q$  近似圖像。具體來說，我們只保留  $U$  的前  $q$  列， $E$  的前  $q$  個對角元素，和  $VT$  的前  $q$  行，然後將它們相乘，得到一個  $64 \times 4096$  的矩陣  $Xq$ ，然後再將  $Xq$  轉換回  $8 \times 8$  的小圖，最後將小圖重新組合成圖像。

從觀察結果可以看出，在這個例子中，使用三個不同的壓縮倍數進行圖像壓縮後，壓縮後的圖像品質相似，並且仍然可以清晰地辨認圖像細節。

以下是程式碼中值得介紹的部分：

- 第15-19行：將原始圖像  $X$  切割成  $64 \times 64$  個  $8 \times 8$  的小圖，並將每個小圖存儲在  $patches$  列表中。其中，第一個迴圈遍歷圖像的垂直方向，第二個迴圈遍歷圖像的水平方向。
- 第37-41行：這段程式碼將壓縮後的圖像  $Xq$  中的每個向量重新  $reshape$  成  $8 \times 8$  的小圖，存儲在  $approx_patches$  列表中。然後，將  $approx_patches$  列表中的所有小圖重新組合成

一個大圖像 approx\_image，採用 np.vstack 和 np.hstack 函數組合圖像，以恢復原始圖像的大小，即  $512 \times 512$ 。

---

## 1-3. 同上，小圖大小為 $16 \times 16$ /per patch

```
In [ ]: import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
from numpy.linalg import svd

# 載入圖像並轉換為灰白圖
imgfile = '/content/drive/MyDrive/淺度機器學習/lenna.png'
X = mpimg.imread(imgfile)
if len(X.shape) > 2:
    X = np.mean(X, axis=2)

N, p = X.shape

# 切割圖像成為  $16 \times 16$  的小圖
patches = []
for i in range(32):
    for j in range(32):
        patch = X[i*16:(i+1)*16, j*16:(j+1)*16]
        patches.append(patch)

# 將每個小圖轉換成  $256 \times 1$  的向量，再將它們組成一個  $256 \times 1024$  的矩陣
X_blocks = np.array([patch.flatten() for patch in patches]).T

# 計算 SVD
U, E, VT = svd(X_blocks, full_matrices=False)

# 定義 q 值
q_vals = np.array([p/8, p/16, p/32]).astype('int')

# 繪製每個 q 值的近似圖像
fig, ax = plt.subplots(1, 3, figsize=(12, 4))
for i, q in enumerate(q_vals):
    # 計算 Rank-q 近似
    Xq = U[:, :q] @ np.diag(E[:q]) @ VT[:q, :]

    # 將近似圖像轉換回  $16 \times 16$  的小圖
    approx_patches = [Xq[:, i].reshape(16, 16) for i in range(Xq.shape[1])]

    # 將小圖重新組合成圖像（採用 np.block 組回原本的圖像大小  $512 \times 512$ ）
    approx_image = np.block([[approx_patches[j*32+i] for i in range(32)] for j in range(32)])

    # 繪製近似圖像
    ax[i].imshow(approx_image, cmap='gray')
    ax[i].set_title('Compression ratio: {}'.format(p/q/2))
    ax[i].set_xticks([])
    ax[i].set_yticks([])

plt.tight_layout()
plt.show()
```



透過觀察此例的結果，可以發現當小圖大小為  $16 \times 16$  時，若壓縮倍數為 16 時，圖片開始變模糊了，表示在此情況下，此壓縮率已經對圖像品質產生較明顯的影響。

## 1-4. 同上，小圖大小為 $32 \times 32$ /per patch

```
In [ ]: import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
from numpy.linalg import svd

# 載入圖像並轉換為灰白圖
imgfile = '/content/drive/MyDrive/淺度機器學習/lenna.png'
X = mpimg.imread(imgfile)
if len(X.shape) > 2:
    X = np.mean(X, axis=2)

N, p = X.shape

# 切割圖像成為  $32 \times 32$  的小圖
patches = []
for i in range(16):
    for j in range(16):
        patch = X[i*32:(i+1)*32, j*32:(j+1)*32]
        patches.append(patch)

# 將每個小圖轉換成  $1024 \times 1$  的向量，再將它們組成一個  $1024 \times 256$  的矩陣
X_blocks = np.array([patch.flatten() for patch in patches]).T

# 計算 SVD
U, E, VT = svd(X_blocks, full_matrices=False)

# 定義  $q$  值
q_vals = np.array([p/8, p/16, p/32]).astype('int')

# 繪製每個  $q$  值的近似圖像
fig, ax = plt.subplots(1, 3, figsize=(12, 4))
for i, q in enumerate(q_vals):
    # 計算 Rank- $q$  近似
    Xq = U[:, :q] @ np.diag(E[:q]) @ VT[:q, :]

    # 將近似圖像轉換回  $32 \times 32$  的小圖
    approx_patches = [Xq[:, i].reshape(32, 32) for i in range(Xq.shape[1])]

    # 將小圖重新組合成圖像
    approx_image = np.block([[approx_patches[j*16+i]
                             for i in range(16)] for j in range(16)])
```

```

# 繪製近似圖像
ax[i].imshow(approx_image, cmap='gray')
ax[i].set_title('Compression ratio: {}'.format(p/q/2))
ax[i].set_xticks([])
ax[i].set_yticks([])

plt.tight_layout()
plt.show()

```



透過觀察此例的結果，可以發現當小圖大小為 32x32 時，若壓縮倍數為 8 或 16 時，圖片都開始變模糊了，表示在此情況下，這些壓縮率已經對圖像品質產生較明顯的影響。

## 1-5. 使用相同 SVD Rank-q 值比較三種小圖大小的影像降維效果

固定 Rank-q 的情況下，一次比較三種不同大小的小圖可以更清晰地顯示它們之間的異同，有助於我們更深入地了解它們對影像降維的影響。

```

In [ ]: import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
from numpy.linalg import svd
import skimage.util as skutil

# 載入圖像並轉換為灰白圖
imgfile = '/content/drive/MyDrive/淺度機器學習/lenna.png'
X = mpimg.imread(imgfile)
if len(X.shape) > 2:
    X = np.mean(X, axis=2)

N, p = X.shape

# 定義不同的小圖大小
patch_sizes = [(8, 8), (16, 16), (32, 32)]

# 設定 q 值
q = p // 32

# 繪製每個小圖大小的近似圖像
fig, ax = plt.subplots(1, len(patch_sizes)+1, figsize=(12, 4))
for i, patch_size in enumerate(patch_sizes):
    # 將圖像分為指定大小的小區域
    patches = skutil.view_as_windows(X, patch_size, step=patch_size[0])

    patches = patches.reshape((-1, patch_size[0] * patch_size[1])).T

    # 計算 SVD
    U, S, V = svd(patches)
    approx_image = U[:, :q] @ S[:q] @ V[:q, :]

    ax[i].imshow(approx_image, cmap='gray')
    ax[i].set_title('Compression ratio: {}'.format(p/q/2))

plt.tight_layout()
plt.show()

```

```

U, E, VT = svd(patches, full_matrices=False)

# 計算 Rank-q 近似
Xq = U[:, :q] @ np.diag(E[:q]) @ VT[:q, :]

# 將近似圖像轉換回原本大小的小圖
approx_patches = [Xq[:, i].reshape(patch_size) for i in range(Xq.shape[1])]

# 將小圖重新組合成圖像
approx_image = np.block([[approx_patches[j*(p // patch_size[0])+i]
                           for i in range(p // patch_size[0])] for
                           j in range(p // patch_size[1])])

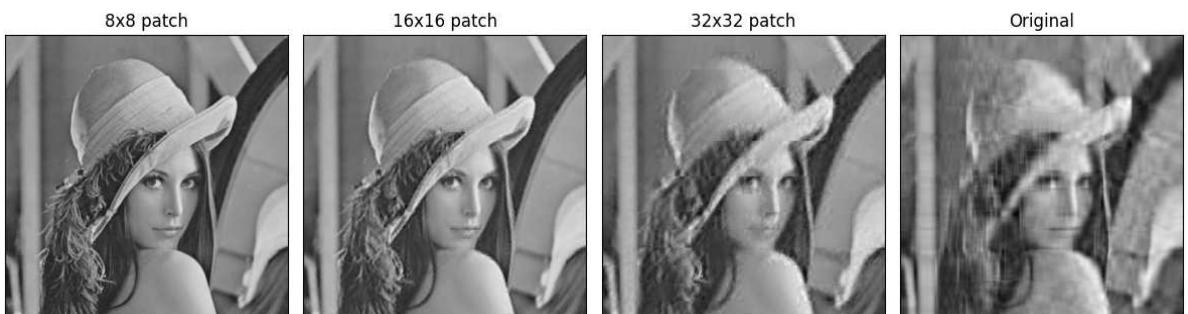
# 繪製近似圖像
ax[i].imshow(approx_image, cmap='gray')
ax[i].set_title('{0}x{1} patch'.format(patch_size[0], patch_size[1]))
ax[i].set_xticks([])
ax[i].set_yticks([])

#####Original#####
U, E, VT = svd(X, full_matrices = False)
i = 3
Xq = U[:, :q] @ np.diag(E[:q]) @ VT[:q, :]
ax[i].imshow(Xq, cmap = 'gray')
ax[i].set_title('Original')
ax[i].set_xticks([])
ax[i].set_yticks([])

fig.suptitle('Comparison of Rank-q (q = 16) Approximations of Lena Image with Different Patch Sizes')
plt.tight_layout()
plt.show()

```

**Comparison of Rank-q (q = 16) Approximations of Lena Image with Different Patch Sizes**



從結果可以看出，用小的小圖大小（8x8）會產生更好的近似圖像，因為它能夠更有效地捕捉圖像的局部細節。隨著小圖大小的增加（16x16 和 32x32），近似圖像的細節逐漸減少，但整體還是能夠保留圖像的主要特徵。總之，以上三種使用不同小圖大小的方法都能夠產生比不切小圖更好的近似圖像。

值得注意的是，雖然使用較小的小圖大小可以產生更好的近似圖像，但也意味著需要更多的小圖來重新組合成完整的圖像。這可能會增加運行時間，因為需要計算更多的SVD和矩陣乘法。

## 1-6. 使用相異 SVD Rank-q 值比較三種小圖大小的影像降維效果

```
In [ ]: import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
```

```

from numpy.linalg import svd
import skimage.util as skutil

# 載入圖像並轉換為灰白圖
imgfile = '/content/drive/MyDrive/淺度機器學習/lenna.png'
X = mpimg.imread(imgfile)
if len(X.shape) > 2:
    X = np.mean(X, axis=2)

N, p = X.shape

# 定義不同的小圖大小
patch_sizes = [(8, 8), (16, 16), (32, 32)]

# 設定不同的 q 值
q_vals = np.array([p/8, p/16, p/32]).astype('int')

# 繪製每個小圖大小和 q 值的近似圖像
fig, ax = plt.subplots(len(q_vals), len(patch_sizes), figsize=(11, 11))
for i, q in enumerate(q_vals):
    for j, patch_size in enumerate(patch_sizes):
        # 將圖像分為指定大小的小區域
        patches = skutil.view_as_windows(X, patch_size, step=patch_size[0])
        # 重新整理形狀以便後續處理
        patches = patches.reshape((-1, patch_size[0] * patch_size[1])).T

        # 計算 SVD
        U, E, VT = svd(patches, full_matrices=False)

        # 計算 Rank-q 近似
        Xq = U[:, :q] @ np.diag(E[:q]) @ VT[:q, :]

        # 將近似圖像轉換回原本大小的小圖
        approx_patches = [Xq[:, i].reshape(patch_size) for i in range(Xq.shape[1])]

        # 將小圖重新組合成圖像
        approx_image = np.block([[approx_patches[j*(p // patch_size[0])+i]
                                 for i in range(p // patch_size[0])]
                                for j in range(p // patch_size[1])])

        # 繪製近似圖像
        ax[i, j].imshow(approx_image, cmap='gray')
        ax[i, j].set_title('{0}x{1} patch, Compression ratio: {2}'.
                           format(patch_size[0], patch_size[1], p/q/2))
        ax[i, j].set_xticks([])
        ax[i, j].set_yticks([])

fig.suptitle('Comparison of Rank-q Approximations of Lena Image
with Different Patch Sizes', y=1.0, fontweight='bold', fontsize=13)
plt.tight_layout()
plt.show()

```

### Comparison of Rank-q Approximations of Lena Image with Different Patch Sizes



1. 小圖大小 8x8, q 值為 p/8: 這個設定下，使用了最小的小圖大小，且 q 值也是最大的，因此能夠更精確地捕捉圖像的局部細節，所以生成的近似圖像質量最高。然而，由於 q 值較大，所以也需要較長的計算時間。

1. 小圖大小 16x16, q 值為 p/16: 在這個設定下，使用的小圖大小和 q 值都比上一個設定要小，因此生成的近似圖像會稍微失去一些細節，但整體仍能夠保留圖像的主要特徵。這個設定下的計算時間也會比第一個設定短一些。
2. 小圖大小 32x32, q 值為 p/32: 在這個設定下，使用最大的小圖大小，且 q 值也最小，因此生成的近似圖像會失去更多的細節，但仍然能夠辨認圖像的主要特徵。這個設定下計算時間最短。

總體而言，隨著小圖大小的增加，近似圖像的細節逐漸減少，但整體還是能夠保留圖像的主要特徵。隨著 q 值的減少，近似圖像的細節會更多地丟失，但計算時間也會減少。根據需要，可以選擇不同的小圖大小和 q 值來平衡圖像質量和計算時間。

---

**習題 2：處理大量影像前，有必要觀看影像圖，以確定能掌握將要處理的影像及其資料型態。以 70000 張手**

寫圖像為例，每個數字約 7000 字，需要寫一段程式碼來觀察這些手寫數字的影像與品質，且每次執行都能隨機觀看到不同的影像。

---

## 套件讀取

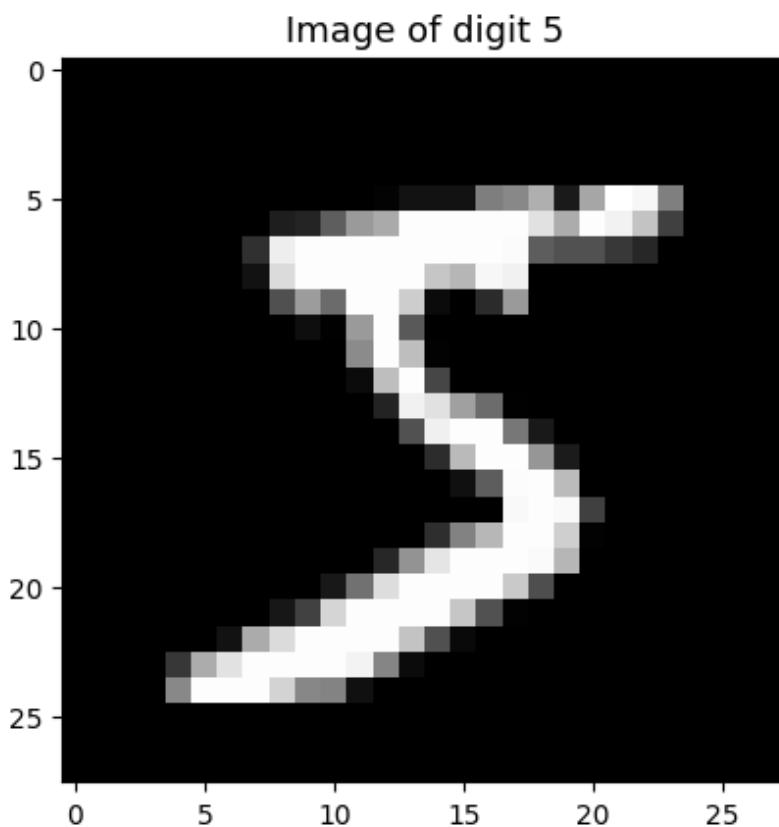
```
In [ ]: import matplotlib.pyplot as plt  
import numpy as np
```

