第三次作品(專題)(3-1)：淺度機器學習分類器的評比實驗

學號：410978040

姓名：黃冠翔

作品目標：

- 利用多元羅吉斯回歸、支援向量機 SVM 和神經網路對資料進行分類學習與測試。
- 學習分類器的原理並進行評比實驗。

本專題計畫執行分類器比較，即採用三種分類器分別對三組資料進行分類學習與測試。其中分類器包括： (1)多元羅吉斯回歸 (Multinomial Logistic Regression) (2)支援向量機 (Support Vector Machine) (3)神經網路 (Neural Network)

三組資料包括： (1)來自 3 個產區，178 瓶葡萄酒，含 13 種葡萄酒成分。 (2)來自 AT&T 40 個人的人臉影像共 400 張，每張大小 64×64。 (3)來自 Yale Face 38 人的人臉影像共 2410 張，每張大小 192×168。

此檔案先以葡萄酒資料進行分類學習與測試。

先讀取資料並設定變數，同時將資料標準化。

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Read data
df = pd.read_excel('D:\\vs\\venv_name\wine.xlsx')
X = np.array(df.iloc[:, :-1]) # 排除最後一欄標籤
y = np.array(df.iloc[:, -1]) # 標籤欄
# Split data into training and testing data
X_train, X_test, y_train, y_test = train_test_split(
X, y, test_size=0.30)
# Standardize data
scaler = StandardScaler()
X_train_ = scaler.fit_transform(X_train)
X_test_ = scaler.fit_transform(X_test)
```

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

(a)原始資料

1.使用 lbfgs 的演算法

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'lbfgs' # 'lbfgs' is the default
```

```
# solver = ’liblinear’
# solver = ’newton-cg’
clf_original = LogisticRegression(solver = solver, **opts)
clf_original.fit(X_train_, y_train)
y_pred = clf_original.predict(X_test_)
# 測試資料之準確率回報
print(f"{accuracy_score(y_test, y_pred):.2%}\n")
print(f"{clf_original.score(X_test_, y_test):.2%}\n")
print(classification_report(y_test, y_pred))
```

```
96.30%

96.30%

              precision    recall  f1-score   support

           1       1.00      1.00      1.00        19
           2       1.00      0.91      0.95        23
           3       0.86      1.00      0.92        12

    accuracy                           0.96        54
   macro avg       0.95      0.97      0.96        54
weighted avg       0.97      0.96      0.96        54


[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1
concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s finished
```

2.使用 liblinear 的演算法

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'liblinear'
# solver = ’newton-cg’
clf_original = LogisticRegression(solver = solver, **opts)
clf_original.fit(X_train_, y_train)
y_pred = clf_original.predict(X_test_)
# 測試資料之準確率回報
print(f"{accuracy_score(y_test, y_pred):.2%}\n")
print(f"{clf_original.score(X_test_, y_test):.2%}\n")
print(classification_report(y_test, y_pred))
```

```
[LibLinear]96.30%

96.30%

              precision    recall  f1-score   support
```

```
           1       1.00      1.00      1.00        19
           2       1.00      0.91      0.95        23
           3       0.86      1.00      0.92        12

    accuracy                           0.96        54
   macro avg       0.95      0.97      0.96        54
weighted avg       0.97      0.96      0.96        54
```

3.使用 newton-cg 的演算法

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'newton-cg'
clf_original = LogisticRegression(solver = solver, **opts)
clf_original.fit(X_train_, y_train)
y_pred = clf_original.predict(X_test_)
# 測試資料之準確率回報
print(f"{accuracy_score(y_test, y_pred):.2%}\n")
print(f"{clf_original.score(X_test_, y_test):.2%}\n")
print(classification_report(y_test, y_pred))
```

```
96.30%

96.30%

              precision    recall  f1-score   support

           1       1.00      1.00      1.00        19
           2       1.00      0.91      0.95        23
           3       0.86      1.00      0.92        12

    accuracy                           0.96        54
   macro avg       0.95      0.97      0.96        54
weighted avg       0.97      0.96      0.96        54


[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1
concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s finished
```

討論：

- 使用 lbfgs 的演算法時，準確率為 96.30%。
- 使用 liblinear 的演算法時，準確率為 96.30%。
- 使用 newton-cg 的演算法時，準確率為 96.30%。
- 綜上所述，三者準確率相同。

(b)主成分資料

1.取 2 個主成分並使用 lbfgs 的演算法

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 2).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'lbfgs' # ’lbfgs’ is the default
clf_PCA = LogisticRegression(solver = solver, **opts)
clf_PCA.fit(Z_train, y_train)
y_pred = clf_PCA.predict(Z_test)
print(f"{clf_PCA.score(Z_test, y_test):.2%}\n")

98.15%


[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1
concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s finished
```

2.取 4 個主成分並使用 lbfgs 的演算法

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 4).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'lbfgs' # ’lbfgs’ is the default
clf_PCA = LogisticRegression(solver = solver, **opts)
clf_PCA.fit(Z_train, y_train)
y_pred = clf_PCA.predict(Z_test)
print(f"{clf_PCA.score(Z_test, y_test):.2%}\n")

96.30%


[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1
concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s finished
```

3.取 6 個主成分並使用 lbfgs 的演算法

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 6).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'lbfgs' # ’lbfgs’ is the default
```

```
clf_PCA = LogisticRegression(solver = solver, **opts)
clf_PCA.fit(Z_train, y_train)
y_pred = clf_PCA.predict(Z_test)
print(f"{clf_PCA.score(Z_test, y_test):.2%}\n")
```

```
96.30%


[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1
concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s finished
```

4.取 2 個主成分並使用 liblinear 的演算法

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 2).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'liblinear' # ’lbfgs’ is the default
clf_PCA = LogisticRegression(solver = solver, **opts)
clf_PCA.fit(Z_train, y_train)
y_pred = clf_PCA.predict(Z_test)
print(f"{clf_PCA.score(Z_test, y_test):.2%}\n")
```

```
[LibLinear]96.30%
```

5.取 4 個主成分並使用 liblinear 的演算法

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 4).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'liblinear' # ’lbfgs’ is the default
clf_PCA = LogisticRegression(solver = solver, **opts)
clf_PCA.fit(Z_train, y_train)
y_pred = clf_PCA.predict(Z_test)
print(f"{clf_PCA.score(Z_test, y_test):.2%}\n")
```

```
[LibLinear]96.30%
```

6.取 6 個主成分並使用 liblinear 的演算法

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 6).fit(X_train_)
Z_train = pca.transform(X_train_)
```

```
Z_test = pca.transform(X_test_)
opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'liblinear' # ’lbfgs’ is the default
clf_PCA = LogisticRegression(solver = solver, **opts)
clf_PCA.fit(Z_train, y_train)
y_pred = clf_PCA.predict(Z_test)
print(f"{clf_PCA.score(Z_test, y_test):.2%}\n")
```

```
[LibLinear]96.30%
```

7.取2個主成分並使用newton-cg的演算法

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 6).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'newton-cg' # ’lbfgs’ is the default
clf_PCA = LogisticRegression(solver = solver, **opts)
clf_PCA.fit(Z_train, y_train)
y_pred = clf_PCA.predict(Z_test)
print(f"{clf_PCA.score(Z_test, y_test):.2%}\n")
```

```
96.30%


[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1
concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s finished
```

8.取4個主成分並使用newton-cg的演算法

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 6).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'newton-cg' # ’lbfgs’ is the default
clf_PCA = LogisticRegression(solver = solver, **opts)
clf_PCA.fit(Z_train, y_train)
y_pred = clf_PCA.predict(Z_test)
print(f"{clf_PCA.score(Z_test, y_test):.2%}\n")
```

```
96.30%


[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1
concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s finished
```

