

```

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

#Visualization
import matplotlib.pyplot as plt

#Machine Learning
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn import metrics

#System
import os
print(os.listdir("../input"))

❸ ['olivetti_faces_target.npy', 'olivetti_faces.npy']

import warnings
warnings.filterwarnings('ignore')
print("Warnings ignored!!")

Warnings ignored!!

```

[Go to Contents Menu](#) | Quick Links: [1.](#) | [3.](#) | [3.1.](#) | [3.2.](#) | [4.](#) | [4.1.](#) | [4.2.](#) | [4.3.](#) | [4.4.](#) | [4.5.](#) | [4.6.](#) | [4.7.](#) | [4.8.](#) | [4.9.](#) | [4.10.](#) | [4.11.](#) | [4.12.](#)

## 1. Summary

In this study, face recognition was performed using the face images in the Olivetti data set. The steps for face recognition are as follows:

- Principal components of face images were obtained by PCA.
- Adequate number of principal components determined
- According to three different classification models, accuracy score obtained.
- According to three different classification models, cross-validation accuracy score were obtained.
- Parameter optimization of the best model has been made.

[Go to Contents Menu](#) | Quick Links: [1.](#) | [3.](#) | [3.1.](#) | [3.2.](#) | [4.](#) | [4.1.](#) | [4.2.](#) | [4.3.](#) | [4.4.](#) | [4.5.](#) | [4.6.](#) | [4.7.](#) | [4.8.](#) | [4.9.](#) | [4.10.](#) | [4.11.](#) | [4.12.](#)

## 2. Face Recognition

The first study on automatic facial recognition systems was performed by Bledsoe between 1964 and 1966. This study was semi-automatic. The feature points on the face are determined manually and placed in the table called RAND. Then, a computer would perform the recognition process by classifying these points. However, a fully functional facial recognition application was performed in 1977 by Kanade. A feature-based approach was proposed in the study. After this date, two-dimensional (2D) face recognition have studied intensively. Three-dimensional (3D) face studies were started to be made after the 2000s.

3D facial recognition approaches developed in a different way than 2D facial recognition approaches. Therefore, it will be more accurate to categorize in 2D and 3D when discussing face recognition approaches.

We can classify the face recognition researches carried out with 2D approach in three categories; analytical (feature-based, local), global (appearance) and hybrid methods. While analytical approaches want to recognize by comparing the properties of the facial components, global approaches try to achieve a recognition with data derived from all the face. Hybrid approaches, together with local and global approaches, try to obtain data that expresses the face more accurately.

Face recognition performed in this kernel can assessed under global face recognition approaches.

In analytical approaches, the distance of the determined feature points and the angles between them, the shape of the facial features or the variables containing the regional features are obtained from the face image are used in face recognition. Analytical methods examine the face images in two different ways according to the pattern and geometrical properties. In these methods, the face image is represented by smaller size data, so the big data size problem that increases the computation cost in face recognition is solved.

Global-based methods are applied to face recognition by researchers because they perform facial recognition without feature extraction which is troublesome in feature based methods. Globally based methods have been used in face recognition since the 1990s, since they significantly improve facial recognition efficiency. Kirby and Sirovich (1990) first developed a method known as Eigenface, which is used in facial

representation and recognition based on Principal Component Analysis . With this method, Turk and Pentland transformed the entire face image into vectors and computed eigenfaces with a set of samples. PCA was able to obtain data representing the face at the optimum level with the data obtained from the image. The different facial and illumination levels of the same person were evaluated as the weakness point of PCA.

The face recognition performed in this kernel totally based on Turk and Pentland work.

[Go to Contents Menu](#) | Quick Links: [1](#).[3](#).[3.1](#).[3.2](#).[4](#).[4.1](#).[4.2](#).[4.3](#).[4.4](#).[4.5](#).[4.6](#).[4.7](#).[4.8](#).[4.9](#).[4.10](#).[4.11](#).[4.12](#).