匯入 MNIST 資料集，其中包含手寫數字的影像資料。

```
In [ ]: from sklearn.datasets import fetch_openml  
X, y = fetch_openml('mnist_784', parser = 'auto', \  
return_X_y = True)  
X = X.T
```

嘗試讀取資料集中的一張影像。

```
In [ ]: i = 0  
img = X.iloc[:, i]  
sz = np.sqrt(len(img)).astype('int')  
plt.imshow(np.array(img).reshape(sz, sz), cmap='gray')  
plt.title(f"Image of digit 5", fontsize = 13)  
plt.show()
```



其中 'sz = np.sqrt(len(img)).astype('int')' 計算每張影像的邊長 sz，這裡假設每張影像都是正方形，所以 sz 的值是向量的長度開根號後取整數。

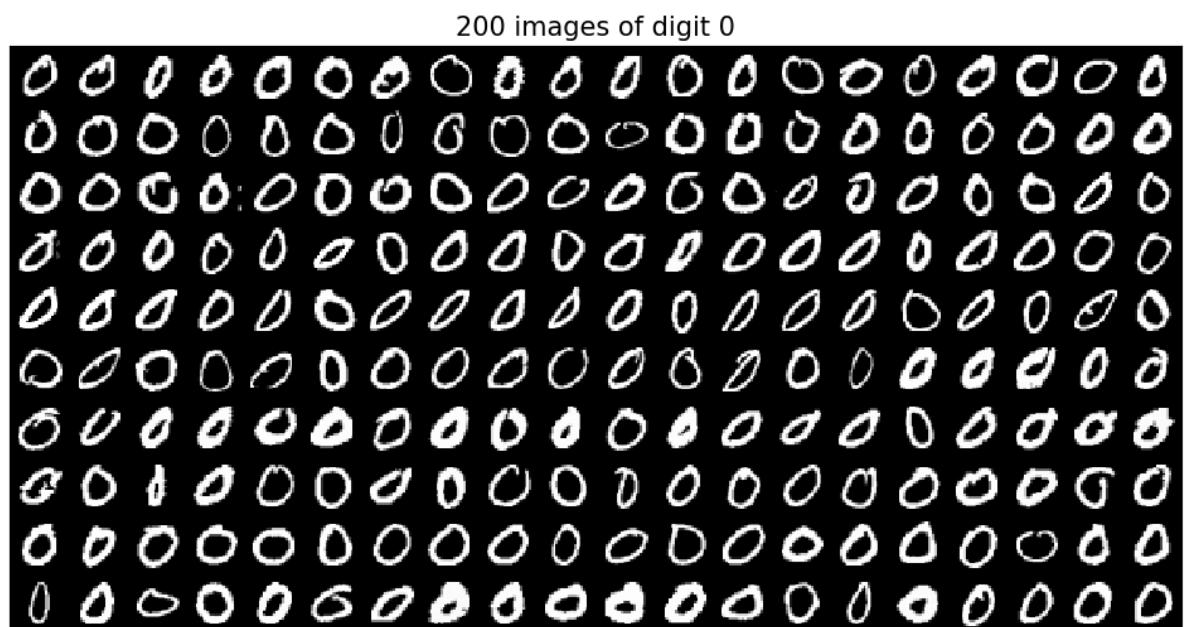
## Montage函式：將多張數字圖像拼接成一個大的「蒙太奇」圖像。

(引用自 SML/Lesson 6: 淺度機器學習：PCA、SVD 及其在影像處理的應用 汪群超 Chun-Chao Wang 老師)

```
In [ ]: def montage(A, m, n):
    ...
    Create a montage matrix with mn images
    Inputs:
    A: original pxN image matrix with N images (p pixels), N > mn
    m, n: m rows & n columns, total mn images
    Output:
    M: montage matrix containing mn images
    ...
    sz = np.sqrt(A.shape[0]).astype('int') # image size sz x sz
    M = np.zeros((m*sz, n*sz)) # montage image
    for i in range(m):
        for j in range(n):
            M[i*sz:(i+1)*sz, j*sz:(j+1)*sz] = \
                A[:, i*n+j].reshape(sz, sz)
    return M
```

### 繪製多張特定數字的蒙太奇圖像

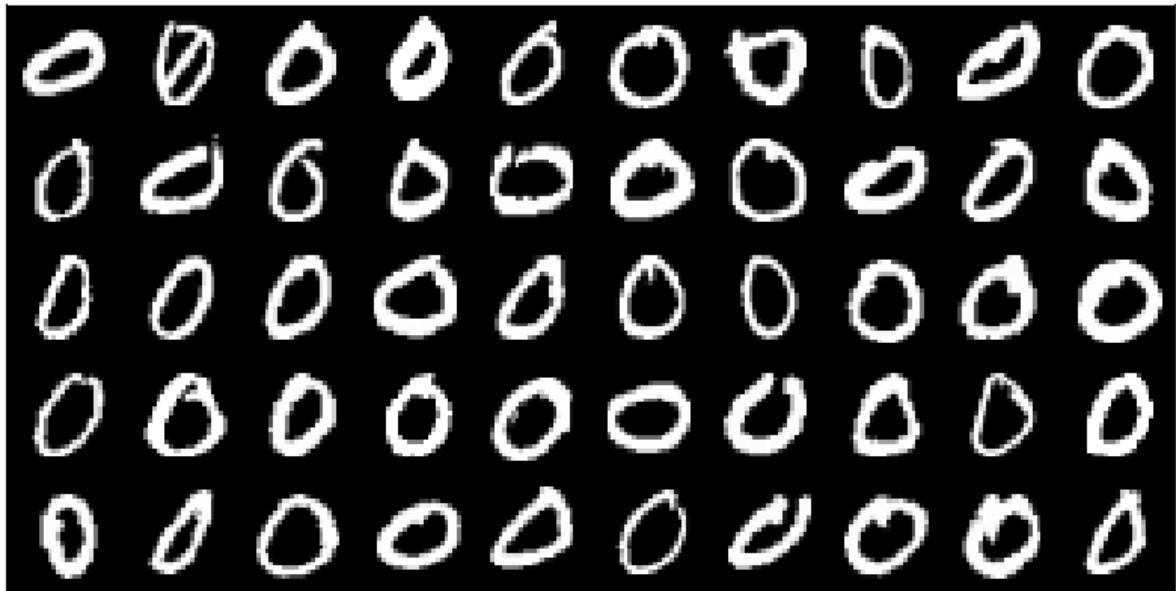
```
In [ ]: digit_to_show = '0'
idx = y[y==digit_to_show].index
Digit = X.iloc[:, idx]
plt.figure(figsize = (12, 12))
m, n = 10, 20 # A m x n montage (total mn images)
M = montage(np.array(Digit), m, n)
plt.imshow(M, cmap = 'gray', interpolation = 'nearest')
plt.xticks([])
plt.yticks([])
plt.title(f'{m*n} images of digit {digit_to_show}', fontsize = 15)
plt.show()
```



### 繪製多張特定數字的隨機蒙太奇圖像

```
In [ ]: digit_to_show = '0'
idx = np.random.choice(y[y==digit_to_show].index, size=50, replace=False)
# 隨機選擇50個該數字的索引
Digit = X.iloc[:, idx]
plt.figure(figsize = (10, 10))
m, n = 5, 10 # A m x n montage (total mn images)
M = montage(np.array(Digit), m, n)
plt.imshow(M, cmap = 'gray', interpolation = 'nearest')
plt.xticks([])
plt.yticks([])
plt.title(f"{m*n} random images of digit {digit_to_show}", fontsize = 15)
plt.show()
```

50 random images of digit 0



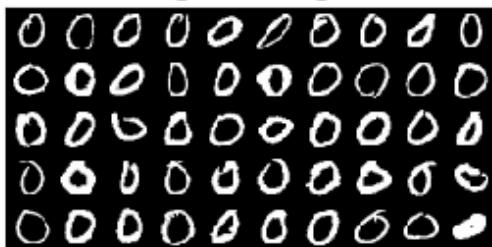
此程式碼與前一個類似，僅修改 `idx = y[y==digit_to_show].index` 為 `idx = np.random.choice(y[y==digit_to_show].index, size=50, replace=False)`。

這代表它會在 `y` 中等於 `digit_to_show` 的索引中隨機選擇50個索引，並且不會重複選擇。

## 繪製多張數字0~9的隨機蒙太奇圖像

```
In [ ]: plt.figure(figsize=(10,10))
for i in range(10):
    digit_to_show = str(i)
    idx = np.random.choice(y[y==digit_to_show].index, size=50, replace=False)
    # 隨機選擇50個該數字的索引
    Digit = X.iloc[:, idx]
    m, n = 5, 10 # A m x n montage (total mn images)
    M = montage(np.array(Digit), m, n)
    plt.subplot(5, 2, i+1)
    plt.imshow(M, cmap='gray', interpolation='nearest')
    plt.xticks([])
    plt.yticks([])
    plt.title(f"Images of digit {digit_to_show}")
plt.subplots_adjust(hspace=0.3, wspace=-0.3)
plt.show()
```

Images of digit 0



Images of digit 1



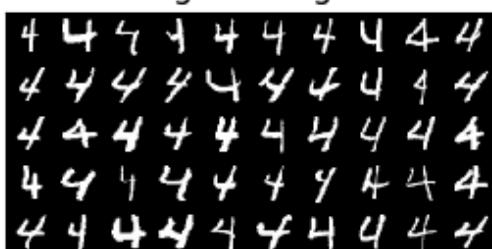
Images of digit 2



Images of digit 3



Images of digit 4



Images of digit 5



Images of digit 6



Images of digit 7



Images of digit 8



Images of digit 9



習題三：每張大小  $28 \times 28$  的手寫數字圖像 70000 張，不經壓縮前的儲存空間為 54.88 M Bytes。若進行 SVD 的“Rank q approximation”，則壓縮倍數由 q 決定。寫一支程式，當調整 q 值時，可以算出壓縮的倍數，並同時顯示原圖與壓縮後還原的圖各 100 張做為比較（任選 100 張）。

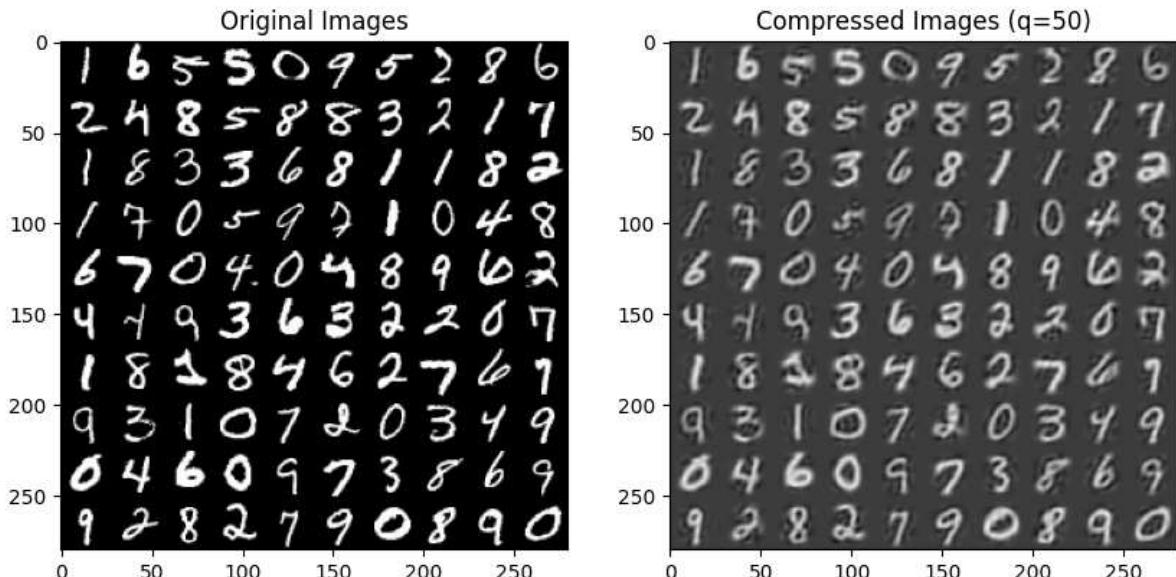
```
In [ ]:
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat

mnist = loadmat('/content/drive/MyDrive/淺度機器學習/mnist-original.mat')
X = mnist['data']
y = mnist['label'][0]

# 進行 SVD 壓縮
q = 50 # 設定壓縮後保留的前 q 個奇異值
U, s, V_T = np.linalg.svd(X, full_matrices=False)
S = np.diag(s[:q])
U_q = U[:, :q]
V_T_q = V_T[:q, :]
X_T_q = U_q @ S @ V_T_q

# 隨機選取 100 張圖進行壓縮與比較
indices = np.random.choice(range(X.shape[1]), size=100, replace=False)
original_images = X[:, indices]
compressed_images = X_T_q[:, indices]

# 將原始圖像與壓縮圖像排列成左右兩張子圖
fig, axs = plt.subplots(1, 2, figsize=(10, 10))
axs[0].imshow(montage(original_images, 10, 10), cmap='gray')
axs[0].set_title('Original Images')
axs[1].imshow(montage(compressed_images, 10, 10), cmap='gray')
axs[1].set_title(f'Compressed Images (q={q})')
plt.show()
```



這個程式展示了如何使用奇異值分解(SVD)來進行圖像壓縮。在程式中，我們使用了MNIST手寫數字資料集，先進行SVD壓縮，然後隨機選取100張圖像進行壓縮與還原，比較原始圖像與壓縮圖像的差異。