9.取 6 個主成分並使用 newton-cg 的演算法

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 6).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
opts = dict(tol = 1e-6, max_iter = int(1e6), verbose=1)
solver = 'newton-cg' # ’lbfgs’ is the default
clf_PCA = LogisticRegression(solver = solver, **opts)
clf_PCA.fit(Z_train, y_train)
y_pred = clf_PCA.predict(Z_test)
print(f"{clf_PCA.score(Z_test, y_test):.2%}\n")
```

```
96.30%


[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1
concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s finished
```

討論：

使用 lbfgs 的演算法：

- 取 2 個主成分時，準確率為 98.15%。
- 取 4 個主成分時，準確率為 96.30%。
- 取 6 個主成分時，準確率為 96.30%。

使用 liblinear 的演算法：

- 取 2 個主成分時，準確率為 96.30%。
- 取 4 個主成分時，準確率為 96.30%。
- 取 6 個主成分時，準確率為 96.30%。

使用 newton-cg 的演算法：

- 取 2 個主成分時，準確率為 96.30%。
- 取 4 個主成分時，準確率為 96.30%。
- 取 6 個主成分時，準確率為 96.30%。

小結：

- 使用 lbfgs 的演算且取 2 個主成分時，準確率最高(98.15%)，其餘演算法之準確率皆相同(96.30%)。

(2)支援向量機 (Support Vector Machine)

(a)原始資料

1.使用 kernel="linear"

```python
from sklearn.svm import SVC, LinearSVC
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
# opts = dict(C = C, decision_function_shape = ’ovo’, \
# tol = 1e−6, max_iter = int(1e6))
clf_svm = SVC(kernel="linear", **opts)
# clf_svm = SVC(kernel=”rbf”, gamma=0.2, **opts)
# clf_svm = SVC(kernel=”poly”, degree=3, gamma=”auto”, **opts)
# clf_svm = LinearSVC(**opts) # one vs the rest
clf_svm.fit(X_train_, y_train)
predictions = clf_svm.predict(X_test_)
print(f"{accuracy_score(y_test, predictions):.2%}\n")
print(classification_report(y_test, predictions))
```

```
94.44%

              precision    recall  f1-score   support

           1       1.00      0.95      0.97        19
           2       0.95      0.91      0.93        23
           3       0.86      1.00      0.92        12

    accuracy                           0.94        54
   macro avg       0.94      0.95      0.94        54
weighted avg       0.95      0.94      0.95        54
```

2.使用 kernel="rbf"

```python
from sklearn.svm import SVC, LinearSVC
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
# opts = dict(C = C, decision_function_shape = ’ovo’, \
# tol = 1e−6, max_iter = int(1e6))
#clf_svm = SVC(kernel="linear", **opts)
clf_svm = SVC(kernel="rbf", gamma=0.2, **opts)
# clf_svm = SVC(kernel=”poly”, degree=3, gamma=”auto”, **opts)
# clf_svm = LinearSVC(**opts) # one vs the rest
clf_svm.fit(X_train_, y_train)
predictions = clf_svm.predict(X_test_)
print(f"{accuracy_score(y_test, predictions):.2%}\n")
print(classification_report(y_test, predictions))
```

```
87.04%

              precision    recall  f1-score   support

           1       1.00      0.74      0.85        19
           2       0.79      0.96      0.86        23
           3       0.92      0.92      0.92        12
```

```
    accuracy                           0.87        54
   macro avg       0.90      0.87      0.88        54
weighted avg       0.89      0.87      0.87        54
```

3.使用 kernel="poly"

```
from sklearn.svm import SVC, LinearSVC
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
# opts = dict(C = C, decision_function_shape = ’ovo’, \
# tol = 1e-6, max_iter = int(1e6))
#clf_svm = SVC(kernel="linear", **opts)
# clf_svm = SVC(kernel=”rbf”, gamma=0.2, **opts)
clf_svm = SVC(kernel="poly", degree=3, gamma="auto", **opts)
# clf_svm = LinearSVC(**opts) # one vs the rest
clf_svm.fit(X_train_, y_train)
predictions = clf_svm.predict(X_test_)
print(f"{accuracy_score(y_test, predictions):.2%}\n")
print(classification_report(y_test, predictions))
```

```
98.15%

              precision    recall  f1-score   support

           1       1.00      0.95      0.97        19
           2       0.96      1.00      0.98        23
           3       1.00      1.00      1.00        12

    accuracy                           0.98        54
   macro avg       0.99      0.98      0.98        54
weighted avg       0.98      0.98      0.98        54
```

討論：

- 使用 kernel="linear"時，準確率為 94.44%。
- 使用 kernel="rbf"時，準確率為 87.04%。
- 使用 kernel="poly"時，準確率為 98.15%。
- 綜上所述，kernel="poly"之準確率最高，kernel="linear"次之，kernel="rbf"之準確率最低。

(b)主成分資料

1.取 2 個主成分並使用 kernel="linear"

```
from sklearn.decomposition import PCA
from sklearn.svm import SVC, LinearSVC
```

```
pca = PCA(n_components = 2).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
clf_svm = SVC(kernel="linear", **opts)
clf_svm.fit(Z_train, y_train)
predictions = clf_svm.predict(Z_test)
print(f"{clf_svm.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

```
98.15%

              precision    recall  f1-score   support

           1       1.00      1.00      1.00        19
           2       1.00      0.96      0.98        23
           3       0.92      1.00      0.96        12

    accuracy                           0.98        54
   macro avg       0.97      0.99      0.98        54
weighted avg       0.98      0.98      0.98        54
```

2.取 4 個主成分並使用 kernel="linear"

```
from sklearn.decomposition import PCA
from sklearn.svm import SVC, LinearSVC

pca = PCA(n_components = 4).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
clf_svm = SVC(kernel="linear", **opts)
clf_svm.fit(Z_train, y_train)
predictions = clf_svm.predict(Z_test)
print(f"{clf_svm.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

```
96.30%

              precision    recall  f1-score   support

           1       1.00      1.00      1.00        19
           2       1.00      0.91      0.95        23
           3       0.86      1.00      0.92        12

    accuracy                           0.96        54
   macro avg       0.95      0.97      0.96        54
```

```
weighted avg       0.97      0.96      0.96        54
```

3.取 6 個主成分並使用 kernel="linear"

```
from sklearn.decomposition import PCA
from sklearn.svm import SVC, LinearSVC

pca = PCA(n_components = 6).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
clf_svm = SVC(kernel="linear", **opts)
clf_svm.fit(Z_train, y_train)
predictions = clf_svm.predict(Z_test)
print(f"{clf_svm.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

```
94.44%

              precision    recall  f1-score   support

           1       0.95      1.00      0.97        19
           2       1.00      0.87      0.93        23
           3       0.86      1.00      0.92        12

    accuracy                           0.94        54
   macro avg       0.94      0.96      0.94        54
weighted avg       0.95      0.94      0.94        54
```

4.取 2 個主成分並使用 kernel="rbf"

```
from sklearn.decomposition import PCA
from sklearn.svm import SVC, LinearSVC

pca = PCA(n_components = 2).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
clf_svm = SVC(kernel="rbf", **opts)
clf_svm.fit(Z_train, y_train)
predictions = clf_svm.predict(Z_test)
print(f"{clf_svm.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

```
98.15%
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        19
           2       1.00      0.96      0.98        23
           3       0.92      1.00      0.96        12

    accuracy                           0.98        54
   macro avg       0.97      0.99      0.98        54
weighted avg       0.98      0.98      0.98        54
```

5.取 4 個主成分並使用 kernel="rbf"

```
from sklearn.decomposition import PCA
from sklearn.svm import SVC, LinearSVC

pca = PCA(n_components = 4).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
clf_svm = SVC(kernel="rbf", **opts)
clf_svm.fit(Z_train, y_train)
predictions = clf_svm.predict(Z_test)
print(f"{clf_svm.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

```
92.59%

              precision    recall  f1-score   support

           1       0.95      0.95      0.95        19
           2       0.95      0.87      0.91        23
           3       0.86      1.00      0.92        12

    accuracy                           0.93        54
   macro avg       0.92      0.94      0.93        54
weighted avg       0.93      0.93      0.93        54
```