### 3. Olivetti Dataset

Brief information about Olivetti Dataset:

- Face images taken between April 1992 and April 1994.
- There are ten different images of each of 40 distinct people
- There are 400 face images in the dataset
- Face images were taken at different times, varying lighting, facial expression and facial detail
- All face images have black background
- The images are gray level
- Size of each image is 64x64
- Image pixel values were scaled to [0, 1] interval
- Names of 40 people were encoded to an integer from 0 to 39

```
data=np.load("../input/olivetti_faces.npy")
target=np.load("../input/olivetti_faces_target.npy")
```

Let's verify above information

```
print("There are {} images in the dataset".format(len(data)))
print("There are {} unique targets in the dataset".format(len(np.unique(target))))
print("Size of each image is {}x{}".format(data.shape[1],data.shape[2]))
print("Pixel values were scaled to [0,1] interval. e.g:{}".format(data[0][0,:4]))

There are 400 images in the dataset
There are 40 unique targets in the dataset
Size of each image is 64x64
Pixel values were scaled to [0,1] interval. e.g:[0.30991736 0.3677686 0.41735536 0.44214877]

print("unique target number:",np.unique(target))

unique target number: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39]
```

[Go to Contents Menu](#) | Quick Links: [1](#).[3](#).[3.1](#).[3.2](#).[4](#).[4.1](#).[4.2](#).[4.3](#).[4.4](#).[4.5](#).[4.6](#).[4.7](#).[4.8](#).[4.9](#).[4.10](#).[4.11](#).[4.12](#).

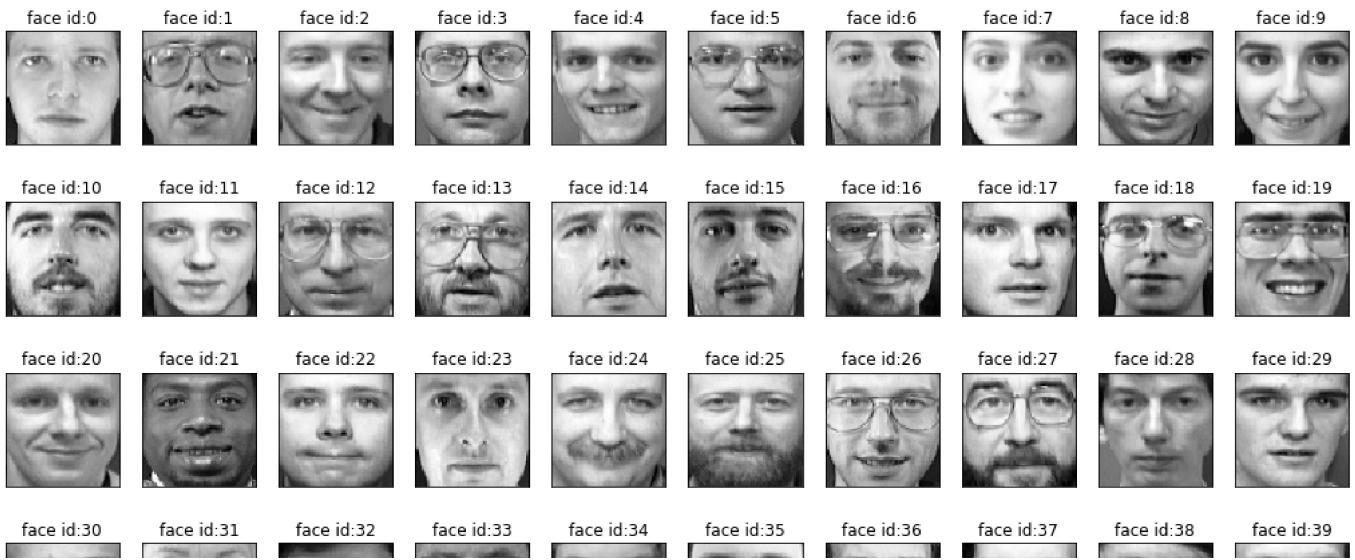
#### 3.1. Show 48 Distinct People in the Olivetti Dataset

```
def show_40_distinct_people(images, unique_ids):
    #Creating 4X10 subplots in 18x9 figure size
    fig, axarr=plt.subplots(nrows=4, ncols=10, figsize=(18, 9))
    #For easy iteration flattened 4X10 subplots matrix to 40 array
    axarr=axarr.flatten()

    #Iterating over user ids
    for unique_id in unique_ids:
        image_index=unique_id*10
        axarr[unique_id].imshow(images[image_index], cmap='gray')
        axarr[unique_id].set_xticks([])
        axarr[unique_id].set_yticks([])
        axarr[unique_id].set_title("face id:{}".format(unique_id))
    plt.suptitle("There are 40 distinct people in the dataset")

show_40_distinct_people(data, np.unique(target))
```

There are 40 distinct people in the dataset



As seen in the photo gallery above, the data set has 40 different person-owned, facial images.