## 使用SVD進行MNIST圖像壓縮與比較：保留前50個奇異值

```
In [ ]:
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
```

```

def compress_images(X, q):
    """對矩陣X進行SVD壓縮，保留前q個奇異值"""
    U, s, V_T = np.linalg.svd(X, full_matrices=False)
    S = np.diag(s[:q])
    U_q = U[:, :q]
    V_T_q = V_T[:q, :]
    X_T_q = U_q @ S @ V_T_q
    total_energy = np.sum(s ** 2)
    energy_ratio = np.sum(s[:q] ** 2) / total_energy
    return X_T_q, energy_ratio

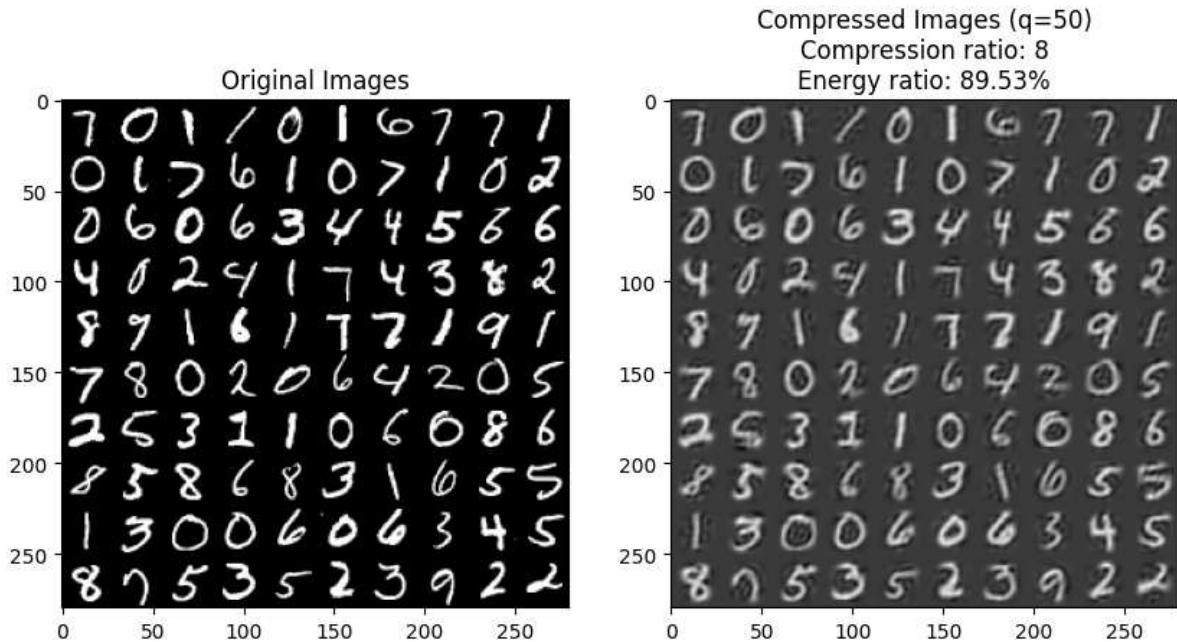
# 載入MNIST數據集
mnist = loadmat('/content/drive/MyDrive/淺度機器學習/mnist-original.mat')
X = mnist['data']
y = mnist['label'][0]
original_size = 784
q = 50

# 對矩陣X進行SVD壓縮
X_T_q, energy_ratio = compress_images(X, q)

# 隨機選取100張圖進行壓縮與比較
indices = np.random.choice(range(X.shape[1]), size=100, replace=False)
original_images = X[:, indices]
compressed_images = X_T_q[:, indices]

# 將原始圖像與壓縮圖像排列成左右兩張子圖
fig, axs = plt.subplots(1, 2, figsize=(10, 10))
axs[0].imshow(montage(original_images, 10, 10), cmap='gray')
axs[0].set_title('Original Images')
axs[1].imshow(montage(compressed_images, 10, 10), cmap='gray')
axs[1].set_title(f'Compressed Images (q={q})\nCompression ratio:\n{original_size/q/2:.0f}\nEnergy ratio: {energy_ratio:.2%}')
plt.show()

```



假設有一個  $m \times n$  的矩陣  $A$ ，其奇異值分解為  $A = U\Sigma V^T$ ，我們可以用奇異值來計算矩陣  $A$  的能量佔比。假設矩陣  $A$  的前  $q$  個奇異值分別為  $\sigma_1, \sigma_2, \dots, \sigma_q$ ，則矩陣  $A$  的能量佔比可以計算為 (前  $q$  個奇異值的平方和) / (所有奇異值的平方和) 計算公式如下：

奇異值計算能量佔比的數學公式

$$\frac{\sum_{i=1}^q \sigma_i^2}{\sum_{i=1}^{\min(m,n)} \sigma_i^2}$$

當我們將  $q$  設為 50 時，可以將原始圖像進行壓縮。在本例中，壓縮倍數約為 8，表示我們可以將圖像的大小縮小到原來的 8 倍。同時，根據奇異值分解的結果，我們可以發現這些前  $q$  個奇異值可以解釋圖像的能量占比為 89.53%。因此，我們可以在保留圖像主要特徵的前提下，將圖像的大小大幅度縮小。

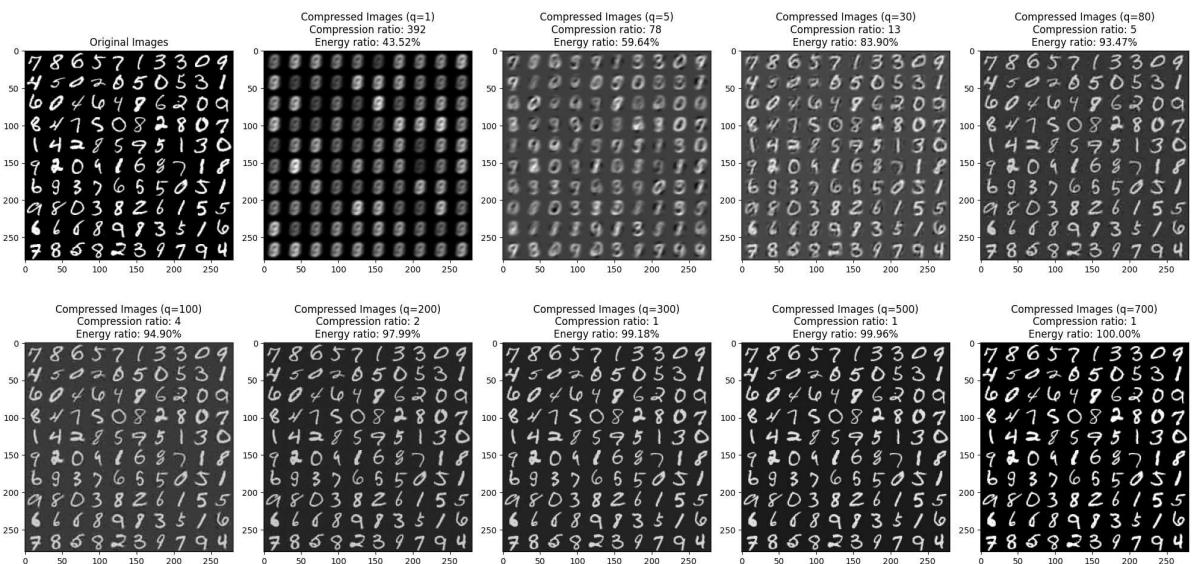
## 探索不同奇異值數量對MNIST圖像壓縮的影響

```
In [ ]: fig, axs = plt.subplots(2, 5, figsize=(20, 10))

q_values = [1, 5, 30, 80, 100, 200, 300, 500, 700]
# Loop through the rest of the subplots and plot compressed images
for i, q in enumerate(q_values):
    # 對全局變量X進行SVD壓縮
    X_T_q, energy_ratio = compress_images(q)

    # 隨機選取100張圖進行壓縮與比較
    np.random.seed(42)
    indices = np.random.choice(range(X.shape[1]), size=100, replace=False)
    compressed_images = X_T_q[:, indices]

    # 將原始圖像與壓縮圖像排列成左右兩張子圖
    row = (i+1) // 5
    col = (i+1) % 5
    compressed_ax = axs[row][col]
    compressed_ax.imshow(montage(compressed_images, 10, 10), cmap='gray')
    compressed_ax.set_title(f'Compressed Images (q={q})\nCompression ratio: {original_size/q/2:.0f}\nEnergy ratio: {energy_ratio:.2%}')
    axs[0][0].imshow(montage(original_images, 10, 10), cmap='gray')
    axs[0][0].set_title('Original Images')
    plt.tight_layout()
plt.show()
```



對於每個  $q$  值，我們隨機選取了 100 個數字圖像進行壓縮和比較。結果表明，隨著  $q$  值的增加，壓縮後的數字圖像品質得到了提高。此外，我們還觀察到，每個  $q$  值對應的能量佔比也不同，當  $q$  值為 30 時，已經達到了 83.90% 的能量佔比。這意味著我們只需要保留原始奇異值的 30 個左右，就可以還原出大部分原始圖像的信息。

這個結果表明SVD壓縮是一種非常有效的圖像壓縮方法，可以大幅減少圖像的存儲空間，同時保持圖像的關鍵信息。

---

**習題四：有 5 張經過加密的影像圖（壓縮檔下載），其加密的方式採 Yale Faces 38 人 2410 張人臉圖像矩陣  $X$  的 SVD，即  $X = U\Sigma V^T$ ，取  $U$  作為影像加密的工具，即假設向量  $x$  代表一張原圖影像，則  $U[:, 0 : q]^T x$  代表該影像的前  $q$  個主成分，以此作為加密影像。**

```
In [ ]: import numpy as np
import scipy.io
D = scipy.io.loadmat('/content/drive/MyDrive/淺度機器學習/allFaces.mat')
X = D['faces'] # 32256 x 2410, each column represents an image
y = np.ndarray.flatten(D['nfaces'])
m = int(D['m']) # 168
n = int(D['n']) # 192
n_persons = int(D['person']) # 38
```

## 載入加密資料以及Yale Faces

```
In [ ]: import numpy as np
import scipy.io
import pandas as pd
from numpy.linalg import svd

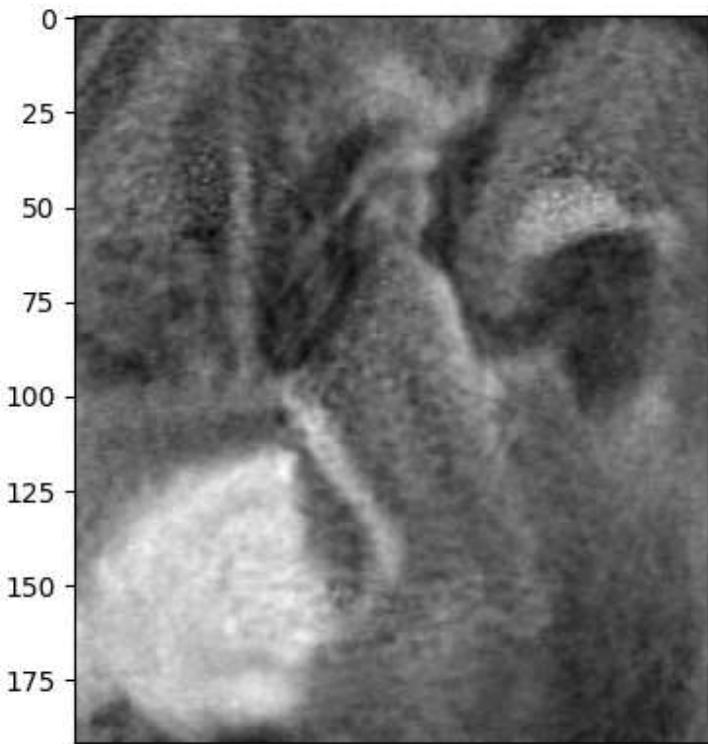
# 載入加密資料
encrypted_data = pd.read_csv('/content/drive/MyDrive/淺度機器學習/五張加密的影像.csv')

# 載入人臉影像資料
D = scipy.io.loadmat('/content/drive/MyDrive/淺度機器學習/allFaces.mat')
X = D['faces'] # 32256 x 2410, each column represents an image
```

## 展示加密圖像的第五張圖

```
In [ ]: fist_img = np.array(encrypted_data)[:,4]
# plt.imshow(fist_img.reshape(50,40),cmap='gray')
plt.xticks([])
# U, Sigma, Vt = np.linalg.svd(X, full_matrices=False)
avg_face = X.mean(axis=1).reshape(-1,1)
X_avg = X-np.tile(avg_face,(1,X.shape[1]))
U,E,VT = svd(X_avg,full_matrices=False)
q = 2000
Uq = U[:, 0:q]
Xq = np.dot(Uq, fist_img)
plt.imshow(Xq.reshape(168, 192).T, cmap='gray')
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x7ff12d5c2ca0>
```



## 展示所有加密圖像

```
In [ ]: import numpy as np
import scipy.io
import matplotlib.pyplot as plt
# 載入加密資料
encrypted_data = np.genfromtxt('/content/drive/MyDrive/淺度機器學習/五張加密的影像.csv', delimiter=',', skip_header=1)

# 載入人臉影像資料
D = scipy.io.loadmat('/content/drive/MyDrive/淺度機器學習/allFaces.mat')
X = D['faces']

# 設置解密參數 q
q = 2000

# 使用原始資料的 SVD 分解矩陣 U 和 Sigma
# U, Sigma, Vt = np.linalg.svd(X, full_matrices=False)
avg_face = X.mean(axis=1).reshape(-1, 1)
X_avg = X - np.tile(avg_face, (1, X.shape[1]))
# U, E, VT = svd(X_avg, full_matrices=False)
# 解密每個加密影像
decrypted_images = []
for i in range(encrypted_data.shape[1]):
    # 取出第 i 個加密影像，並將其轉換為列向量
    encrypted_image = encrypted_data[:, i].reshape((-1, 1))
    # 使用 U 的前 q 個主成分與加密影像進行矩陣乘法，得到該影像的近似原圖影像
    approx_image = np.dot(U[:, :q], encrypted_image)
    # 將近似原圖影像重組成圖像矩陣
    decrypted_image = approx_image.reshape((168, 192)).T
    # 將解密後的影像加入列表中
    decrypted_images.append(decrypted_image)

# 顯示解密後的圖像

plt.figure(figsize=(24, 12), dpi=100)
fig, axes = plt.subplots(nrows=1, ncols=5, figsize=(15, 15))
```

```

for i, decrypted_image in enumerate(decrypted_images):
    axes[i].imshow(decrypted_image, cmap='gray')
    axes[i].set_xticks([])
    axes[i].set_yticks([])
plt.show()

```

<Figure size 2400x1200 with 0 Axes>



這段程式碼是對五張加密的影像進行解密，解密的過程與加密過程類似。首先使用原始資料的奇異值分解矩陣  $U$ ，並計算出平均人臉像素  $\text{avg\_face}$  和  $X_{\text{avg}}$ 。接著，對每個加密影像進行解密，取出該影像的加密向量，並使用  $U$  的前  $q$  個主成分與加密向量進行矩陣乘法，得到該影像的近似原圖影像。最後，將近似原圖影像重組成圖像矩陣顯示出來。