6.取 6 個主成分並使用 kernel="rbf"

```
from sklearn.decomposition import PCA
from sklearn.svm import SVC, LinearSVC

pca = PCA(n_components = 6).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
```

```
clf_svm = SVC(kernel="rbf", **opts)
clf_svm.fit(Z_train, y_train)
predictions = clf_svm.predict(Z_test)
print(f"{clf_svm.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

```
94.44%

              precision    recall  f1-score   support

           1       1.00      0.95      0.97        19
           2       0.95      0.91      0.93        23
           3       0.86      1.00      0.92        12

    accuracy                           0.94        54
   macro avg       0.94      0.95      0.94        54
weighted avg       0.95      0.94      0.95        54
```

7.取 2 個主成分並使用 kernel="poly"

```
from sklearn.decomposition import PCA
from sklearn.svm import SVC, LinearSVC

pca = PCA(n_components = 2).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
clf_svm = SVC(kernel="poly", **opts)
clf_svm.fit(Z_train, y_train)
predictions = clf_svm.predict(Z_test)
print(f"{clf_svm.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

```
88.89%

              precision    recall  f1-score   support

           1       1.00      0.74      0.85        19
           2       0.81      0.96      0.88        23
           3       0.92      1.00      0.96        12

    accuracy                           0.89        54
   macro avg       0.91      0.90      0.90        54
weighted avg       0.90      0.89      0.89        54
```

8.取 4 個主成分並使用 kernel="poly"

```
from sklearn.decomposition import PCA
from sklearn.svm import SVC, LinearSVC

pca = PCA(n_components = 4).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
clf_svm = SVC(kernel="poly", **opts)
clf_svm.fit(Z_train, y_train)
predictions = clf_svm.predict(Z_test)
print(f"{clf_svm.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

```
96.30%

              precision    recall  f1-score   support

           1       1.00      0.95      0.97        19
           2       0.96      0.96      0.96        23
           3       0.92      1.00      0.96        12

    accuracy                           0.96        54
   macro avg       0.96      0.97      0.96        54
weighted avg       0.96      0.96      0.96        54
```

9.取 6 個主成分並使用 kernel="poly"

```
from sklearn.decomposition import PCA
from sklearn.svm import SVC, LinearSVC

pca = PCA(n_components = 6).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)
C = 1 # SVM regularization parameter
opts = dict(C = C, tol = 1e-6, max_iter = int(1e6))
clf_svm = SVC(kernel="poly", **opts)
clf_svm.fit(Z_train, y_train)
predictions = clf_svm.predict(Z_test)
print(f"{clf_svm.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

```
100.00%

              precision    recall  f1-score   support

           1       1.00      1.00      1.00        19
           2       1.00      1.00      1.00        23
           3       1.00      1.00      1.00        12
```

```
    accuracy                           1.00        54
   macro avg       1.00      1.00      1.00        54
weighted avg       1.00      1.00      1.00        54
```

討論：

使用 kernel="linear"：

- 取 2 個主成分時，準確率為 98.15%。
- 取 4 個主成分時，準確率為 96.30%。
- 取 6 個主成分時，準確率為 94.44%。

使用 kernel="rbf"：

- 取 2 個主成分時，準確率為 98.15%。
- 取 4 個主成分時，準確率為 92.59%。
- 取 6 個主成分時，準確率為 94.44%。

使用 kernel="poly"：

- 取 2 個主成分時，準確率為 88.89%。
- 取 4 個主成分時，準確率為 96.30%。
- 取 6 個主成分時，準確率為 100.00%。

小結：

- 使用 kernel="poly"且取 6 個主成分時，準確率最高(100.00%)。
- 使用 kernel="poly"且取 2 個主成分時，準確率最低(88.89)。

(3)神經網路 (Neural Network)

(a)原始資料

1.使用 activation = 'logistic'且 hidden_layers = (30,)

```
from sklearn.neural_network import MLPClassifier
# hidden_layers = (512,) # one hidden layer
# activation = ’relu’ # the default
hidden_layers = (30,)
activation = 'logistic'
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 here
solver = 'adam' # default solver
clf_MLP = MLPClassifier(solver = solver, **opts)
clf_MLP.fit(X_train_, y_train)
predictions = clf_MLP.predict(X_test_)
```

```
print(f"{accuracy_score(y_test, predictions):.2%}\n")
print(classification_report(y_test, predictions))
```

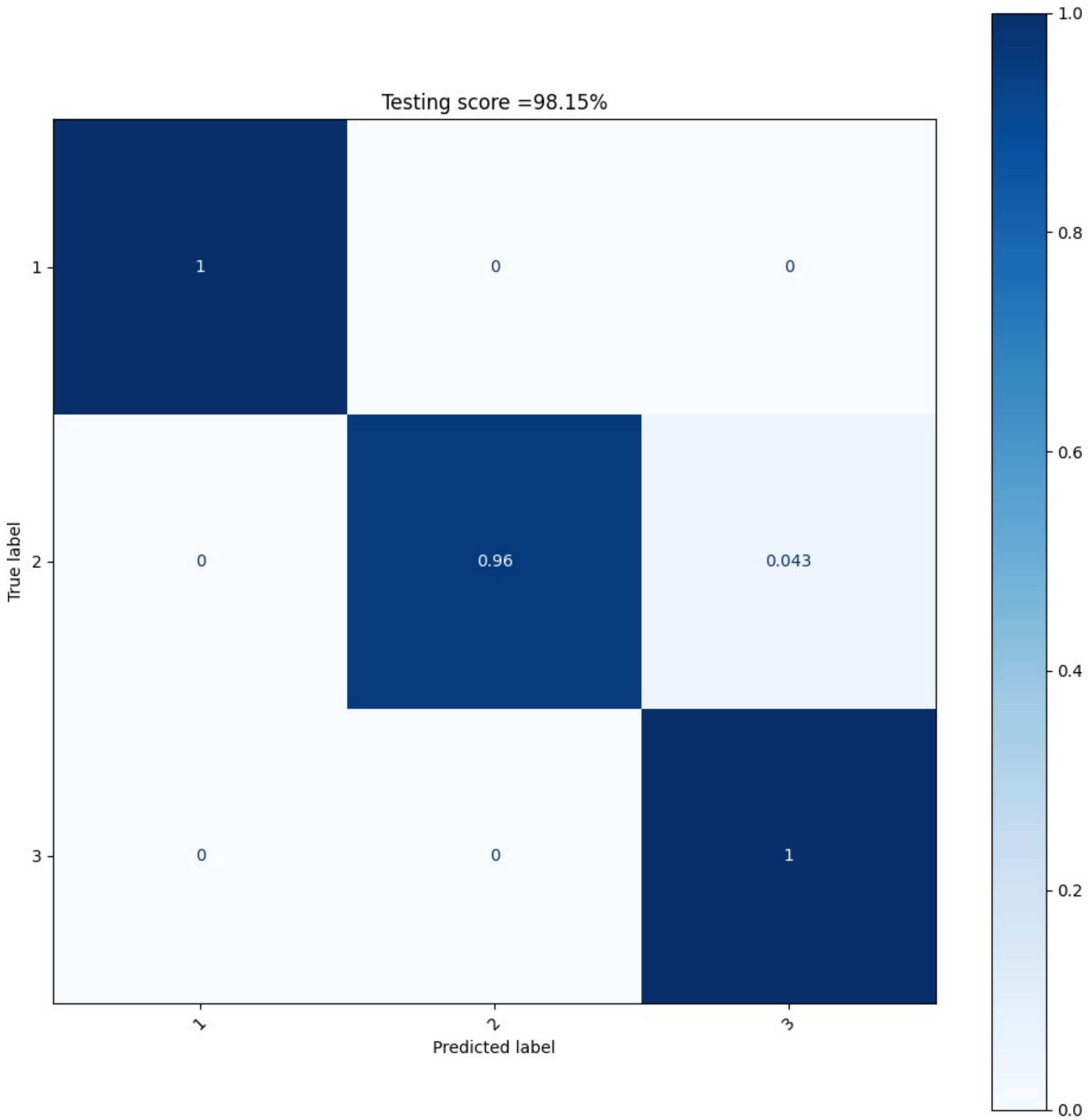
```
98.15%

              precision    recall  f1-score   support

           1       1.00      1.00      1.00        19
           2       1.00      0.96      0.98        23
           3       0.92      1.00      0.96        12

    accuracy                           0.98        54
   macro avg       0.97      0.99      0.98        54
weighted avg       0.98      0.98      0.98        54
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
fig, ax = plt.subplots(1, 1, figsize=(12,12))
score = 100*clf_MLP.score(X_test_, y_test)
title = 'Testing score ={:.2f}%'.format(score)
disp = ConfusionMatrixDisplay.from_estimator(
clf_MLP,
X_test_,
y_test,
xticks_rotation=45, #’vertical’,
# display_labels=class_names,
cmap=plt.cm.Blues,
normalize='true',
ax = ax
)
disp.ax_.set_title(title)
plt.show()
```