[Go to Contents Menu](#) | Quick Links: [1](#).[3](#).[3.1](#).[3.2](#).[4](#).[4.1](#).[4.2](#).[4.3](#).[4.4](#).[4.5](#).[4.6](#).[4.7](#).[4.8](#).[4.9](#).[4.10](#).[4.11](#).[4.12](#).

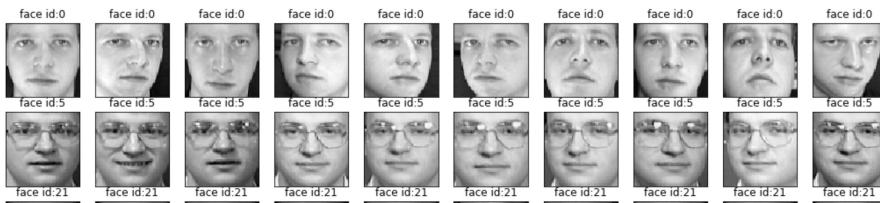
### 3.2. Show 10 Face Images of Selected Target

```
def show_10_faces_of_n_subject(images, subject_ids):
    cols=10# each subject has 10 distinct face images
    rows=(len(subject_ids)*10)/cols #
    rows=int(rows)

    fig, axarr=plt.subplots(nrows=rows, ncols=cols, figsize=(18,9))
    #axarr=axarr.flatten()

    for i, subject_id in enumerate(subject_ids):
        for j in range(cols):
            image_index=subject_id*10 + j
            axarr[i,j].imshow(images[image_index], cmap="gray")
            axarr[i,j].set_xticks([])
            axarr[i,j].set_yticks([])
            axarr[i,j].set_title("face id:{}".format(subject_id))

    #You can playaround subject_ids to see other people faces
    show_10_faces_of_n_subject(images=data, subject_ids=[0,5, 21, 24, 36])
```



Each face of a subject has different characteristic in context of varying lighting, facial express and facial detail(glasses, beard)



[Go to Contents Menu](#) | Quick Links: [1.](#) [3.](#) [3.1.](#) [3.2.](#) [4.](#) [4.1.](#) [4.2.](#) [4.3.](#) [4.4.](#) [4.5.](#) [4.6.](#) [4.7.](#) [4.8.](#) [4.9.](#) [4.10.](#) [4.11.](#) [4.12.](#)

#### 4. Machine Learning Model fo Face Recognition

Machine learning models can work on vectors. Since the image data is in the matrix form, it must be converted to a vector.



```
#We reshape images for machine learning model
X=data.reshape((data.shape[0],data.shape[1]*data.shape[2]))
print("X shape:",X.shape)

X shape: (400, 4096)
```

[Go to Contents Menu](#) | Quick Links: [1.](#) [3.](#) [3.1.](#) [3.2.](#) [4.](#) [4.1.](#) [4.2.](#) [4.3.](#) [4.4.](#) [4.5.](#) [4.6.](#) [4.7.](#) [4.8.](#) [4.9.](#) [4.10.](#) [4.11.](#) [4.12.](#)