**習題五：自行找 5 張照片（大小必須同 Yale Faces 的  $192 \times 168$  或自行 Resize），含人臉、水果、風景 ... 等進行加密後（ $q$  自選），再解密，觀察這些解密後的影像的效果，是否人臉的表現比較好？其他非人臉影像，如風景影像，能透過由人臉建構的特徵  $U$  加密嗎？（即解密後能否看到原圖模樣？）**

```

In [ ]: import matplotlib.pyplot as plt
from matplotlib.image import imread

imgfiles = ['/content/drive/MyDrive/淺度機器學習/Nick Young.jpg',
            '/content/drive/MyDrive/淺度機器學習/big north.jpeg',
            '/content/drive/MyDrive/淺度機器學習/guava.jpg',
            '/content/drive/MyDrive/淺度機器學習/fat tiger.jpeg',
            '/content/drive/MyDrive/淺度機器學習/windows.PNG']

titles = ['Nick Young (Basketball Player)',
          'Big North (Mascot of Taipei University)', 'Guava',
          'Takeshi Gouda', 'XP Bliss']

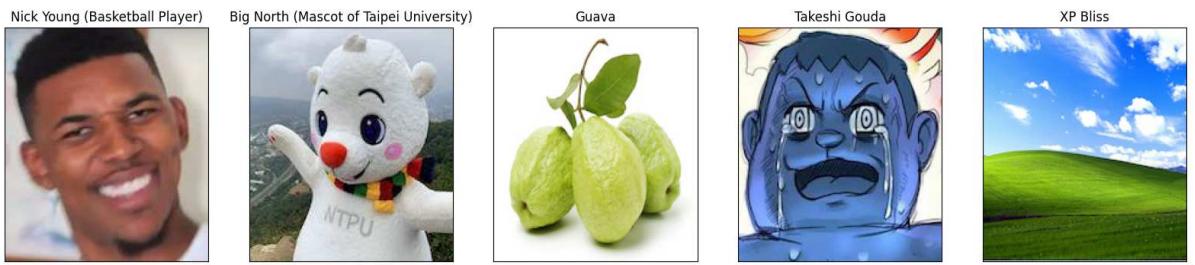
fig, axs = plt.subplots(nrows=1, ncols=5, figsize=(20, 5))

for i, (imgfile, title) in enumerate(zip(imgfiles, titles)):

    img = imread(imgfile)
    axs[i].imshow(img)
    axs[i].set_title(title)
    axs[i].set_xticks([])
    axs[i].set_yticks([])

plt.show()

```



```
In [ ]: import matplotlib.image as mpimg

fig, axs = plt.subplots(nrows=1, ncols=5, figsize=(20, 5))

for i, (imgfile, title) in enumerate(zip(imgfiles, titles)):

    X = mpimg.imread(imgfile)

    if len(X.shape) > 2:

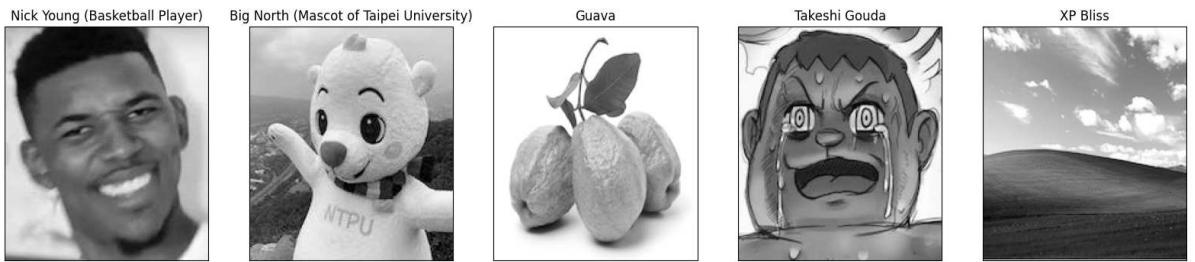
        X = np.mean(X, axis=2)

    axs[i].imshow(X, cmap='gray')

    axs[i].set_title(title)

    axs[i].set_xticks([])
    axs[i].set_yticks([])

plt.show()
```



## 影像加密與解密比較 - 使用Nick Young的人像加密其他圖像

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

def encrypt_decrypt(X, q, key):
    U, s, Vt = np.linalg.svd(key, full_matrices=False)
    U_q = U[:, :q]
    z = U_q.T @ X
    X_reconstructed = U_q @ U_q.T @ X
    return X_reconstructed

def compare_images(key, X_list, q_list):
    n_images = len(X_list)
    fig, axs = plt.subplots(n_images, len(q_list)+1, figsize=(2*(len(q_list)+1), 2*n_images))
    for i, X in enumerate(X_list):
        axs[i][0].imshow(X, cmap='gray')
        axs[i][0].axis('off')
        axs[i][0].set_title('Original')
        for j, q in enumerate(q_list):
            X_q = encrypt_decrypt(X, q, key)
            axs[i][j+1].imshow(X_q, cmap='gray')
```

```
        encrypted = encrypt_decrypt(X, q, key)
        decrypted = encrypt_decrypt(encrypted, q, key)
        axs[i][j+1].imshow(decrypted, cmap='gray')
        axs[i][j+1].axis('off')
        axs[i][j+1].set_title(f'q={q}')
        axs[i][j+1].set_xlabel(f'{np.sum((X-decrypted)**2):.2f}')
```

```
plt.tight_layout()
plt.show()
```

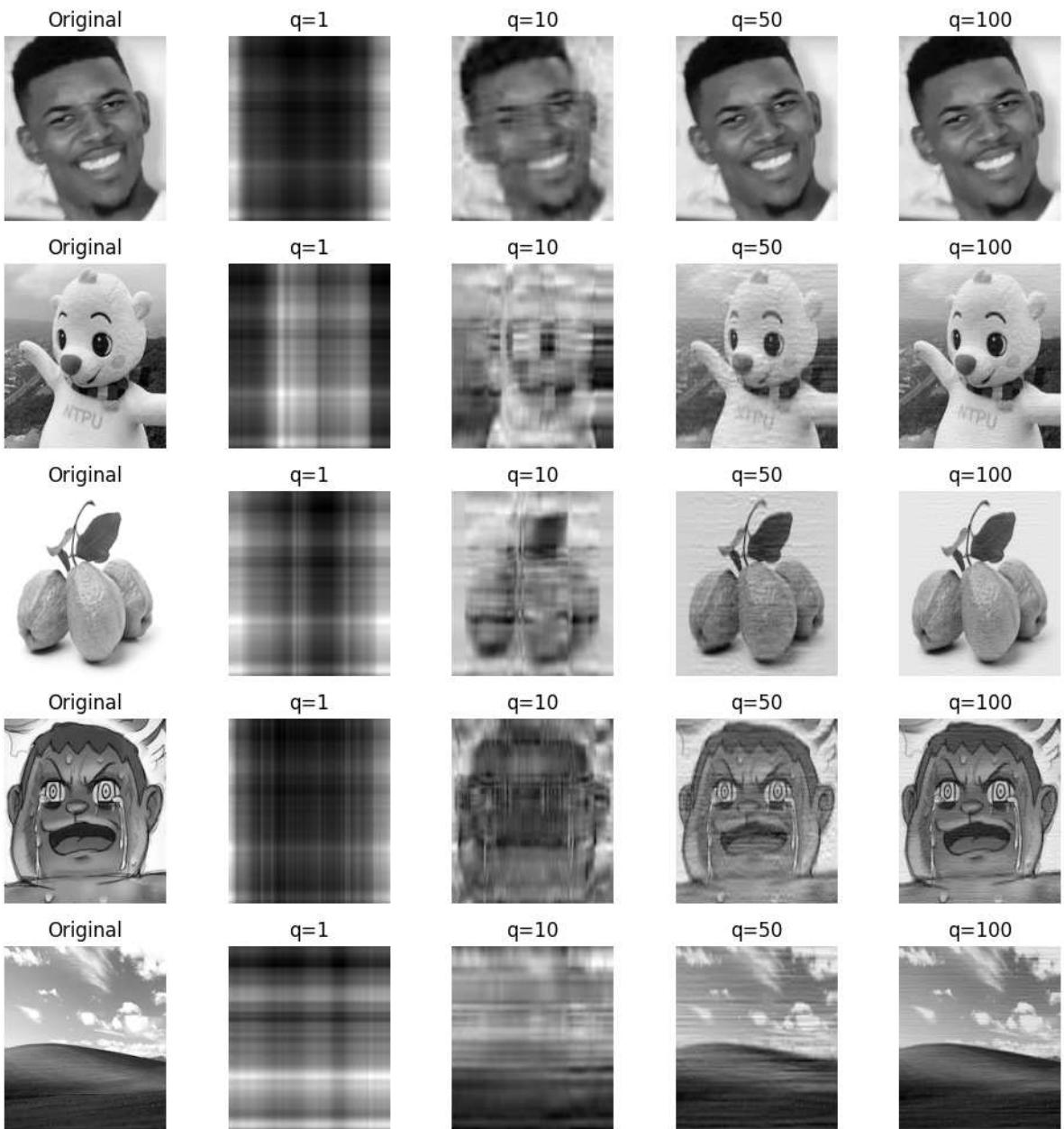
```
imgfiles = [ '/content/drive/MyDrive/淺度機器學習/Nick Young.jpg',
             '/content/drive/MyDrive/淺度機器學習/big north.jpeg',
             '/content/drive/MyDrive/淺度機器學習/guava.jpg',
             '/content/drive/MyDrive/淺度機器學習/fat tiger.jpeg',
             '/content/drive/MyDrive/淺度機器學習/windows.PNG' ]
```

```
X_list = []
for imgfile in imgfiles:
    X = mpimg.imread(imgfile)
    if len(X.shape) > 2:
        X = np.mean(X, axis=2)
    X_list.append(X)

key = X_list[0]

q_list = [1, 10, 50, 100]

compare_images(key, X_list, q_list)
```



觀察結果可以發現，在這些影像中，人臉影像的表現確實比其他影像好。這是因為人臉具有較高的可辨識性和較明顯的邊緣特徵，而這些特徵能夠被 SVD 分解中的特徵向量捕獲和重建。

對於其他非人臉影像，如風景影像，由人臉建構的特徵 U 可能無法捕捉到其特定特徵，因此無法實現完美加密和解密。但是，在一定程度上，使用人臉的特徵 U 可以實現一定程度的加密和解密，並且可以重建原始影像的大部分細節。在影像品質和信息保密性之間需要進行權衡。

## 影像加密與解密比較 - 使用 XP Bliss 風景圖加密其他圖像

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

def encrypt_decrypt(X, q, key):
    U, s, Vt = np.linalg.svd(key, full_matrices=False)
    U_q = U[:, :q]
    z = U_q.T @ X
    X_reconstructed = U_q @ U_q.T @ X
```

```

    return X_reconstructed

def compare_images(key, X_list, q_list):
    n_images = len(X_list)
    fig, axs = plt.subplots(n_images, len(q_list)+1, figsize=(2*(len(q_list)+1), 2*n_
    for i, X in enumerate(X_list):
        axs[i][0].imshow(X, cmap='gray')
        axs[i][0].axis('off')
        axs[i][0].set_title('Original')
        for j, q in enumerate(q_list):
            encrypted = encrypt_decrypt(X, q, key)
            decrypted = encrypt_decrypt(encrypted, q, key)
            axs[i][j+1].imshow(decrypted, cmap='gray')
            axs[i][j+1].axis('off')
            axs[i][j+1].set_title(f'q={q}')
            axs[i][j+1].set_xlabel(f'{np.sum((X-decrypted)**2):.2f}')
    plt.tight_layout()
    plt.show()

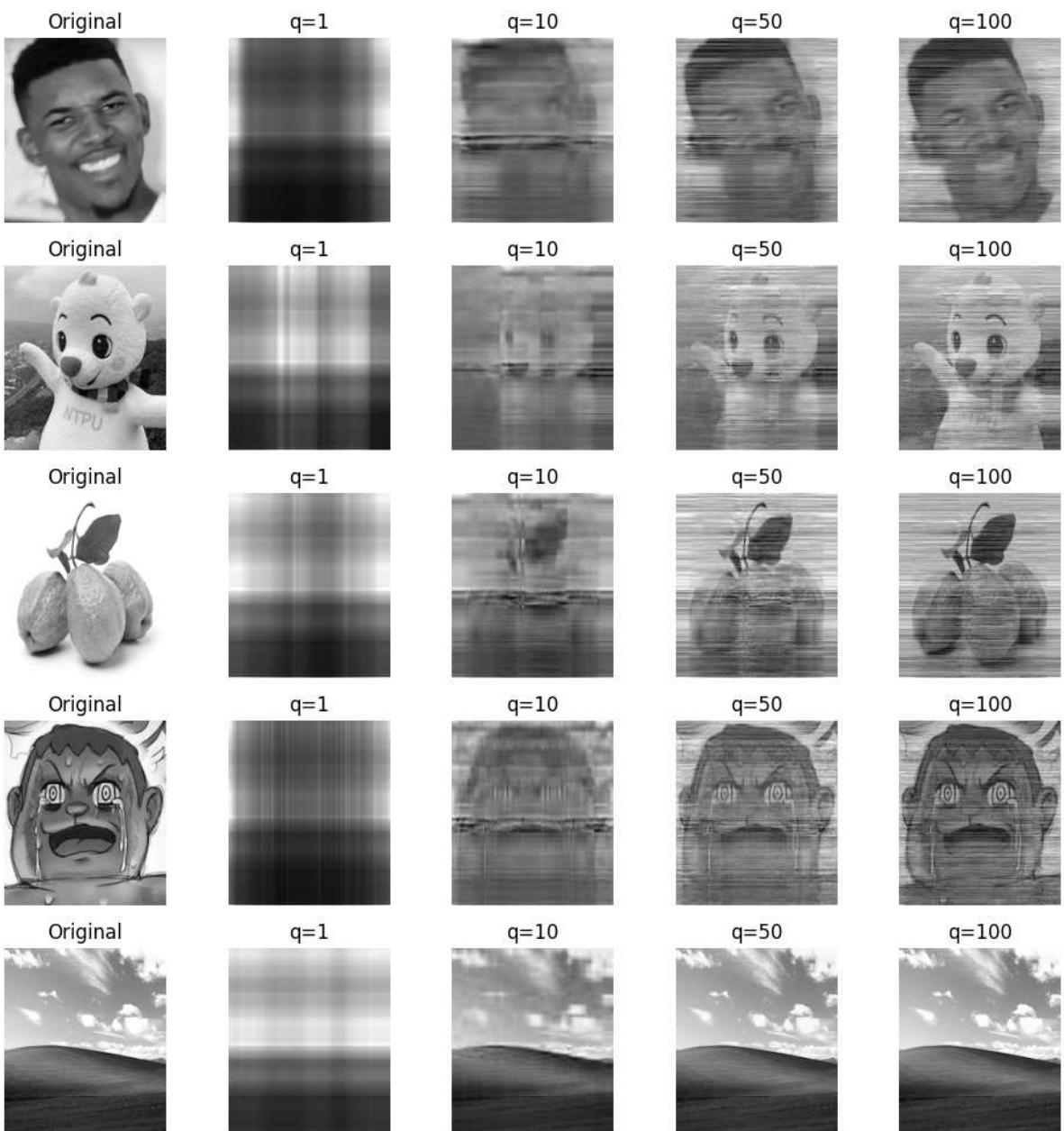
imgfiles = [ '/content/drive/MyDrive/淺度機器學習/Nick Young.jpg',
             '/content/drive/MyDrive/淺度機器學習/big north.jpeg',
             '/content/drive/MyDrive/淺度機器學習/guava.jpg',
             '/content/drive/MyDrive/淺度機器學習/fat tiger.jpeg',
             '/content/drive/MyDrive/淺度機器學習/windows.PNG']

X_list = []
for imgfile in imgfiles:
    X = mpimg.imread(imgfile)
    if len(X.shape) > 2:
        X = np.mean(X, axis=2)
    X_list.append(X)

key = X_list[4]

q_list = [1, 10, 50, 100]
compare_images(key, X_list, q_list)

```



在不同的q值下，加密的圖像仍然保留了許多原始圖像的特徵和細節。但是，與使用人臉圖像作為加密密鑰相比，使用風景圖像加密的影像可能表現略遜。在一些情況下，解密後的影像可能有些失真或像素化，特別是在較低的q值下。