2.使用 activation = 'relu'且 hidden_layers = (512,)

```
from sklearn.neural_network import MLPClassifier
hidden_layers = (512,) # one hidden layer
activation = 'relu' # the default
# hidden_layers = (30,)
# activation = 'logistic'
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 here
```

```
solver = 'adam' # default solver
clf_MLP = MLPClassifier(solver = solver, **opts)
clf_MLP.fit(X_train_, y_train)
predictions = clf_MLP.predict(X_test_)
print(f"{accuracy_score(y_test, predictions):.2%}\n")
print(classification_report(y_test, predictions))
```

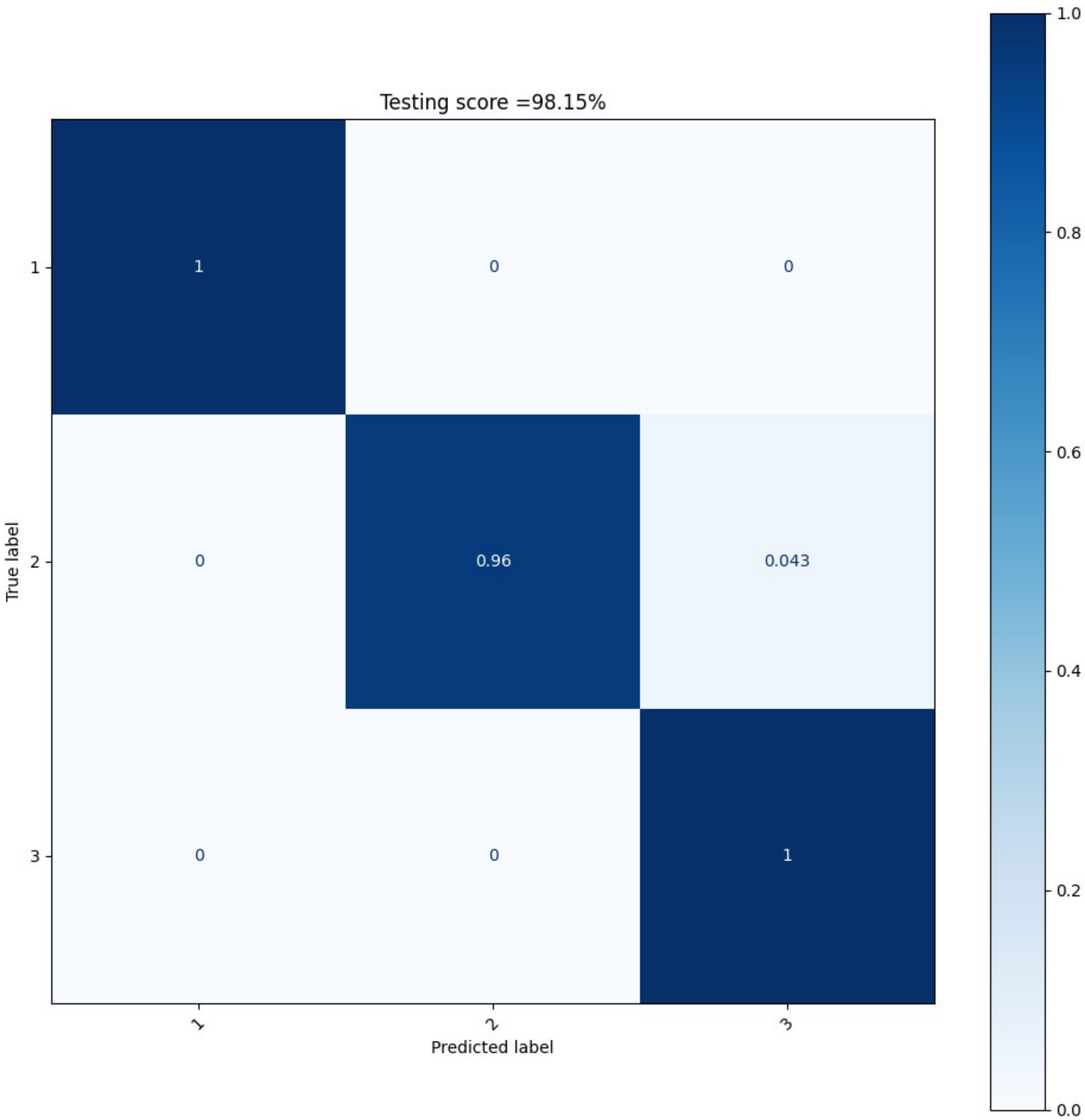
```
98.15%

              precision    recall  f1-score   support

           1       1.00      1.00      1.00        19
           2       1.00      0.96      0.98        23
           3       0.92      1.00      0.96        12

    accuracy                           0.98        54
   macro avg       0.97      0.99      0.98        54
weighted avg       0.98      0.98      0.98        54
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
fig, ax = plt.subplots(1, 1, figsize=(12,12))
score = 100*clf_MLP.score(X_test_, y_test)
title = 'Testing score ={:.2f}%'.format(score)
disp = ConfusionMatrixDisplay.from_estimator(
clf_MLP,
X_test_,
y_test,
xticks_rotation=45, #’vertical’,
# display_labels=class_names,
cmap=plt.cm.Blues,
normalize='true',
ax = ax
)
disp.ax_.set_title(title)
plt.show()
```



討論：

- 使用 activation = 'logistic'且 hidden_layers = (30,)時，準確率為 98.15%。
- 使用 activation = 'relu'且 hidden_layers = (512,)時，準確率為 98.15%。
- 綜上所述，兩者準確率相同。