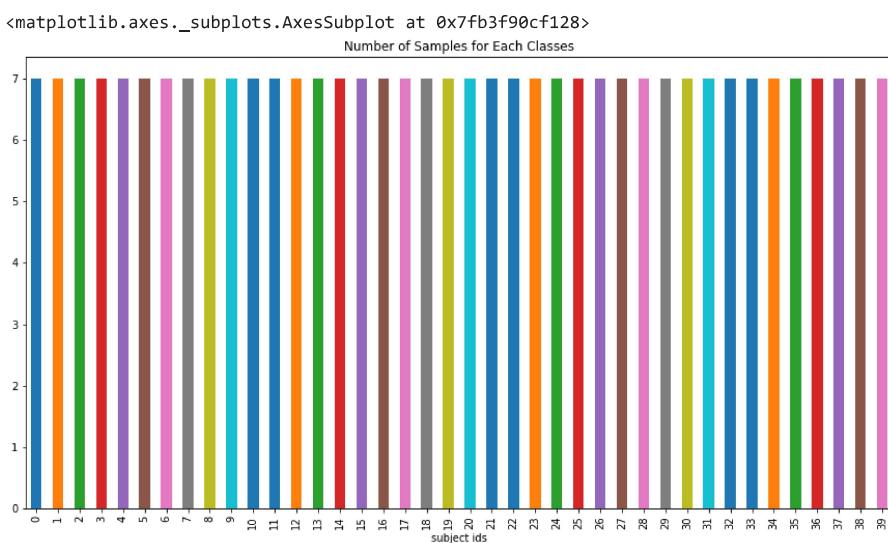
##### 4.1. Split data and target into Random train and test Subsets

The data set contains 10 face images for each subject. Of the face images, 70 percent will be used for training, 30 percent for testing. Uses stratify feature to have equal number of training and test images for each subject. Thus, there will be 7 training images and 3 test images for each subject. You can play with training and test rates.

```
X_train, X_test, y_train, y_test=train_test_split(X, target, test_size=0.3, stratify=target, random_state=0)
print("X_train shape:",X_train.shape)
print("y_train shape:{}".format(y_train.shape))

X_train shape: (280, 4096)
y_train shape:(280,)
```

```
y_frame=pd.DataFrame()
y_frame['subject ids']=y_train
y_frame.groupby(['subject ids']).size().plot.bar(figsize=(15,8),title="Number of Samples for Each Classes")
```



[Go to Contents Menu](#) | Quick Links: [1.](#) [3.](#) [3.1.](#) [3.2.](#) [4.](#) [4.1.](#) [4.2.](#) [4.3.](#) [4.4.](#) [4.5.](#) [4.6.](#) [4.7.](#) [4.8.](#) [4.9.](#) [4.10.](#) [4.11.](#) [4.12.](#)

##### 4.2.Principle Component Analysis

Machine learning methods are divided into two: supervised learning and unsupervised learning. In supervised learning, the data set is divided into two main parts: 'data' and 'output'. The data holds the values of the sample in the data set, while the 'output' holds the class (for classification) or the target value (for regression). In unsupervised learning, the data set consists of only the data section.

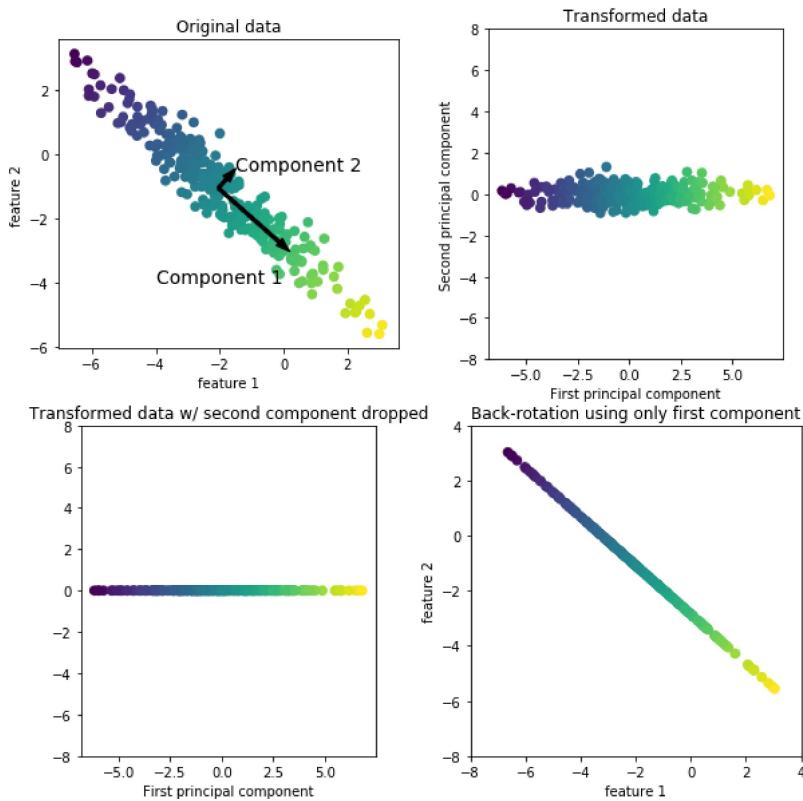
Non-supervised learning is generally divided into two: data transformation and clustering. In this study, the transformation of the data will be carried out using unsupervised learning. Unsupervised transformation methods allow for easier interpretation of data by computers and people.