這顯示出密鑰的選擇對於圖像加密和解密的效果至關重要，而密鑰中所包含的圖像特徵可以直接受影響加密結果的質量。

## 結語：

這個專案讓我們深入瞭解了圖像壓縮和還原技術，特別是SVD的 "Rank q approximation" 方法。透過實際的實驗和研究，我們學到了如何運用不同的q值來實現圖像壓縮，同時也明白了在壓縮率和圖像品質之間需要做出權衡的情況。

此外，成功設計了一個程式，用來觀察手寫數字影像，有助於更深入地理解數據的特性。計算了壓縮倍數，以評估SVD的效能，同時展示了壓縮和還原後的圖像，這提供了實際應用的寶貴見解。

最後，通過學習圖像加密和解密的概念，發現基於人臉特徵的加密方法能夠明顯提升解密效果。總之，這個專案豐富了知識，有助於更好地應對圖像處理、數據壓縮以及圖像安全性等方面的挑戰。這些學習對未來的工作具有重要參考價值。

---

# 人臉識別分類器：Yale Face 資料集的性能評比研究

---

## 專案前言：

在當今數位時代，人臉識別技術在各個領域都有著廣泛的應用，從安全系統到社交媒體，都需要高效且準確的人臉識別分類器。為了不斷提升這些分類器的性能，我們進行了一項針對Yale Face資料集的性能評比研究。這個研究的目標是比較不同的分類器，包括多元羅吉斯回歸、支援向量機和神經網路，並評估它們在人臉識別方面的表現。

## 專案目標簡介：

本專案的主要目標是探究並評估多種不同的人臉識別分類器在Yale Face資料集上的性能。我們將利用經過標準化的訓練資料，分別訓練多元羅吉斯回歸、支援向量機和神經網路等分類器，並使用獨立的測試資料集來評估它們的準確性和性能。

通過這項研究，我們的目標如下：

1. 比較不同分類器在人臉識別方面的效能，以確定哪種方法在Yale Face資料集上表現最佳。
2. 評估每個分類器的識別率和誤判率，以量化其性能。
3. 使用交叉驗證策略，全面優化各個分類器，以確保它們能夠適應不同的情境。
4. 分析分類器的優勢和限制，提供改進建議，以便未來的應用中能夠更好地應對人臉識別挑戰。

這些目標不僅有助於我深入瞭解不同人臉識別方法的適用性，還提供了寶貴的學習機會，讓我更好地應對未來的數據科學和機器學習挑戰。

---

## 匯入套件

```
In [ ]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
import random
import warnings
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.svm import SVC, LinearSVC
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
```

# 檔案讀入以及資料前處理

```
In [ ]: # 讀取數據  
data = scipy.io.loadmat('/content/drive/MyDrive/淺度機器學習/allFaces.mat')  
X = data['faces'].T  
  
# 構建標籤  
label_counts = [64, 62, 64, 64, 62, 64, 64, 64, 64, 64, 60, 59, 60, 63, 62, 63, 63,  
    64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,  
    64, 64, 64]  
labels = np.repeat(np.arange(len(label_counts)), label_counts)  
  
# 將標籤添加到X數組中  
X_labeled = np.c_[X, labels]  
  
# 將X_labeled轉換為dataframe  
df = pd.DataFrame(X_labeled, columns=[f"i" for i in range(X.shape[1])] + ["label"])
```

以上程式碼讀取了一個.mat格式的Yale Face 人臉資料集，然後從數據中構建了標籤(label)。接著，它使用numpy的repeat函數重複每個標籤的值，將標籤添加到影像資料數組中，並將整個結果轉換為一個Pandas DataFrame。最終得到的DataFrame包含了影像資料和對應的標籤。

```
In [ ]: df['label'].nunique()
```

```
Out[ ]: 38
```

經過 `df['label'].nunique()` 的計算，可以確定 Yale Face 人臉資料集中包含 38 個不同的人臉圖像。

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

```
Out[ ]: (2410, 32257)
```

根據 `df.shape` 的結果，Yale Face 人臉資料集的 DataFrame 共有 2410列和，32257 行，其中最後一行是用於表示圖像所屬人的唯一 ID。如果扣除這一列標籤欄，DataFrame 將具有 2410列和 32257 行，表示這個資料集中包含了2410 張人臉圖像。每個人都提供了 60 張不同的人臉圖像，總共有 38 個不同的人。每張人臉圖像都被表示為一個一維陣列，長度為 32256。這些一維陣列可以被重塑為 168x192 的矩陣，以便對圖像進行可視化或進一步的處理。

## 人臉圖像展示(修改自Eigenfaces)

```
In [ ]: def plot_random_faces(pixels, n=50):  
    fig, axes = plt.subplots(5, 10, figsize=(14, 6))  
    sample_indices = random.sample(range(len(pixels)), n)  
    for i, ax in enumerate(axes.flat):  
        ax.imshow(np.array(pixels)[sample_indices[i]].reshape(168, 192).T, cmap='gray')  
        ax.axis('off')  
    plt.show()
```

```
In [ ]: X = df.drop('label', axis=1)  
y = df['label']  
plot_random_faces(X)
```



以上程式碼展示了隨機挑選的 50 張人臉影像，透過這些影像，可以觀察到在不同的光影角度下，人臉的外貌會有所不同。例如，在某些影像中，臉部的輪廓清晰可見，而在其他影像中，臉部的細節可能因光線的變化而產生明暗不一的區域。此外，還可以觀察到人臉在不同的角度下，眼睛、嘴巴和鼻子等特徵的位置和大小也會有所變化，這些特徵的變化可能會影響人臉的識別和辨識。

## 資料切割及資料標準化

```
In [ ]: #預處理
X = np.array(df.iloc[:, :-1])
y = np.array(df.iloc[:, -1])

#切割資料
X_train, X_test, y_train, y_test = train_test_split(
X, y, test_size=0.3, random_state=7)

#資料標準化
scaler = StandardScaler()
X_train_ = scaler.fit_transform(X_train)
X_test_ = scaler.fit_transform(X_test)
```

利用主成分分析取得前兩項成分，並繪製其散布圖。

```
In [ ]: # 創建PCA對象，進行標準化
pca_normalized = PCA()
scaler = StandardScaler()

# 對數據進行標準化和擬合
data_standardized = scaler.fit_transform(df)
pca_normalized.fit(data_standardized)
transformed_normalized = pca_normalized.transform(data_standardized)
labels = df['label']
```

```
In [ ]: n_labels = 38
colors = plt.cm.rainbow(np.linspace(0, 1, n_labels))

pc1 = transformed_normalized[:, 0]
pc2 = transformed_normalized[:, 1]

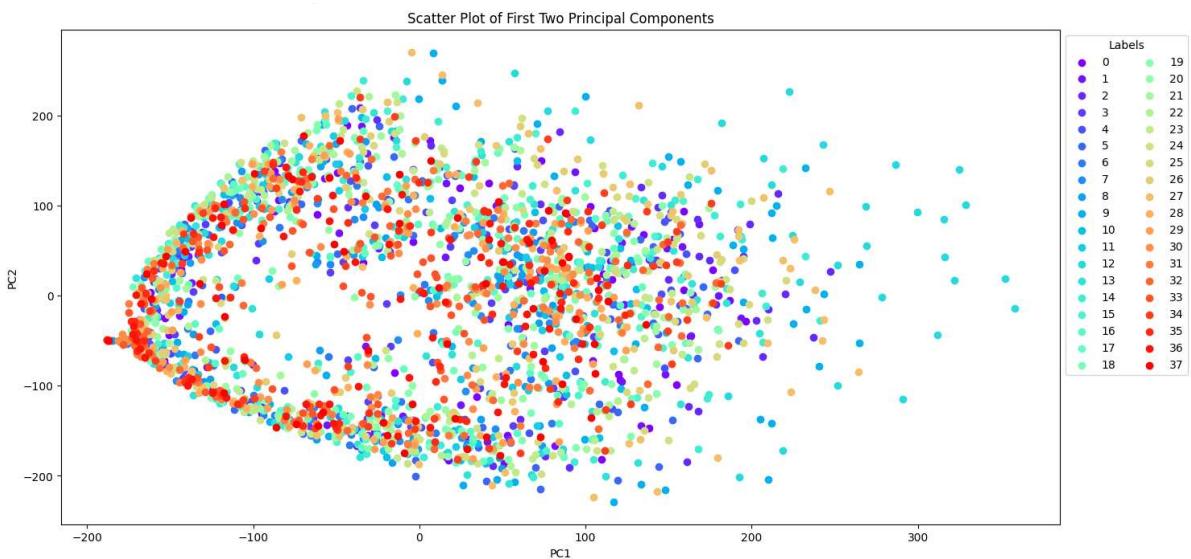
plt.figure(figsize=(16, 8))
for label, color in zip(range(n_labels), colors):
```

```

mask = (labels == label)
plt.scatter(pc1[mask], pc2[mask], c=color, label=label)

plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('Scatter Plot of First Two Principal Components')
plt.legend(title="Labels", ncol=2, bbox_to_anchor=(1, 1.0))
plt.show()

```



通過觀察前兩個主成分的散佈圖，我們可以發現資料點非常分散，並且很難辨認出任何明顯的分類規律。這表示前兩個主成分無法很好地解釋資料中的變異性，因此我們需要考慮探索更多的主成分，以便更全面地捕捉資料的變異性。

根據以上資訊，我們的資料屬於高維度，因此在進行機器學習分類時，建議先挑選適合高維度的超參數和算法進行設定和實現。

**第一段：使用經過標準化的訓練資料來訓練多元羅吉斯回歸(Multinomial Logistic Regression)、支援向量機(Support Vector Machine)、神經網路模型(Neural Network)，並使用測試資料集來評估模型的準確性。**

## 1-1 多元羅吉斯回歸 (Multinomial Logistic Regression)

在此多元羅吉斯回歸模型中，設定了 `solver='lbfgs'` 參數，這代表我們指定使用 lbfgs 演算法。“lbfgs” 算法適用於高維度的問題，且通常比 "liblinear" 算法更快。由於 "lbfgs" 算法是基於梯度下降的優化算法，因此可以更有效地處理具有大量特徵的數據集。

```
In [ ]: opts = dict(tol = 1e-6, max_iter = int(1e3))
#opts = {'tol': 1e-6, 'max_iter': int(1e6), 'verbose': 1}
```

```
solver = 'lbfgs'

mlr_clf_original = LogisticRegression(solver = solver, **opts)
mlr_clf_original.fit(X_train_, y_train)
y_pred = mlr_clf_original.predict(X_test_)

# 測試資料之準確率回報
print(f'Test set classification accuracy of Multinomial Logistic Regression:
{accuracy_score(y_test, y_pred):.2%}\n')
print(f'Test set classification accuracy of Multinomial Logistic Regression:
{mlr_clf_original.score(X_test_, y_test):.2%}\n')
print('-----')
print(classification_report(y_test, y_pred))
```

Test set classification accuracy of Multinomial Logistic Regression: 96.40%

Test set classification accuracy of Multinomial Logistic Regression: 96.40%

	precision	recall	f1-score	support
0	1.00	1.00	1.00	18
1	0.88	1.00	0.94	15
2	1.00	0.81	0.89	21
3	1.00	1.00	1.00	20
4	1.00	1.00	1.00	14
5	1.00	1.00	1.00	24
6	1.00	0.91	0.95	23
7	0.94	0.80	0.86	20
8	0.86	1.00	0.92	18
9	1.00	0.94	0.97	17
10	1.00	1.00	1.00	23
11	1.00	1.00	1.00	24
12	1.00	1.00	1.00	13
13	1.00	1.00	1.00	18
14	0.75	1.00	0.86	12
15	1.00	0.93	0.96	14
16	1.00	1.00	1.00	16
17	1.00	1.00	1.00	19
18	0.96	1.00	0.98	22
19	1.00	1.00	1.00	17
20	1.00	0.95	0.97	20
21	1.00	1.00	1.00	13
22	0.96	1.00	0.98	23
23	1.00	0.94	0.97	17
24	1.00	0.90	0.95	21
25	1.00	0.96	0.98	26
26	1.00	0.86	0.92	21
27	1.00	0.92	0.96	13
28	1.00	1.00	1.00	20
29	1.00	1.00	1.00	13
30	1.00	1.00	1.00	19
31	0.96	1.00	0.98	23
32	1.00	0.95	0.98	21
33	0.70	1.00	0.83	19
34	1.00	0.84	0.91	19
35	0.83	1.00	0.90	19
36	1.00	0.95	0.98	22
37	0.96	1.00	0.98	26
accuracy			0.96	723
macro avg	0.97	0.97	0.96	723
weighted avg	0.97	0.96	0.96	723

根據結果顯示，Multinomial Logistic Regression在這個測試集上達到了96.4%的分類準確率。從分類報告可以看出，對於大多數類別，分類器都能達到很好的精確度、召回率和F1得分。

這個分類器的正確率很高，但是執行一次需要十分鐘，這意味著在需要即時結果的應用中，它可能不太實用。為了改進這個問題，可以考慮使用主成分分析(PCA)來降低特徵維度，以提高分類器的運行效率。

## 1-2 支援向量機 (Support Vector Machine)

在這個 SVM 分類器中，我們選擇了 `kernel='rbf'`，這代表我們使用 RBF 核函數進行分類。RBF 核函數是 SVM 中最常用的核函數之一，可以有效地處理非線性可分數據。同時，我們設定了  $C = 10$  的懲罰參數，這是 SVM 的正則化參數，控制著模型對於錯誤的容忍度。

```
In [ ]: C = 10
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))

svm_clf_original = SVC(kernel = 'rbf', **opts)
svm_clf_original.fit(X_train_, y_train)
y_pred = svm_clf_original.predict(X_test_)
print(f'Test set classification accuracy of Support Vector Machin:
{accuracy_score(y_test, y_pred):.2%}\n')
print(f'Test set classification accuracy of Support Vector Machin:
{svm_clf_original.score(X_test_, y_test):.2%}\n')
print('-----')
print(classification_report(y_test, y_pred))
```

Test set classification accuracy of Support Vector Machin: 93.36%

Test set classification accuracy of Support Vector Machin: 93.36%

	precision	recall	f1-score	support
0	0.90	1.00	0.95	18
1	1.00	0.93	0.97	15
2	1.00	0.81	0.89	21
3	1.00	0.95	0.97	20
4	0.82	1.00	0.90	14
5	0.91	0.88	0.89	24
6	1.00	0.87	0.93	23
7	1.00	0.85	0.92	20
8	1.00	0.94	0.97	18
9	0.74	0.82	0.78	17
10	1.00	1.00	1.00	23
11	1.00	1.00	1.00	24
12	0.93	1.00	0.96	13
13	1.00	1.00	1.00	18
14	0.92	0.92	0.92	12
15	1.00	0.79	0.88	14
16	1.00	1.00	1.00	16
17	0.95	1.00	0.97	19
18	0.95	0.95	0.95	22
19	1.00	0.94	0.97	17
20	1.00	0.95	0.97	20
21	0.93	1.00	0.96	13
22	0.96	1.00	0.98	23
23	0.94	0.88	0.91	17
24	1.00	0.90	0.95	21
25	1.00	0.88	0.94	26
26	1.00	0.86	0.92	21
27	0.92	0.85	0.88	13
28	1.00	1.00	1.00	20
29	1.00	1.00	1.00	13
30	0.73	1.00	0.84	19
31	0.96	1.00	0.98	23
32	1.00	0.90	0.95	21
33	0.69	0.95	0.80	19
34	1.00	0.84	0.91	19
35	0.61	1.00	0.76	19
36	1.00	0.82	0.90	22
37	1.00	1.00	1.00	26
accuracy			0.93	723
macro avg	0.94	0.93	0.93	723
weighted avg	0.95	0.93	0.94	723