(b)主成分資料

1.取 2 個主成分並使用 activation = 'logistic'且 hidden_layers = (30,)

```
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier
```

```
pca = PCA(n_components = 2).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)

hidden_layers = (30,)
activation = 'logistic'
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 here
solver = 'adam' # default solver
clf_MLP = MLPClassifier(solver = solver, **opts)
clf_MLP.fit(Z_train, y_train)
predictions = clf_MLP.predict(Z_test)
print(f"{clf_MLP.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

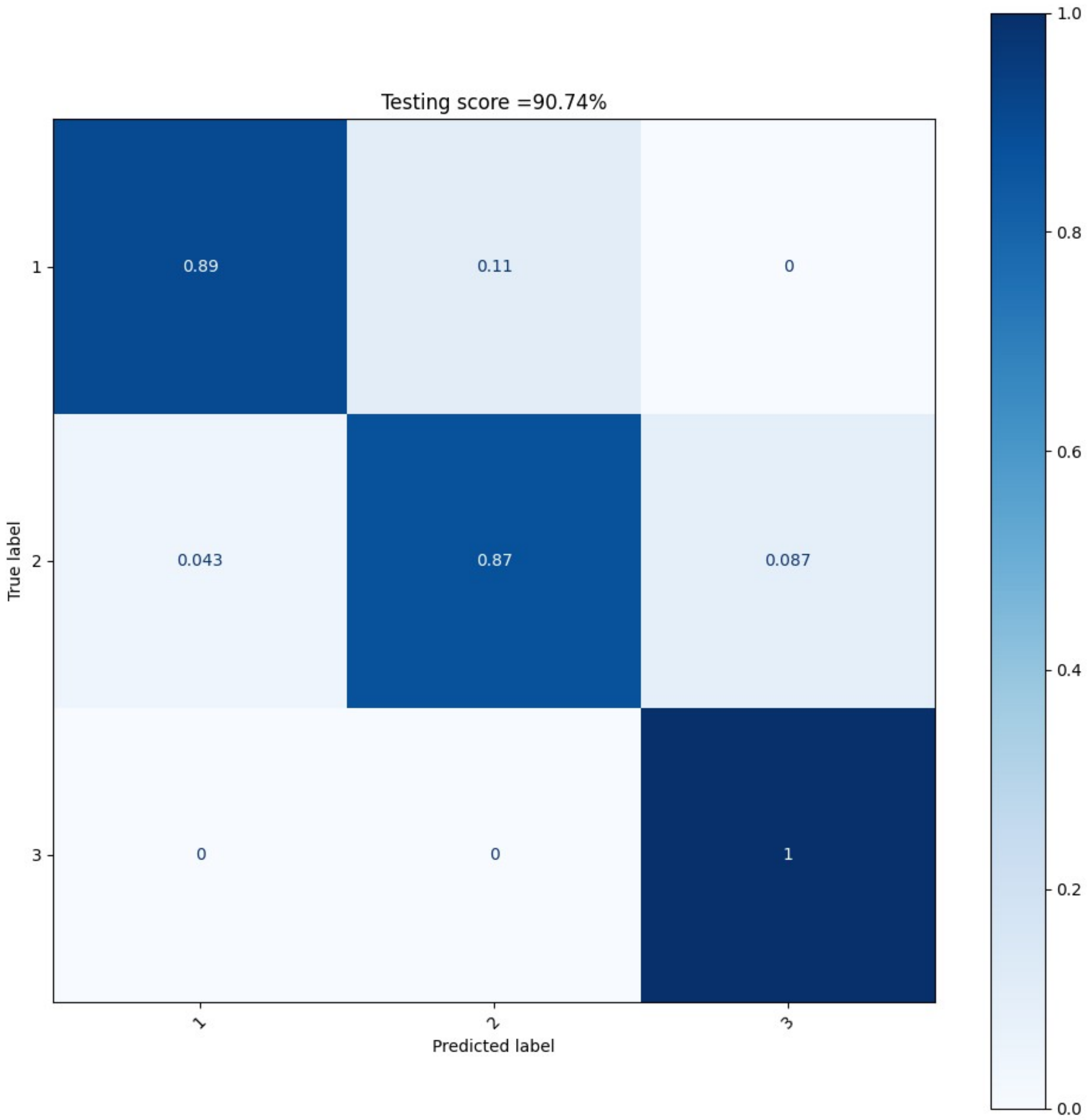
```
90.74%

              precision    recall  f1-score   support

           1       0.94      0.89      0.92        19
           2       0.91      0.87      0.89        23
           3       0.86      1.00      0.92        12

    accuracy                           0.91        54
   macro avg       0.90      0.92      0.91        54
weighted avg       0.91      0.91      0.91        54
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
fig, ax = plt.subplots(1, 1, figsize=(12,12))
score = 100*clf_MLP.score(Z_test, y_test)
title = 'Testing score ={:.2f}%'.format(score)
disp = ConfusionMatrixDisplay.from_estimator(
clf_MLP,
Z_test,
y_test,
xticks_rotation=45, #’vertical’,
# display_labels=class_names,
cmap=plt.cm.Blues,
normalize='true',
ax = ax
)
disp.ax_.set_title(title)
plt.show()
```



2.取 4 個主成分並使用 activation = 'logistic'且 hidden_layers = (30,)

```
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier

pca = PCA(n_components = 4).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)

hidden_layers = (30,)
activation = 'logistic'
```

```
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 here
solver = 'adam' # default solver
clf_MLP = MLPClassifier(solver = solver, **opts)
clf_MLP.fit(Z_train, y_train)
predictions = clf_MLP.predict(Z_test)
print(f"{clf_MLP.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

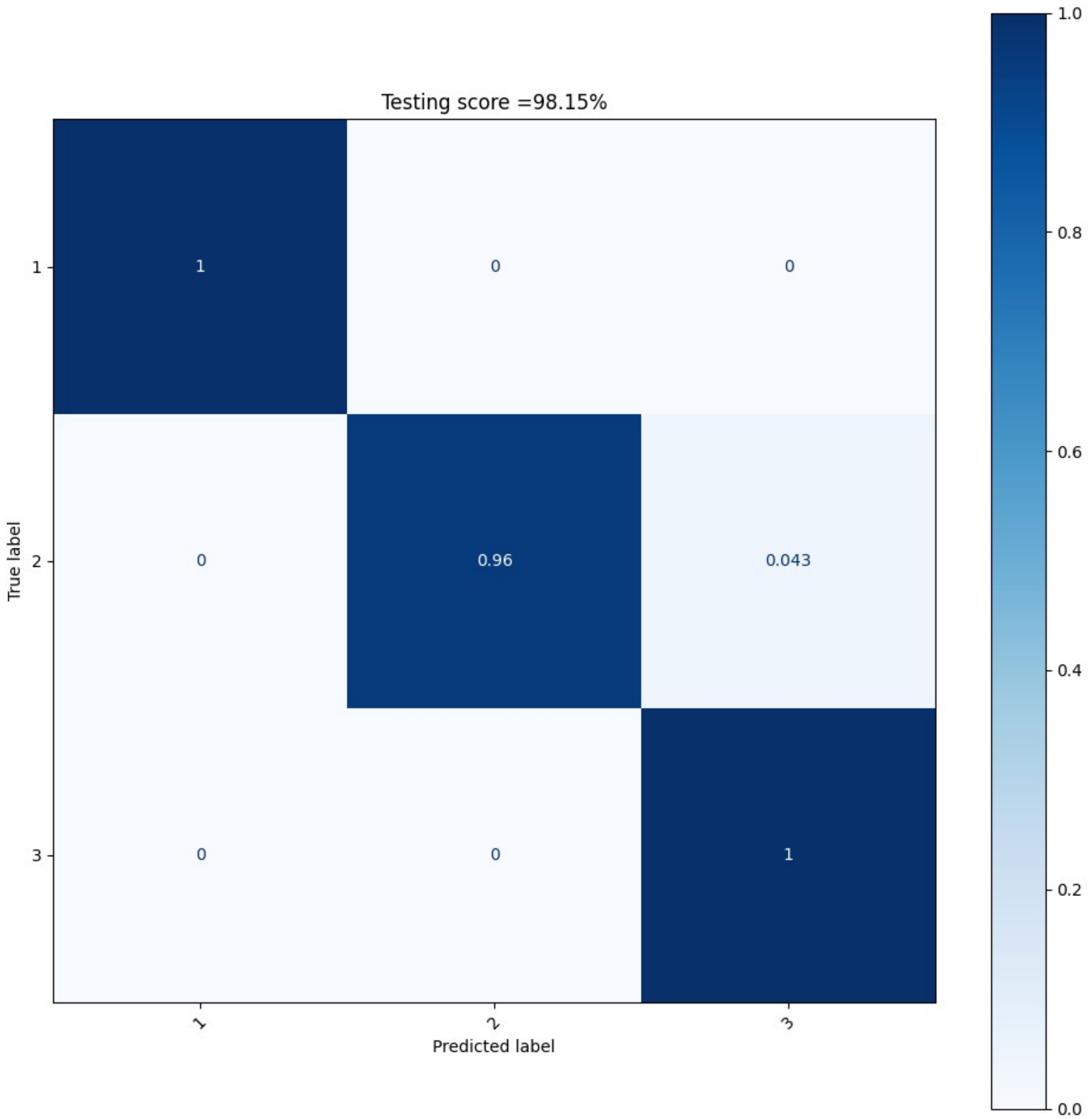
```
98.15%

              precision    recall  f1-score   support

           1       1.00      1.00      1.00        19
           2       1.00      0.96      0.98        23
           3       0.92      1.00      0.96        12

    accuracy                           0.98        54
   macro avg       0.97      0.99      0.98        54
weighted avg       0.98      0.98      0.98        54
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
fig, ax = plt.subplots(1, 1, figsize=(12,12))
score = 100*clf_MLP.score(Z_test, y_test)
title = 'Testing score ={:.2f}%'.format(score)
disp = ConfusionMatrixDisplay.from_estimator(
clf_MLP,
Z_test,
y_test,
xticks_rotation=45, #’vertical’,
# display_labels=class_names,
cmap=plt.cm.Blues,
normalize='true',
ax = ax
)
disp.ax_.set_title(title)
plt.show()
```