The most common unsupervised transformation applications is to reduce data size. In the size reduction process, the dimension of the data is reduced.

Principle Component Analysis (PCA) is a method that allows data to be represented in a lesser size. According to this method, the data is transformed to new components and the size of the data is reduced by selecting the most important components.

```
import mglearn
```

```
mglearn.plots.plot_pca_illustration()
```



The above illustration shows a simple example on a synthetic two-dimensional data set. The first drawing shows the original data points colored to distinguish points. The algorithm first proceeds by finding the direction of the maximum variance labeled "Component 1". This refers to the direction in which most of the data is associated, or in other words, the properties that are most related to each other.

Then, when the algorithm is orthogonal (at right angle), it finds the direction that contains the most information in the first direction. There are only one possible orientation in two dimensions at a right angle, but there will be many orthogonal directions (infinite) in high dimensional spaces.

[Go to Contents Menu](#) | Quick Links: [1](#).[3](#).[3.1](#).[3.2](#).[4](#).[4.1](#).[4.2](#).[4.3](#).[4.4](#).[4.5](#).[4.6](#).[4.7](#).[4.8](#).[4.9](#).[4.10](#).[4.11](#).[4.12](#).

#### 4.3. PCA Projection of Defined Number of Target

```
from sklearn.decomposition import PCA
pca=PCA(n_components=2)
pca.fit(X)
X_pca=pca.transform(X)
```

```

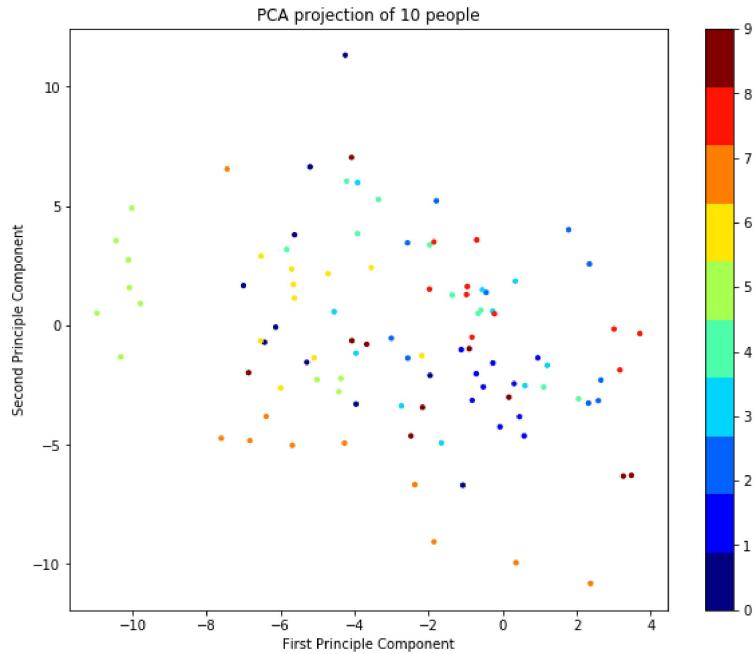
number_of_people=10
index_range=number_of_people*10
fig=plt.figure(figsize=(10,8))
ax=fig.add_subplot(1,1,1)
scatter=ax.scatter(X_pca[:index_range,0],
                   X_pca[:index_range,1],
                   c=target[:index_range],
                   s=10,
                   cmap=plt.get_cmap('jet', number_of_people)
                  )

ax.set_xlabel("First Principle Component")
ax.set_ylabel("Second Principle Component")
ax.set_title("PCA projection of {} people".format(number_of_people))

fig.colorbar(scatter)