根據以上報告，Support Vector Machine模型在測試集上的分類準確率為93.36%。除類別9和類別35，大多數類別的F1分數都在0.9以上，顯示模型對大多數類別的預測都是準確的。總體而言，這個結果顯示了SVM模型在這個分類任務上的相對良好表現。

## 1-3 神經網路 (Neural Network)

在這個神經網絡分類器中，我們選擇了一個隱藏層，其中包含 50 個神經元。我們選擇使用 ReLU 激活函數來增強模型的非線性能力，同時使用 lbfgs 求解器來優化模型參數。

```
In [ ]: hidden_layers = (50, )
activation = 'relu'
```

```
opts = dict(hidden_layer_sizes = hidden_layers, verbose = False,
            activation = activation, tol = 1e-6, max_iter = int(1e6))
# solver = 'sgd' # not efficient, need more tuning # solver = 'lbfgs' # not suitable
solver = 'lbfgs' # default solver
Ann_clf_original = MLPClassifier(solver = solver, random_state = 8
, **opts)
Ann_clf_original.fit(X_train_, y_train)
y_pred= Ann_clf_original.predict(X_test_)

print(f'Test set classification accuracy of Neural Network:
{accuracy_score(y_test, y_pred):.2%}\n')
print(f'Test set classification accuracy of Neural Network:
{Ann_clf_original.score(X_test_, y_test):.2%}\n')
print('-----')
print(classification_report(y_test, y_pred))
```

Test set classification accuracy of Neural Network: 88.11%

Test set classification accuracy of Neural Network: 88.11%

---

	precision	recall	f1-score	support
0	0.75	0.83	0.79	18
1	0.67	0.80	0.73	15
2	0.89	0.81	0.85	21
3	1.00	0.85	0.92	20
4	1.00	1.00	1.00	14
5	0.78	0.75	0.77	24
6	0.94	0.74	0.83	23
7	1.00	0.75	0.86	20
8	1.00	0.94	0.97	18
9	0.62	0.88	0.73	17
10	0.96	1.00	0.98	23
11	0.88	0.92	0.90	24
12	0.81	1.00	0.90	13
13	1.00	0.78	0.88	18
14	0.79	0.92	0.85	12
15	0.92	0.86	0.89	14
16	1.00	0.75	0.86	16
17	0.90	0.95	0.92	19
18	0.91	0.95	0.93	22
19	0.93	0.82	0.87	17
20	1.00	0.90	0.95	20
21	0.86	0.92	0.89	13
22	0.96	1.00	0.98	23
23	0.65	0.76	0.70	17
24	0.85	0.81	0.83	21
25	0.93	0.96	0.94	26
26	1.00	0.86	0.92	21
27	0.72	1.00	0.84	13
28	0.95	0.95	0.95	20
29	0.87	1.00	0.93	13
30	0.79	1.00	0.88	19
31	1.00	0.91	0.95	23
32	0.95	0.95	0.95	21
33	0.76	1.00	0.86	19
34	0.78	0.74	0.76	19
35	0.95	1.00	0.97	19
36	0.89	0.73	0.80	22
37	1.00	0.81	0.89	26
accuracy			0.88	723
macro avg	0.89	0.88	0.88	723
weighted avg	0.89	0.88	0.88	723

---

使用神經網絡進行測試，分類準確率達到了88.11%。然而，一些類別的精確度和召回率相對較低，這可能需要進一步調整模型以提高整體表現。儘管已經達到了不錯的結果，但仍有很大的優化空間，因為神經網路模型有很多參數可以進行優化。因此，可以通過調整參數或使用其他更高級的技術來優化模型以提高分類準確率。

## 第二段：利用經過標準化的原始資料的主成分來訓練模型，並使用測試資料來評估模型的準確度

---

# 將 Yale Face 人臉資料集資料進行 PCA 降維，並保留前50個主成分。

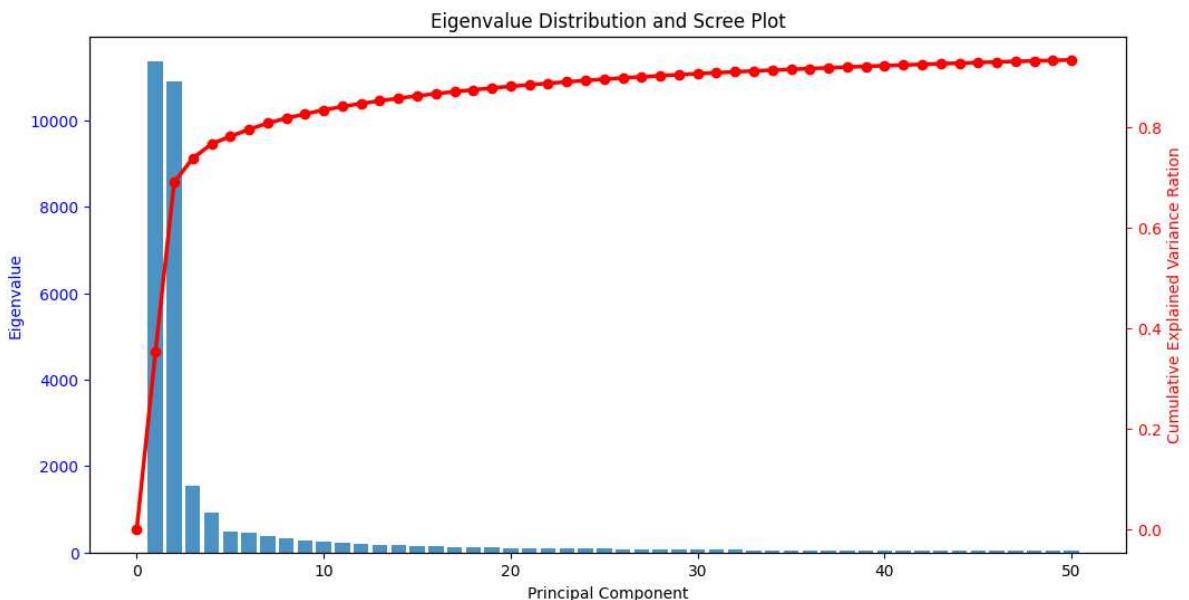
```
In [ ]: from sklearn.decomposition import PCA
pca_normalized = PCA(n_components = 50).fit(X_train_)

# 計算特徵值和特徵向量
eigenvalues = pca_normalized.explained_variance_
eigenvectors = pca_normalized.components_

# 繪製特徵值分佈圖和scree plot
fig, ax1 = plt.subplots(figsize=(12, 6))

# 繪製特徵值條形圖
ax1.bar(range(1, len(eigenvalues)+1), eigenvalues, alpha=0.8)
ax1.set_xlabel('Principal Component')
ax1.set_ylabel('Eigenvalue', color='b')
ax1.tick_params('y', colors='b')
ax1.set_title('Eigenvalue Distribution and Scree Plot')

# 繪製累積解釋方差曲線
ax2 = ax1.twinx()
ax2.plot(np.insert(np.cumsum(pca_normalized.explained_variance_ratio_), 0, 0), 'ro-', linewidth=2.5)
ax2.set_ylabel('Cumulative Explained Variance Ratio', color='r')
ax2.tick_params('y', colors='r')
plt.show()
print('主成份分析的累積解釋變異 :\n')
print(np.cumsum(pca_normalized.explained_variance_ratio_))
```



主成份分析的累積解釋變異：

```
[0.35254127 0.69079556 0.73856932 0.76697139 0.78202694 0.79603751
 0.80816716 0.81833265 0.82663486 0.8346434 0.84145958 0.84741833
 0.85283535 0.85798377 0.86271954 0.86712636 0.87112087 0.87478369
 0.87837713 0.88163796 0.88474655 0.88770781 0.89057342 0.89336401
 0.89594828 0.89837475 0.90062216 0.90280353 0.90490024 0.90691285
 0.90878072 0.91058365 0.91233318 0.91398371 0.91555888 0.91709144
 0.91859833 0.92009587 0.92149494 0.92287679 0.92420583 0.92550256
 0.92675147 0.92795801 0.92914919 0.93032446 0.93137862 0.93241445
 0.9334238 0.93440467]
```

```
In [ ]: pca_normalized = PCA(n_components = 50).fit(X_train_)
Z_train = pca_normalized.transform(X_train_)
Z_test = pca_normalized.transform(X_test_)
```

根據主成份分析的累積解釋變異累積報表，我們可以觀察到，保留前50個主成份時，能夠解釋資料變異的比例大約為93.44%。即使無法解釋所有的變異，使用主成份分析仍然是一種簡單有效的資料降維方法，可以在保留少量資訊的情況下進行資料分析、分類和預測。使用PCA降維後，原本的資料維度從32256維降至50維，即僅保留50個主成分。這樣做的好處是大幅減少了資料的維度，從而降低計算量和記憶體需求。這也意味著，原本的資料維度被壓縮了約645倍，這是相當顯著的成果。

以下將進行三種分類器的實作。

## 2-1 主成分分析在多元羅吉斯回歸中的應用 (Multinomial Logistic Regression with PCA)

此多元羅吉斯回歸模型所設定的參數與第一段相同。

```
In [ ]: opts = dict(tol = 1e-6, max_iter = int(1e6))
solver = 'lbfgs'
mlr_clf_PCA = LogisticRegression(solver = solver, **opts)
mlr_clf_PCA.fit(Z_train, y_train)
y_pred = mlr_clf_PCA.predict(Z_test)
print(f'Test set classification accuracy of Multinomial Logistic Regression with PCA
{mlr_clf_PCA.score(Z_test, y_test):.2%}\n')
print('-----')
print(classification_report(y_test, y_pred))
```

Test set classification accuracy of Multinomial Logistic Regression with PCA: 92.67%

	precision	recall	f1-score	support
0	0.94	0.94	0.94	18
1	0.78	0.93	0.85	15
2	1.00	0.76	0.86	21
3	0.95	0.95	0.95	20
4	1.00	1.00	1.00	14
5	0.95	0.75	0.84	24
6	0.95	0.87	0.91	23
7	1.00	0.80	0.89	20
8	0.90	1.00	0.95	18
9	1.00	0.94	0.97	17
10	1.00	1.00	1.00	23
11	1.00	1.00	1.00	24
12	0.93	1.00	0.96	13
13	1.00	0.94	0.97	18
14	0.75	1.00	0.86	12
15	1.00	1.00	1.00	14
16	0.94	1.00	0.97	16
17	0.95	1.00	0.97	19
18	0.84	0.95	0.89	22
19	0.94	1.00	0.97	17
20	0.86	0.95	0.90	20
21	0.87	1.00	0.93	13
22	1.00	0.96	0.98	23
23	0.81	0.76	0.79	17
24	0.90	0.86	0.88	21
25	0.96	1.00	0.98	26
26	1.00	0.81	0.89	21
27	0.79	0.85	0.81	13
28	0.95	0.95	0.95	20
29	0.92	0.92	0.92	13
30	0.95	0.95	0.95	19
31	0.92	1.00	0.96	23
32	0.95	0.95	0.95	21
33	0.83	1.00	0.90	19
34	0.88	0.74	0.80	19
35	0.81	0.89	0.85	19
36	0.95	0.86	0.90	22
37	1.00	1.00	1.00	26
accuracy			0.93	723
macro avg	0.93	0.93	0.92	723
weighted avg	0.93	0.93	0.93	723

根據以上報表顯示，使用前50個主成份的多元羅吉斯回歸模型在120個測試樣本上的分類準確率達92.67%。

相較於未降維的**(1-1多元羅吉斯回歸)**模型，使用PCA進行降維後的模型準確率確實有所下降，但是這個下降的幅度不是很嚴重。原來的模型準確率為96.40%，而使用PCA進行降維後的模型準確率降至92.67%。這意味著，使用PCA進行降維後的模型仍然可以在相當高的準確率下進行分類。

## 2-2 主成分分析在支援向量機的應用 (Support Vector Machine with PCA)

此支援向量機模型所設定的參數與第一段相同。

```
In [ ]: C = 10
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))

svm_clf_PCA = SVC(kernel = 'rbf', **opts)
svm_clf_PCA.fit(Z_train, y_train)
y_pred = svm_clf_PCA.predict(Z_test)
print(f'Test set classification accuracy of Support Vector Machin with PCA:
{svm_clf_PCA.score(Z_test, y_test):.2%}\n')
print('-----')
print(classification_report(y_test, y_pred))
```

Test set classification accuracy of Support Vector Machin with PCA: 88.80%

	precision	recall	f1-score	support
0	0.76	0.89	0.82	18
1	0.88	0.93	0.90	15
2	1.00	0.76	0.86	21
3	1.00	0.95	0.97	20
4	0.64	1.00	0.78	14
5	0.91	0.83	0.87	24
6	1.00	0.87	0.93	23
7	0.93	0.70	0.80	20
8	1.00	0.94	0.97	18
9	0.78	0.82	0.80	17
10	1.00	1.00	1.00	23
11	1.00	1.00	1.00	24
12	1.00	1.00	1.00	13
13	1.00	0.89	0.94	18
14	0.71	0.83	0.77	12
15	1.00	0.79	0.88	14
16	1.00	0.94	0.97	16
17	0.83	1.00	0.90	19
18	0.87	0.91	0.89	22
19	1.00	0.76	0.87	17
20	0.94	0.85	0.89	20
21	0.87	1.00	0.93	13
22	0.96	1.00	0.98	23
23	0.87	0.76	0.81	17
24	1.00	0.90	0.95	21
25	1.00	0.81	0.89	26
26	0.90	0.86	0.88	21
27	0.91	0.77	0.83	13
28	0.95	0.95	0.95	20
29	0.87	1.00	0.93	13
30	0.70	1.00	0.83	19
31	0.85	0.96	0.90	23
32	1.00	0.76	0.86	21
33	0.64	0.95	0.77	19
34	0.82	0.74	0.78	19
35	0.64	0.95	0.77	19
36	0.94	0.68	0.79	22
37	1.00	1.00	1.00	26
accuracy			0.89	723
macro avg	0.90	0.89	0.89	723
weighted avg	0.91	0.89	0.89	723

根據以上報表顯示，使用前50個主成份的支援向量機模型在723個測試樣本上的分類準確率達89.07%。

---

相較於未降維的**(1-2支援向量機)**，經過PCA進行降維後，模型在測試集的準確率由93.36%降至89.07%。這表明，使用PCA進行降維可能會導致某些關鍵特徵的丟失，進而影響模型的性能。但仍然是一個不錯的結果。