3.取 6 個主成分並使用 activation = 'logistic'且 hidden_layers = (30,)

```
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier

pca = PCA(n_components = 6).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)

hidden_layers = (30,)
activation = 'logistic'
```

```
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 here
solver = 'adam' # default solver
clf_MLP = MLPClassifier(solver = solver, **opts)
clf_MLP.fit(Z_train, y_train)
predictions = clf_MLP.predict(Z_test)
print(f"{clf_MLP.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

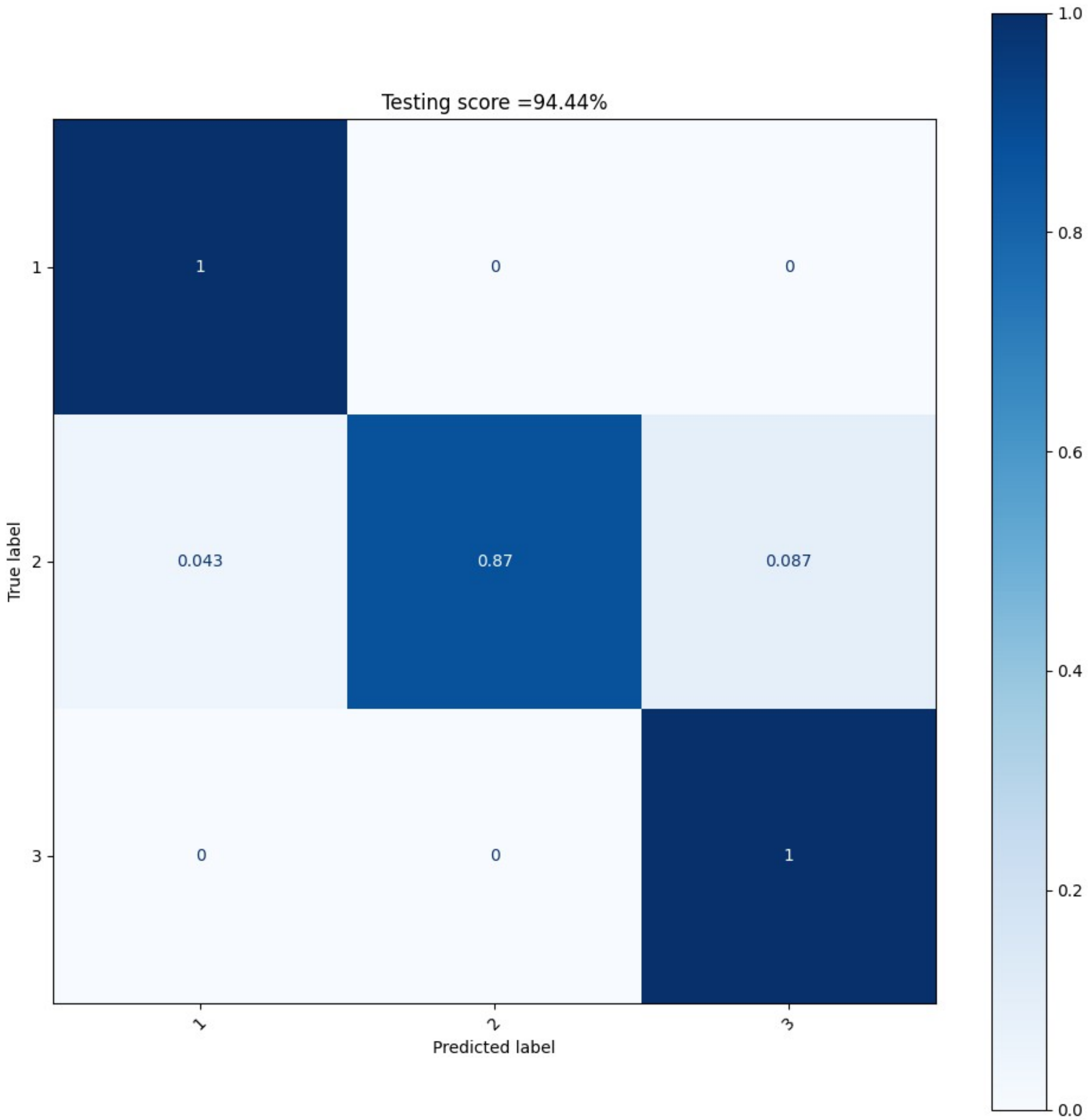
```
94.44%

              precision    recall  f1-score   support

           1       0.95      1.00      0.97        19
           2       1.00      0.87      0.93        23
           3       0.86      1.00      0.92        12

    accuracy                           0.94        54
   macro avg       0.94      0.96      0.94        54
weighted avg       0.95      0.94      0.94        54
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
fig, ax = plt.subplots(1, 1, figsize=(12,12))
score = 100*clf_MLP.score(Z_test, y_test)
title = 'Testing score ={:.2f}%'.format(score)
disp = ConfusionMatrixDisplay.from_estimator(
clf_MLP,
Z_test,
y_test,
xticks_rotation=45, #’vertical’,
# display_labels=class_names,
cmap=plt.cm.Blues,
normalize='true',
ax = ax
)
disp.ax_.set_title(title)
plt.show()
```



4.取 2 個主成分並使用使用 activation = 'relu'且 hidden_layers = (512,)

```
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier

pca = PCA(n_components = 2).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)

hidden_layers = (512,)
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 here
solver = 'adam' # default solver
clf_MLP = MLPClassifier(solver = solver, **opts)
clf_MLP.fit(Z_train, y_train)
predictions = clf_MLP.predict(Z_test)
print(f"{clf_MLP.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