```

<matplotlib.colorbar.Colorbar at 0x7fb3f9455438>



[Go to Contents Menu](#) | Quick Links: [1.](#) | [3.](#) | [3.1.](#) | [3.2.](#) | [4.](#) | [4.1.](#) | [4.2.](#) | [4.3.](#) | [4.4.](#) | [4.5.](#) | [4.6.](#) | [4.7.](#) | [4.8.](#) | [4.9.](#) | [4.10.](#) | [4.11.](#) | [4.12.](#)

#### 4.4. Finding Optimum Number of Principle Component

```

pca=PCA()
pca.fit(X)

plt.figure(1, figsize=(12,8))

plt.plot(pca.explained_variance_, linewidth=2)

plt.xlabel('Components')
plt.ylabel('Explained Variaces')
plt.show()

```



In the figure above, it can be seen that 90 and more PCA components represent the same data. Now let's make the classification process using 90 PCA components.

```
n_components=90
pca=PCA(n_components=n_components, whiten=True)
pca.fit(X_train)

PCA(copy=True, iterated_power='auto', n_components=90, random_state=None,
     svd_solver='auto', tol=0.0, whiten=True)
```

[Go to Contents Menu](#) | Quick Links: [1.](#) | [3.](#) | [3.1.](#) | [3.2.](#) | [4.](#) | [4.1.](#) | [4.2.](#) | [4.3.](#) | [4.4.](#) | [4.5.](#) | [4.6.](#) | [4.7.](#) | [4.8.](#) | [4.9.](#) | [4.10.](#) | [4.11.](#) | [4.12.](#)

#### 4.5. Show Average Face

```
fig,ax=plt.subplots(1,1,figsize=(8,8))
ax.imshow(pca.mean_.reshape((64,64)), cmap="gray")
ax.set_xticks([])
ax.set_yticks([])
ax.set_title('Average Face')

Text(0.5,1,'Average Face')
Average Face
```



[Go to Contents Menu](#) | Quick Links: [1.](#) | [3.](#) | [3.1.](#) | [3.2.](#) | [4.](#) | [4.1.](#) | [4.2.](#) | [4.3.](#) | [4.4.](#) | [4.5.](#) | [4.6.](#) | [4.7.](#) | [4.8.](#) | [4.9.](#) | [4.10.](#) | [4.11.](#) | [4.12.](#)

#### 4.6. Show Eigen Faces

```

number_of_eigenfaces=len(pca.components_)
eigen_faces=pca.components_.reshape((number_of_eigenfaces, data.shape[1], data.shape[2]))

cols=10
rows=int(number_of_eigenfaces/cols)
fig, axarr=plt.subplots(nrows=rows, ncols=cols, figsize=(15,15))
axarr=axarr.flatten()
for i in range(number_of_eigenfaces):
    axarr[i].imshow(eigen_faces[i], cmap="gray")
    axarr[i].set_xticks([])
    axarr[i].set_yticks([])
    axarr[i].set_title("eigen id:{}".format(i))
plt.suptitle("All Eigen Faces".format(10**"=", 10**"="))

Text(0.5,0.98, 'All Eigen Faces')

```

All Eigen Faces



[Go to Contents Menu](#) | Quick Links: [1.](#) | [3.](#) | [3.1.](#) | [3.2.](#) | [4.](#) | [4.1.](#) | [4.2.](#) | [4.3.](#) | [4.4.](#) | [4.5.](#) | [4.6.](#) | [4.7.](#) | [4.8.](#) | [4.9.](#) | [4.10.](#) | [4.11.](#) | [4.12.](#)