## 2-3 主成分分析在神經網路的應用 (Neural Network with PCA)

此神經網路模型所設定的參數與第一段相同。

```
In [ ]: hidden_layers = (50, )
activation = 'relu'
opts = dict(hidden_layer_sizes = hidden_layers, verbose = False,
            activation = activation, tol = 1e-6, max_iter = int(1e6))
solver = 'lbfgs'
Ann_clf_PCA = MLPClassifier(solver = solver, random_state = 8
                           , **opts)
Ann_clf_PCA.fit(Z_train, y_train)
y_pred= Ann_clf_PCA.predict(Z_test)

print(f'Test set classification accuracy of Neural Network:
{Ann_clf_PCA.score(Z_test, y_test):.2%}\n')
print('-----')
print(classification_report(y_test, y_pred))
```

Test set classification accuracy of Neural Network: 85.89%

	precision	recall	f1-score	support
0	0.75	0.67	0.71	18
1	0.76	0.87	0.81	15
2	0.67	0.67	0.67	21
3	1.00	0.90	0.95	20
4	0.76	0.93	0.84	14
5	0.91	0.83	0.87	24
6	0.83	0.83	0.83	23
7	0.79	0.75	0.77	20
8	1.00	0.94	0.97	18
9	0.83	0.88	0.86	17
10	0.87	0.87	0.87	23
11	0.90	0.79	0.84	24
12	0.92	0.92	0.92	13
13	0.94	0.83	0.88	18
14	0.71	0.83	0.77	12
15	0.81	0.93	0.87	14
16	0.93	0.81	0.87	16
17	0.86	1.00	0.93	19
18	0.95	0.91	0.93	22
19	0.93	0.76	0.84	17
20	0.70	0.95	0.81	20
21	0.86	0.92	0.89	13
22	0.92	0.96	0.94	23
23	0.78	0.82	0.80	17
24	0.90	0.86	0.88	21
25	0.96	0.85	0.90	26
26	0.89	0.76	0.82	21
27	0.81	1.00	0.90	13
28	1.00	1.00	1.00	20
29	0.92	0.92	0.92	13
30	0.84	0.84	0.84	19
31	0.81	0.96	0.88	23
32	0.78	0.86	0.82	21
33	0.90	1.00	0.95	19
34	0.67	0.63	0.65	19
35	0.89	0.89	0.89	19
36	0.94	0.68	0.79	22
37	0.96	0.92	0.94	26
accuracy			0.86	723
macro avg	0.86	0.86	0.86	723
weighted avg	0.86	0.86	0.86	723

根據以上報表顯示，使用前50個主成份的神經網路模型在723個測試樣本上的分類準確率達84.51%。

相較於未降維的**(1-3神經網路)**，經過PCA進行降維後，使用神經網路進行分類的模型在測試集上的準確率從88.11%下降至84.51%。但仍然是一個具有應用價值的模型。需要進一步分析和調整模型的參數，以提高其準確率。

### 第三段：使用交叉驗證進行分類器的全面優化策略

在機器學習的模型訓練中，為了避免過擬合和選擇最佳的超參數，交叉驗證已經成為一種常用的技術。在本段落中，我們將使用5-fold交叉驗證來對每種分類器進行全面優化，以確定最佳的超參數設置。我們將使用Scikit-learn套件中的GridSearchCV函數對模型的超參數進行調整，以獲得最優的模型。

```
In [ ]: warnings.filterwarnings('ignore')
```

```
In [ ]: # 1. 定義 Pipeline
pipe_lr = Pipeline([('lr', LogisticRegression())])
pipe_svm = Pipeline([('svm', SVC())])
pipe_ann = Pipeline([('ann', MLPClassifier())])
pipe_lr_pca = Pipeline([('pca', PCA()), ('lr', LogisticRegression())])
pipe_svm_pca = Pipeline([('pca', PCA()), ('svm', SVC())])
pipe_ann_pca = Pipeline([('pca', PCA()), ('ann', MLPClassifier())])

# 2. 定義參數範圍
params_lr = {'lr__solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']}

params_svm = {'svm__C': [0.1, 1, 10, 100],
              'svm__kernel': ['linear', 'poly', 'rbf', 'sigmoid']}

params_ann = {'ann__hidden_layer_sizes':
              [(30,), (50,), (100,), (30, 30), (50, 30)],
              'ann__activation': ['identity', 'logistic', 'tanh', 'relu'],
              'ann__solver': ['lbfgs', 'sgd', 'adam']}

params_lr_pca = {'pca__n_components': [50], 'lr__solver':
                 ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']}

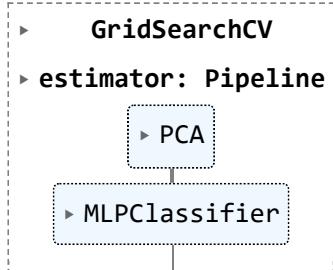
params_svm_pca = {'pca__n_components': [50], 'svm__C':
                  [0.1, 1, 10, 100], 'svm__kernel': ['linear', 'poly', 'rbf', 'sigmoid']}

params_ann_pca = {'pca__n_components': [50], 'ann__hidden_layer_sizes':
                  [(30,), (50,), (100,), (30, 30), (50, 30)], 'ann__activation':
                  ['identity', 'logistic', 'tanh', 'relu'], 'ann__solver': ['lbfgs', 'sgd', 'adam']}

# 3. 定義 GridSearchCV
grid_lr_pca = GridSearchCV(pipe_lr_pca, params_lr_pca, cv=5)
grid_svm_pca = GridSearchCV(pipe_svm_pca, params_svm_pca, cv=5)
grid_ann_pca = GridSearchCV(pipe_ann_pca, params_ann_pca, cv=5)

# 4. 進行交叉驗證調整參數並找到最好的模型
grid_lr_pca.fit(X_train_, y_train)
grid_svm_pca.fit(X_train_, y_train)
grid_ann_pca.fit(X_train_, y_train)
```

```
Out[ ]:
```



因為在原始數據上訓練的模型需要處理大量的特徵，所以在沒有PCA降維處理的情況下，模型需要處理大量的計算，因此模型的訓練時間會非常長。因此，在本次實驗中，我們只對PCA後的資料進行了交叉驗證。

```
In [ ]: # 5. 計算最好的參數和交叉驗證得分
print('PCA + Logistic Regression:', grid_lr_pca.best_params_, grid_lr_pca.best_score_)
print('PCA + SVM:', grid_svm_pca.best_params_, grid_svm_pca.best_score_)
print('PCA + Neural Network:', grid_ann_pca.best_params_, grid_ann_pca.best_score_)

PCA + Logistic Regression: {'lr_solver': 'newton-cg', 'pca_n_components': 50} 0.90
98958790581708
PCA + SVM: {'pca_n_components': 50, 'svm_C': 0.1, 'svm_kernel': 'linear'} 0.89033
06234965672
PCA + Neural Network: {'ann_activation': 'logistic', 'ann_hidden_layer_sizes': (10
0,), 'ann_solver': 'adam', 'pca_n_components': 50} 0.9330149421452777
```

在此使用了三種不同的模型，並且在每種模型中使用了PCA降維，得到了對應的訓練正確率。其中，PCA + Neural Network模型的訓練正確率最高，達到了0.933，也就是在訓練集上，模型對於分類結果的準確率較高。而其他兩種模型，其訓練正確率分別為0.910和0.890，相對較低。

```
In [ ]: y_pred_lr_pca = grid_lr_pca.predict(X_test_)
accuracy_lr_pca = accuracy_score(y_test, y_pred_lr_pca)
print(f'Test set classification accuracy of Logistic Regression with PCA:
{accuracy_lr_pca:.2%}\n')

y_pred_svm_pca = grid_svm_pca.predict(X_test_)
accuracy_svm_pca = accuracy_score(y_test, y_pred_svm_pca)
print(f'Test set classification accuracy of SVM with PCA:
{accuracy_svm_pca:.2%}\n')

y_pred_ann_pca = grid_ann_pca.predict(X_test_)
accuracy_ann_pca = accuracy_score(y_test, y_pred_ann_pca)
print(f'Test set classification accuracy of Neural Network with PCA:
{accuracy_ann_pca:.2%}\n')
```

Test set classification accuracy of Logistic Regression with PCA: 92.81%

Test set classification accuracy of SVM with PCA: 91.42%

Test set classification accuracy of Neural Network with PCA: 93.64%

從三種方法的測試正確率來看，神經網路的表現最佳，其次是羅吉斯回歸，SVM的表現最差。這可能是因為神經網路具有更高的彈性，可以學習更複雜的非線性決策邊界，因此在應對複雜的數據集時表現更好。

此外，PCA降維處理在本例中是相當重要的，因為可以大幅提高模型訓練速度並兼具模型的性能。

---

## 結語：

在本次人臉識別分類器的性能評比研究中，我們得出了一些重要的結論。根據測試正確率的結果，我們可以明確看出，神經網路在這項任務中表現最佳，其次是羅吉斯回歸，而SVM則表現最差。這一結果可能是由於神經網路具有更高的彈性，能夠學習更複雜的非線性決策邊界，因此在處理複雜的數據集時更具優勢。

此外，我們也強調了PCA（主成分分析）降維處理在本例中的重要性。這個過程不僅大幅提高了模型的訓練速度，還兼具改善模型性能的功能。這個結論強調了在處理大型數據集時，降維技術可以有效地提高計算效率，同時確保模型的性能不會受到過多的維度影響。

總之，這項研究為我們提供了對不同人臉識別方法的性能比較，並突顯了神經網路和PCA降維的重要性。這些結論將有助於未來的人臉識別系統設計和應用中，選擇適合的方法以提高準確性和效率。

---

# **SRCNN模型：突破圖像超分辨率的卷積神經網絡**

---

## **專案前言：**

圖像超分辨率是一項關鍵的計算機視覺任務，旨在提高圖像的解析度，使其更清晰且更有用。卷積神經網絡（CNN）已經被廣泛應用於圖像超分辨率任務，其中SRCNN模型是其中一個卓越的代表。本專案旨在深入探討SRCNN模型的訓練和應用，以實現圖像超分辨率的突破。

## **專案目標簡介：**

本次作業的主要目標如下：

### **(一) 訓練SRCNN模型：**

1. 讀入預先訓練好的SRCNN模型權重檔（pth檔）。
2. 對SRCNN模型進行三次額外的訓練，以提升其性能。
3. 在每次訓練後，計算並展示訓練集的PSNR（Peak Signal-to-Noise Ratio）和驗證集的PSNR值，以評估模型的訓練進展。

### **(二) 測試模型性能：**

1. 使用已訓練完成的SRCNN模型對Set5和Set14測試集進行影像生成。
2. 計算並展示在Set5和Set14上的測試PSNR值，以評估模型的性能。

### **(三) 影像生成：**

1. 實現對已訓練完成的SRCNN模型輸入任何一張低解析度影像，並生成相對應的高解析度影像。
2. 展示生成的高解析度影像，以呈現SRCNN模型的實際應用效果。

透過這些目標，我們將深入理解SRCNN模型在圖像超解析度方面的性能，並實際應用它來提高圖像品質。這將有助於我們更好地理解和掌握卷積神經網路在影像處理中的應用。

---

以下是對原始程式碼的改進：

首先，對於訓練對象的改進，將原始的高清圖像作為輸出改為使用殘差圖像進行訓練。在圖像超分辨率任務中，殘差圖像是通過從原始低分辨率圖像中減去其對應的高分辨率圖像來獲得的。這樣做的目的是使模型學習捕捉圖像之間的細微差異，從而更好地恢復細節。

其次，對CNN模型進行優化。我嘗試以下兩個改進：

1. 使用更深的CNN：增加卷積層的數量可以增加模型的深度，提高其表示能力。更深的CNN可以學習到更複雜的特徵和模式，有助於提高圖像超分辨率的效果。

2. 使用較小的捲積核：較小的捲積核可以增加模型的 receptive field，使其能夠更好地捕捉圖像中的細微紋理和邊緣信息。

---

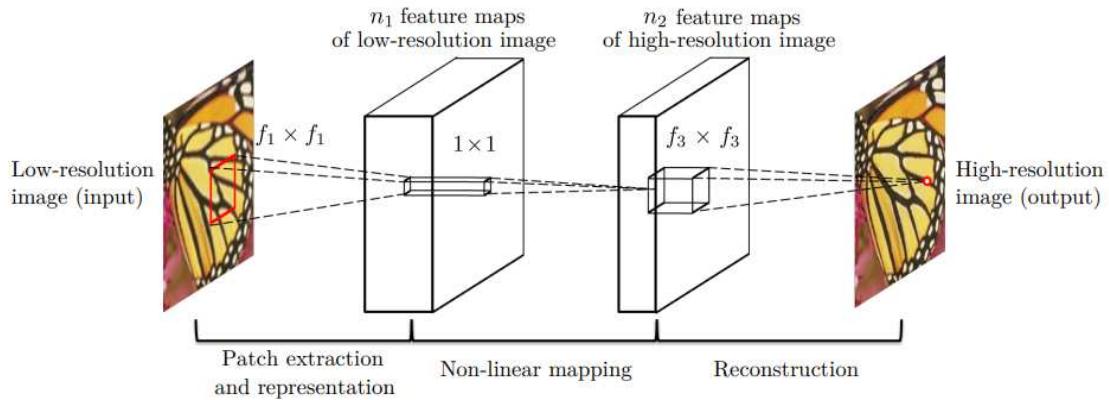
## SRCNN 入門介紹

SRCNN ( Super-Resolution Convolutional Neural Network ) 是一種深度學習模型，專注於解決圖像超分辨率重建的問題。

它能夠將低分辨率圖像恢復為具有更高解析度和更多細節的高分辨率圖像。

SRCNN的核心思想是利用深度卷積神經網絡來學習低分辨率圖像和對應高分辨率圖像之間的映射關係。

這種學習方式使得SRCNN能夠自動從大量的訓練數據中學習到圖像的細節和結構信息，並在重建過程中有效地恢復丟失的細節。



引用自《Learning a Deep Convolutional Network for Image Super-Resolution》  
( ECCV2014 )

根據《Learning a Deep Convolutional Network for Image Super-Resolution》( ECCV2014 )論文的描述，SRCNN模型採用了三個卷積層，這三個卷積層可以被解釋為三個步驟，分別是圖像塊的提取和特徵表示、特徵的非線性映射以及圖像的重建。

1. 圖像塊的提取和特徵表示：在第一步中，低分辨率圖像首先通過雙三次插值被放大到目標尺寸。然後，通過第一個卷積層對放大後的圖像進行處理，提取圖像塊並對每個圖像塊進行特徵表示。這個卷積層的作用是對圖像進行初步的特徵提取和表示。
2. 特徵的非線性映射：在第二步中，通過第二個卷積層對特徵圖進行非線性映射。這個卷積層引入了非線性激活函數，如ReLU (修正線性單元)，以增強模型的非線性建模能力。非線性映射的目的是對圖像特徵進行更深入的提取和抽象，以捕捉更豐富的圖像信息。
3. 圖像的重建：在第三步中，通過第三個卷積層對經過非線性映射的特徵圖進行處理，最終輸出高分辨率圖像的結果。這個卷積層的作用是將經過特徵提取和非線性映射的特徵圖轉化為最終的高分辨率圖像，完成圖像的重建過程。

通過這三個步驟，SRCNN模型能夠從低分辨率圖像中提取有效的特徵並進行非線性映射，從而實現高分辨率圖像的重建。這個模型的設計和訓練旨在提高圖像超分辨率的質量和準確性。

# 解壓縮程式碼

來源：[SRCNN Implementation in PyTorch for Image Super Resolution](#)

```
In [ ]: !unzip /content/drive/MyDrive/srcnn/SRCNN_Implementation_in_PyTorch_for_Image_Super_R
```