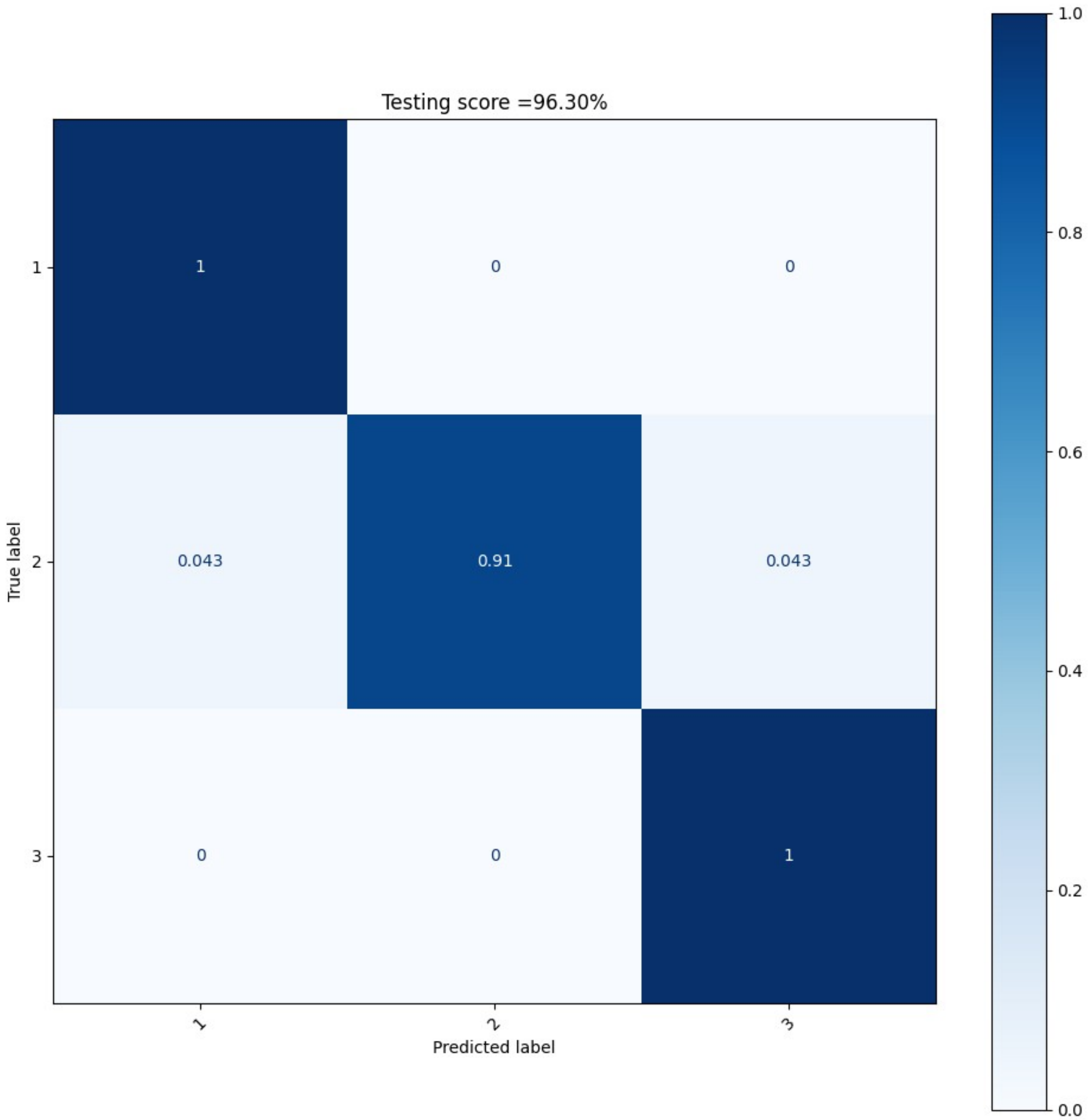
```
96.30%

              precision    recall  f1-score   support

           1       0.95      1.00      0.97        19
           2       1.00      0.91      0.95        23
           3       0.92      1.00      0.96        12

    accuracy                           0.96        54
   macro avg       0.96      0.97      0.96        54
weighted avg       0.97      0.96      0.96        54
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
fig, ax = plt.subplots(1, 1, figsize=(12,12))
score = 100*clf_MLP.score(Z_test, y_test)
title = 'Testing score ={:.2f}%'.format(score)
disp = ConfusionMatrixDisplay.from_estimator(
clf_MLP,
Z_test,
y_test,
xticks_rotation=45, #’vertical’,
# display_labels=class_names,
cmap=plt.cm.Blues,
normalize='true',
ax = ax
)
disp.ax_.set_title(title)
plt.show()
```



5.取 4 個主成分並使用使用 activation = 'relu'且 hidden_layers = (512,)

```
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier

pca = PCA(n_components = 4).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)

hidden_layers = (512,)
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 here
solver = 'adam' # default solver
clf_MLP = MLPClassifier(solver = solver, **opts)
clf_MLP.fit(Z_train, y_train)
predictions = clf_MLP.predict(Z_test)
print(f"{clf_MLP.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

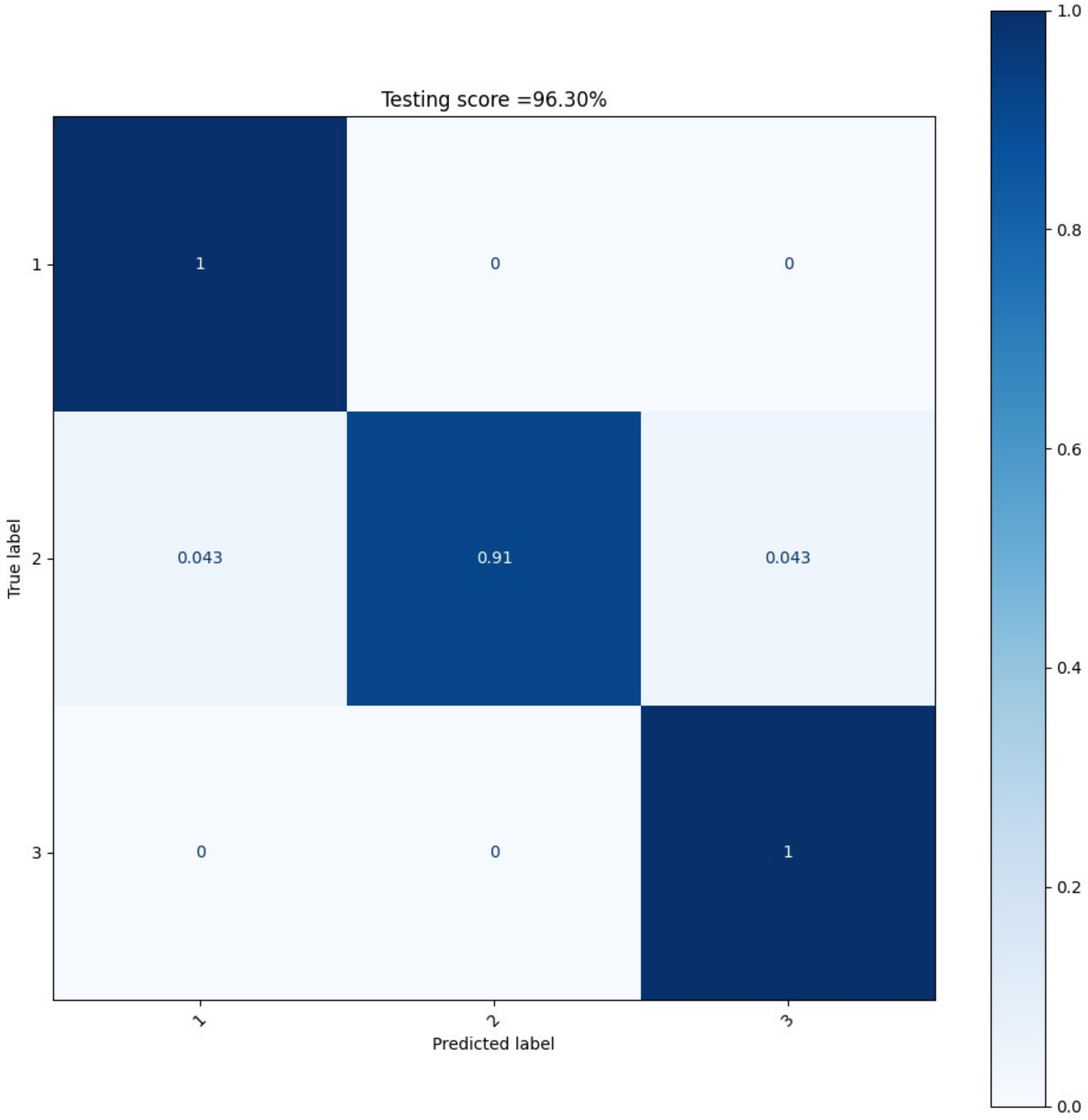
```
96.30%

              precision    recall  f1-score   support

           1       0.95      1.00      0.97        19
           2       1.00      0.91      0.95        23
           3       0.92      1.00      0.96        12

    accuracy                           0.96        54
   macro avg       0.96      0.97      0.96        54
weighted avg       0.97      0.96      0.96        54
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
fig, ax = plt.subplots(1, 1, figsize=(12,12))
score = 100*clf_MLP.score(Z_test, y_test)
title = 'Testing score ={:.2f}%'.format(score)
disp = ConfusionMatrixDisplay.from_estimator(
clf_MLP,
Z_test,
y_test,
xticks_rotation=45, #’vertical’,
# display_labels=class_names,
cmap=plt.cm.Blues,
normalize='true',
ax = ax
)
disp.ax_.set_title(title)
plt.show()
```



6.取 6 個主成分並使用使用 activation = 'relu'且 hidden_layers = (512,)