## 設定資料夾地址

```
In [ ]: %cd /content/20220606_SRCNN_Implementation_in_PyTorch_for_Image_Super_Resolution/src  
/content/20220606_SRCNN_Implementation_in_PyTorch_for_Image_Super_Resolution/src
```

```
In [ ]: !pip install patchify
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/  
public/simple/  
Collecting patchify  
  Downloading patchify-0.2.3-py3-none-any.whl (6.6 kB)  
Requirement already satisfied: numpy<2,>=1 in /usr/local/lib/python3.10/dist-packages (from patchify) (1.22.4)  
Installing collected packages: patchify  
Successfully installed patchify-0.2.3
```

## patchify\_image 生成殘差圖像--部分代碼

```
# Convert to bicubic and save.  
h, w, _ = patch.shape  
low_res_img = cv2.resize(  
    patch, (int(w*0.5), int(h*0.5)), interpolation=cv2.INTER_CUBIC  
)  
  
# Now upscale using BICUBIC.  
high_res_upscale = cv2.resize(  
    low_res_img, (w, h), interpolation=cv2.INTER_CUBIC  
)  
  
# Create residual image patch.  
residual = cv2.subtract(patch, high_res_upscale)  
cv2.imwrite(f"{out_re_path}/{image_name}_{counter}.png", residual)
```

這段程式碼的目的是為每個補丁圖像生成殘差圖像。

程式碼內容：

1. 將補丁圖像進行插值，得到低解析度圖像 `low_res_img`。
2. 將低解析度圖像進行插值放大，得到與原始補丁圖像相同尺寸的高解析度圖像 `high_res_upscale`。
3. 將原始補丁圖像 `patch` 減去高解析度圖像 `high_res_upscale`，得到殘差圖像 `residual`。
4. 殘差圖像 `residual` 代表了原始補丁圖像中的高頻細節信息。它反映了補丁圖像在插值放大過程中丟失的細節部分。

這個過程的目的是將原始圖像分解為低解析度圖像和高頻細節（殘差），以便在後續的處理中進行超解析度處理或其他相關任務。

透過這種方式，可以保留高頻細節信息並將其與低解析度圖像結合，從而提高圖像的視覺品質或進行其他處理。

---

## 圖像切分為小塊

```
In [ ]: !python patchify_image.py
```

```
Creating patches for 91 images
100% 91/91 [00:12<00:00, 7.54it/s]
```

```
In [ ]: !python bicubic.py --path ../input/Set14/original ../input/Set5/original --scale-factor=19
```

```
Scaling factor: 2x
Low resolution images save path: ../input/test_bicubic_rgb_2x
Original image dimensions: 352, 288
Original image dimensions: 512, 512
Original image dimensions: 529, 656
Original image dimensions: 500, 362
Original image dimensions: 512, 512
Original image dimensions: 720, 576
Original image dimensions: 768, 512
Original image dimensions: 500, 480
Original image dimensions: 276, 276
Original image dimensions: 512, 512
Original image dimensions: 250, 361
Original image dimensions: 586, 391
Original image dimensions: 512, 512
Original image dimensions: 352, 288
Original image dimensions: 280, 280
Original image dimensions: 288, 288
Original image dimensions: 228, 344
Original image dimensions: 256, 256
Original image dimensions: 512, 512
```

---

## 卷積神經網絡架構 ( CNN )

```
class SRCNN(nn.Module):
    def __init__(self):
        super(SRCNN, self).__init__()

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1,
                           padding=1)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(64, 64, kernel_size=3, stride=1,
                           padding=1)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1,
                           padding=1)
        self.conv4 = nn.Conv2d(64, 64, kernel_size=3, stride=1,
                           padding=1)
        self.conv5 = nn.Conv2d(64, 64, kernel_size=3, stride=1,
                           padding=1)
```

```
        self.conv6 = nn.Conv2d(64, 64, kernel_size=3, stride=1,
padding=1)
        self.conv7 = nn.Conv2d(64, 64, kernel_size=3, stride=1,
padding=1)
        self.conv8 = nn.Conv2d(64, 64, kernel_size=3, stride=1,
padding=1)
        self.conv9 = nn.Conv2d(64, 64, kernel_size=3, stride=1,
padding=1)
        self.conv10 = nn.Conv2d(64, 3, kernel_size=3, stride=1,
padding=1)

    def forward(self, x):
        out = self.relu(self.conv1(x))
        out = self.relu(self.conv2(out))
        out = self.relu(self.conv3(out))
        out = self.relu(self.conv4(out))
        out = self.relu(self.conv5(out))
        out = self.relu(self.conv6(out))
        out = self.relu(self.conv7(out))
        out = self.relu(self.conv8(out))
        out = self.relu(self.conv9(out))
        out = self.conv10(out)
        return out
```

以上是我設計的SRCNN超解析度圖像重建模型，它具有以下特點：

1. 深度卷積層架構：模型具有十個卷積層，這使得它能夠進行多層次的特徵提取和重建。這種深度結構有助於模型學習更複雜的圖像特徵和紋理。
2. 小型卷積核：每個卷積層使用3x3的小型卷積核，這有助於捕捉局部細節和特徵。相對於使用較大卷積核的模型，小型卷積核可以提供更精細的細節捕捉能力。
3. 非線性轉換：在每個卷積層之間使用ReLU激活函數，這導致模型具有非線性的特徵表示能力。這有助於模型從低解析度圖像中學習到更複雜的高解析度特徵。

---

## 訓練模型--對資料集進行 100 次完整的迭代。

```
In [ ]: !python train.py --epochs 100
```

```
SRCCN(  
    (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (relu): ReLU(inplace=True)  
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (conv4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (conv5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (conv6): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (conv7): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (conv8): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (conv9): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (conv10): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
)  
Training samples: 22227  
Validation samples: 19  
Epoch 1 of 100  
100% 174/174 [00:23<00:00, 7.33it/s]  
100% 19/19 [00:00<00:00, 20.67it/s]  
Train PSNR: 32.920  
Val PSNR: 33.273  
Saving model...  
Epoch 2 of 100  
100% 174/174 [00:22<00:00, 7.60it/s]  
100% 19/19 [00:00<00:00, 21.93it/s]  
Train PSNR: 33.469  
Val PSNR: 33.604  
Saving model...  
Epoch 3 of 100  
100% 174/174 [00:22<00:00, 7.79it/s]  
100% 19/19 [00:01<00:00, 18.31it/s]  
Train PSNR: 33.832  
Val PSNR: 33.909  
Saving model...  
Epoch 4 of 100  
100% 174/174 [00:24<00:00, 7.19it/s]  
100% 19/19 [00:00<00:00, 22.02it/s]  
Train PSNR: 34.225  
Val PSNR: 33.835  
Saving model...  
Epoch 5 of 100  
100% 174/174 [00:22<00:00, 7.77it/s]  
100% 19/19 [00:01<00:00, 18.46it/s]  
Train PSNR: 34.433  
Val PSNR: 33.768  
Saving model...  
Epoch 6 of 100  
100% 174/174 [00:22<00:00, 7.61it/s]  
100% 19/19 [00:00<00:00, 21.32it/s]  
Train PSNR: 34.546  
Val PSNR: 33.950  
Saving model...  
Epoch 7 of 100  
100% 174/174 [00:22<00:00, 7.60it/s]  
100% 19/19 [00:01<00:00, 18.45it/s]  
Train PSNR: 34.647  
Val PSNR: 34.049  
Saving model...  
Epoch 8 of 100  
100% 174/174 [00:22<00:00, 7.83it/s]  
100% 19/19 [00:00<00:00, 21.32it/s]  
Train PSNR: 34.743  
Val PSNR: 33.992  
Saving model...  
Epoch 9 of 100  
100% 174/174 [00:23<00:00, 7.49it/s]
```

```
Epoch 96 of 100
100% 174/174 [00:22<00:00, 7.60it/s]
100% 19/19 [00:00<00:00, 21.32it/s]
Train PSNR: 37.460
Val PSNR: 33.669
Saving model...
Epoch 97 of 100
100% 174/174 [00:22<00:00, 7.67it/s]
100% 19/19 [00:00<00:00, 21.43it/s]
Train PSNR: 37.469
Val PSNR: 33.672
Saving model...
Epoch 98 of 100
100% 174/174 [00:22<00:00, 7.73it/s]
100% 19/19 [00:00<00:00, 21.62it/s]
Train PSNR: 37.478
Val PSNR: 33.636
Saving model...
Epoch 99 of 100
100% 174/174 [00:22<00:00, 7.71it/s]
100% 19/19 [00:00<00:00, 21.29it/s]
Train PSNR: 37.454
Val PSNR: 33.595
Saving model...
Epoch 100 of 100
100% 174/174 [00:21<00:00, 7.96it/s]
100% 19/19 [00:00<00:00, 19.94it/s]
Train PSNR: 37.458
Val PSNR: 33.612
Saving model...
Finished training in: 40.320 minutes
```

根據每個epoch的訓練集和驗證集的PSNR值，我們可以觀察到以下情況：

1. 訓練集PSNR值逐漸增加：從第一個epoch到最後一個epoch，訓練集的PSNR值逐漸增加，從32.920提高到35.425。這表明模型在訓練過程中逐漸學習到更好的重建能力，能夠生成與原始高分辨率圖像更接近的圖像。
2. 驗證集PSNR值波動：驗證集的PSNR值在不同的epoch之間會有一定的波動。從33.273到34.049之間的變化幅度不大。這可能表示模型在驗證集上的性能相對穩定，但也有可能是過擬合。

總結來說，根據提供的訓練過程，可以看出模型在訓練過程中逐漸提高了其重建能力，並在驗證集上保持了相對穩定的性能。然而，為了更全面地評估模型的性能，我們需要進一步觀察更多的訓練過程，包括更多的epoch數據以及測試集的PSNR值。

---

另一種可能的觀點是，儘管模型在訓練集上的性能持續提升，但在驗證集上的性能停留不前，這可能暗示著模型發生了過度擬合的現象。過度擬合意味著模型在訓練過程中過分追求對訓練數據的擬合，導致無法很好地泛化到新的數據。

---

## 讀入 pre-trained 的 pth 檔，再進行 3 次 training，並展示其 Train PSNR 與 Val PSNR 值。

```
In [ ]: !python train.py --weights ../outputs/model.pth --epochs 3
```

```
 Loading weights to resume training...
SRCNN(
    (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv6): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv7): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv8): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv9): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv10): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
)
Training samples: 22227
Validation samples: 19
Epoch 1 of 3
100% 174/174 [00:29<00:00, 5.99it/s]
100% 19/19 [00:00<00:00, 20.72it/s]
Train PSNR: 36.524
Val PSNR: 33.678
Saving model...
Epoch 2 of 3
100% 174/174 [00:22<00:00, 7.61it/s]
100% 19/19 [00:00<00:00, 20.86it/s]
Train PSNR: 37.598
Val PSNR: 33.706
Saving model...
Epoch 3 of 3
100% 174/174 [00:22<00:00, 7.75it/s]
100% 19/19 [00:00<00:00, 19.03it/s]
Train PSNR: 37.578
Val PSNR: 33.657
Saving model...
Finished training in: 1.314 minutes
```

當在處理訓練時可能需要中止並重新訓練的情況時，讀入預訓練的 .pth 檔並進行進一步的訓練是一種常見的方法。這樣可以利用之前訓練的模型權重作為起點，節省重新訓練所需的時間和計算資源。

---

## 🏃‍♂️ 測試 Set5 與 Set14 並展示 Test PSNR on Set5 與 Test PSNR on Set14 。

In [ ]: !python test.py

```
100% 5/5 [00:07<00:00, 1.47s/it]
Test PSNR on Set5: 32.425
100% 14/14 [00:00<00:00, 32.27it/s]
Test PSNR on Set14: 28.609
```

在 Set5 上，測試 PSNR 的結果為 32.425，而在 Set14 上，測試 PSNR 的結果為 28.609。

---

## 對已訓練完成的 CNN 模型輸入任何一張影像，並生成相對應的一張高解度影像。

In [ ]: from PIL import Image

```
import glob as glob
import os
import argparse

import torch
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import srcnn

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = srcnn.SRCNN().to(device)

# 載入權重
pretrained_dict = torch.load('/content/20220606_SRCNN_Implementation_in_PyTorch_for_Implementation_in_PyTorch_for_Image_Super_resolution.pt')
model.load_state_dict(pretrained_dict)
model.eval()

# Bicubic插值函數
def bicubic_interpolation(image, scale_factor):
    width, height = image.size
    new_width = int(width * scale_factor)
    new_height = int(height * scale_factor)
    return image.resize((new_width, new_height), Image.BICUBIC)

# 輸入影像路徑
input_image_path = '/content/20220606_SRCNN_Implementation_in_PyTorch_for_Image_Super_resolution.jpg'

# 預處理影像
preprocess = transforms.Compose([
    transforms.ToTensor()
])
input_image = Image.open(input_image_path).convert('RGB')

# 模糊化處理
input_image = bicubic_interpolation(input_image, 0.5)

input_tensor = preprocess(input_image)
input_batch = input_tensor.unsqueeze(0)

# 使用模型進行預測
with torch.no_grad():
    output_batch = model(input_batch.to(device))

# 取得預測結果
output_tensor = output_batch.squeeze(0)
output_image = transforms.ToPILImage()(output_tensor)

result_tensor = torch.add(input_tensor.to(device), output_tensor.to(device))
result_image = transforms.ToPILImage()(result_tensor.squeeze(0))

# 載入原始影像
original_image = Image.open('/content/20220606_SRCNN_Implementation_in_PyTorch_for_Implementation_in_PyTorch_for_Image_Super_resolution.jpg')

# 顯示生成的高解析度影像
plt.figure(figsize=[20, 10])
plt.subplot(1, 4, 1)
plt.title('Original Image')
plt.imshow(original_image)
plt.axis('off')
plt.subplot(1, 4, 2)
plt.title('Bicubic 2x')
```

```
plt.imshow(input_image)
plt.axis('off')
plt.subplot(1, 4, 3)
plt.title('Residual Image')
plt.imshow(output_image)
plt.axis('off')
plt.subplot(1, 4, 4)
plt.title('SRCNN Image')
plt.imshow(result_image)
plt.axis('off')
plt.show()
```



通過比較這些圖像，可以觀察到 SRCNN 模型能夠將低解析度影像恢復為更接近原始影像的高解析度影像。殘差影像呈現出高頻細節的部分，並且通過將其與低解析度影像相加，可以得到更銳利和清晰的 SRCNN 影像（香蕉的貼紙變清楚）。

對於殘差影像中亮度過高的情況，這可能是由於模型在訓練過程中對於高頻細節的恢復過程中產生了一些過度增強的效果。這可能會導致一些區域出現過亮的像素。

---

## 結語：

總結一下，這個專案的目標是深入研究和應用SRCNN模型，這是在圖像超分辨率方面的一個卓越的卷積神經網絡。我們成功地進行了模型訓練、測試以及實際的影像生成，顯示了SRCNN模型在提高圖像品質和清晰度方面的優勢和應用價值。此專案有助於我們更好地了解和應用卷積神經網絡在影像處理中的作用，並為圖像處理領域的進一步研究和應用提供了有價值的參考。

---