```
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier

pca = PCA(n_components = 6).fit(X_train_)
Z_train = pca.transform(X_train_)
Z_test = pca.transform(X_test_)

hidden_layers = (512,)
activation = 'relu'
```

```python
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 here
solver = 'adam' # default solver
clf_MLP = MLPClassifier(solver = solver, **opts)
clf_MLP.fit(Z_train, y_train)
predictions = clf_MLP.predict(Z_test)
print(f"{clf_MLP.score(Z_test, y_test):.2%}\n")
print(classification_report(y_test, predictions))
```

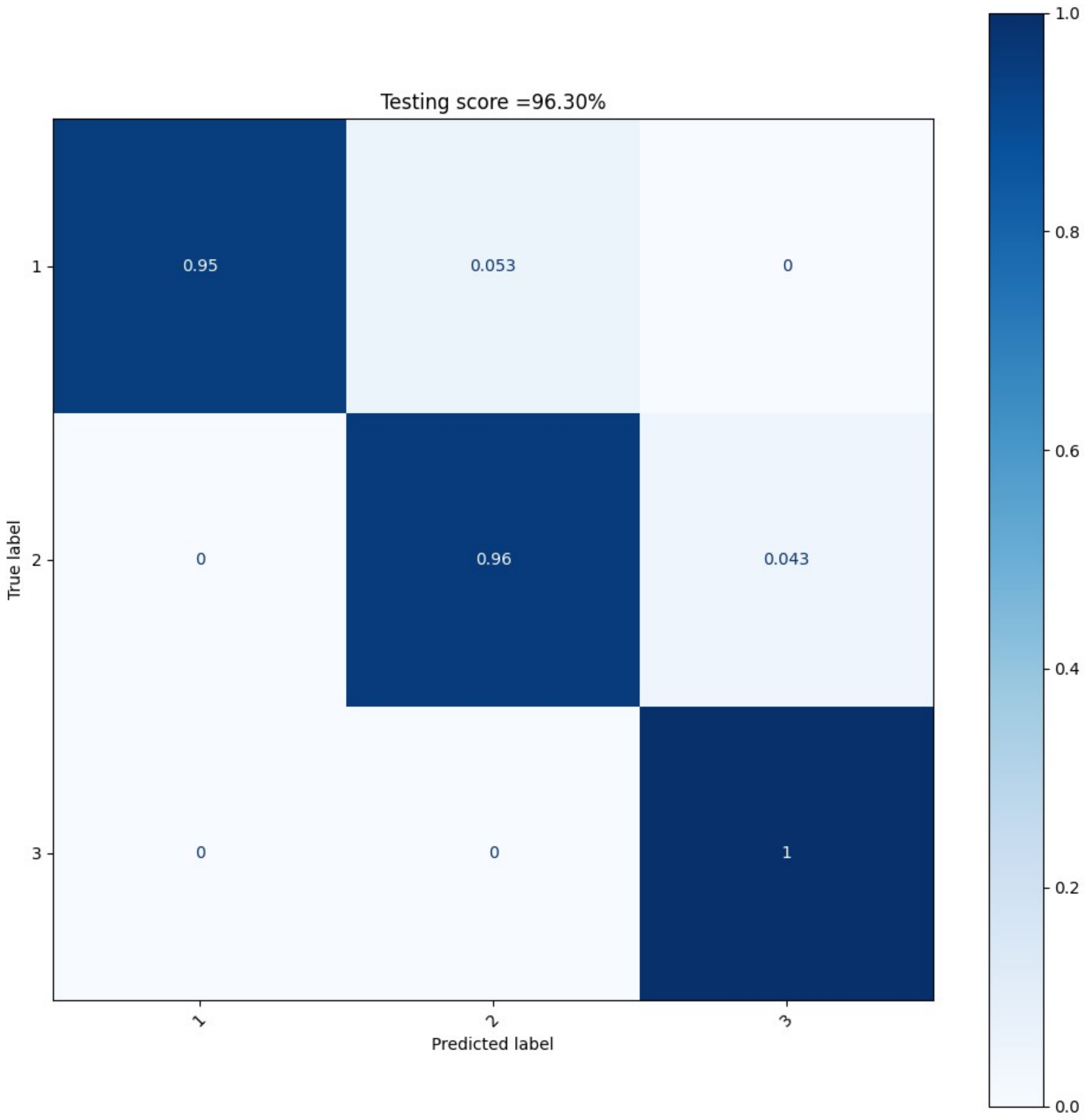
```
96.30%

              precision    recall  f1-score   support

           1       1.00      0.95      0.97        19
           2       0.96      0.96      0.96        23
           3       0.92      1.00      0.96        12

    accuracy                           0.96        54
   macro avg       0.96      0.97      0.96        54
weighted avg       0.96      0.96      0.96        54
```

```python
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
fig, ax = plt.subplots(1, 1, figsize=(12,12))
score = 100*clf_MLP.score(Z_test, y_test)
title = 'Testing score ={:.2f}%'.format(score)
disp = ConfusionMatrixDisplay.from_estimator(
clf_MLP,
Z_test,
y_test,
xticks_rotation=45, #’vertical’,
# display_labels=class_names,
cmap=plt.cm.Blues,
normalize='true',
ax = ax
)
disp.ax_.set_title(title)
plt.show()
```



討論：

使用 activation = 'logistic'且 hidden_layers = (30,)：

- 取 2 個主成分時，準確率為 90.74%。
- 取 4 個主成分時，準確率為 98.15%。
- 取 6 個主成分時，準確率為 94.44%。

使用 activation = 'relu'且 hidden_layers = (512,)：

- 取 2 個主成分時，準確率為 96.30%。
- 取 4 個主成分時，準確率為 96.30%。

- 取 6 個主成分時，準確率為 96.30%。

小結：

- 使用 activation = 'logistic'和 hidden_layers = (30,)且取 4 個主成分時，準確率最高(98.15%)。
- 使用 activation = 'relu'且 hidden_layers = (512,)時，無論取幾個主成分，準確率相同。

總結：

依照準確率比較：

- 多元羅吉斯回歸 (Multinomial Logistic Regression) (1)無論是何種演算法，原始資料的準確率皆為 96.30%。 (2)在主成分資料中，大部分的準確率為 96.30&。 (3)使用 lbfgs 演算法且取 2 個主成分有最高的準確率 98.15%。

- 支援向量機 (Support Vector Machine) (1)在主成分資料中，使用 kernel='poly'時，取愈多主成分，準確率愈高。 (2)在主成分資料中，使用 kernel='linear'或 kernel='rbf'時，取愈多主成分，準確率卻不一定愈高。 (3)取 6 個主成分且使用 kernel='poly'有最高的準確率 100.00%。

- 神經網路 (Neural Network) (1)在原始資料中無論是使用 activation = 'logistic'且 hidden_layers = (30,)或使用 activation = 'relu'且 hidden_layers = (512,)，準確率皆相同。 (2)在主成分資料中，無論是何種 activation 和 hidden_layers，準確率皆比原始資料低。 (3)原始資料和使用 activation = 'logistic'且 hidden_layers = (30,)且取 4 個主成分有最高的準確率 98.15%。

綜上所述：

- 我認為最佳分類器為支援向量機中取 6 個主成分且使用 kernel='poly'，因為它的準確率為所有分類器中最高(100.00%)。
- 由於資料樣本數較小，因此無論何種分類器執行時間都很短。