#### 4.7. Classification Results

```

X_train_pca=pca.transform(X_train)
X_test_pca=pca.transform(X_test)

```

```

clf = SVC()
clf.fit(X_train_pca, y_train)
y_pred = clf.predict(X_test_pca)
print("accuracy score:{:.2f}".format(metrics.accuracy_score(y_test, y_pred)))

accuracy score:0.91

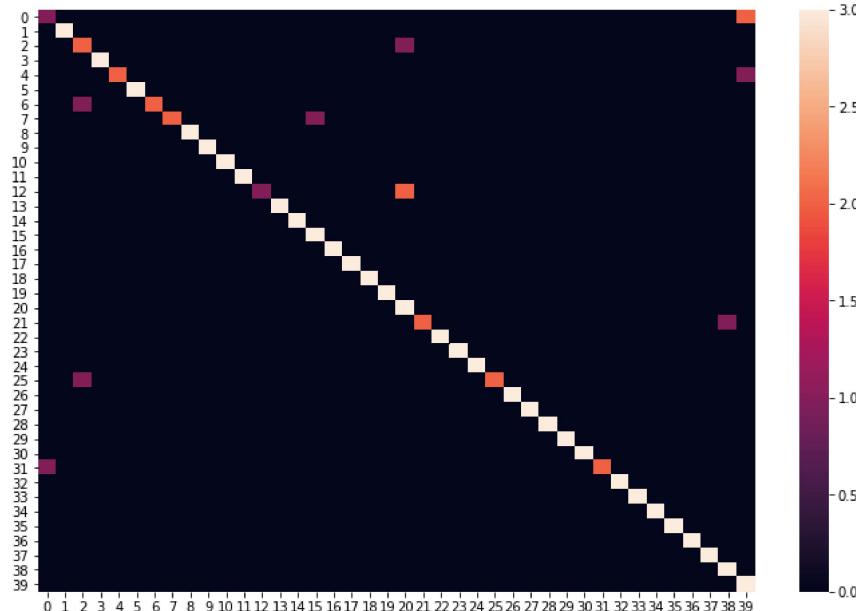
```

```

import seaborn as sns
plt.figure(1, figsize=(12,8))
sns.heatmap(metrics.confusion_matrix(y_test, y_pred))

```

<matplotlib.axes.\_subplots.AxesSubplot at 0x7fb3e805f978>



```
print(metrics.classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	0.50	0.33	0.40	3
1	1.00	1.00	1.00	3
2	0.50	0.67	0.57	3
3	1.00	1.00	1.00	3
4	1.00	0.67	0.80	3
5	1.00	1.00	1.00	3
6	1.00	0.67	0.80	3
7	1.00	0.67	0.80	3
8	1.00	1.00	1.00	3
9	1.00	1.00	1.00	3
10	1.00	1.00	1.00	3
11	1.00	1.00	1.00	3
12	1.00	0.33	0.50	3
13	1.00	1.00	1.00	3
14	1.00	1.00	1.00	3
15	0.75	1.00	0.86	3
16	1.00	1.00	1.00	3
17	1.00	1.00	1.00	3
18	1.00	1.00	1.00	3
19	1.00	1.00	1.00	3
20	0.50	1.00	0.67	3
21	1.00	0.67	0.80	3
22	1.00	1.00	1.00	3
23	1.00	1.00	1.00	3
24	1.00	1.00	1.00	3
25	1.00	0.67	0.80	3
26	1.00	1.00	1.00	3
27	1.00	1.00	1.00	3
28	1.00	1.00	1.00	3
29	1.00	1.00	1.00	3
30	1.00	1.00	1.00	3
31	1.00	0.67	0.80	3
32	1.00	1.00	1.00	3
33	1.00	1.00	1.00	3
34	1.00	1.00	1.00	3
35	1.00	1.00	1.00	3
36	1.00	1.00	1.00	3
37	1.00	1.00	1.00	3

38	0.75	1.00	0.86	3
39	0.50	1.00	0.67	3
micro avg	0.91	0.91	0.91	120
macro avg	0.94	0.91	0.91	120

weighted avg    0.94    0.91    0.91    120

[Go to Contents Menu](#) | Quick Links: [1.](#) | [3.](#) | [3.1.](#) | [3.2.](#) | [4.](#) | [4.1.](#) | [4.2.](#) | [4.3.](#) | [4.4.](#) | [4.5.](#) | [4.6.](#) | [4.7.](#) | [4.8.](#) | [4.9.](#) | [4.10.](#) | [4.11.](#) | [4.12.](#)

#### 4.8. More Results

We can get accuracy results of state of the art machine learning model.

```
models=[]
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(("LR", LogisticRegression()))
models.append(("NB", GaussianNB()))
models.append(("KNN", KNeighborsClassifier(n_neighbors=5)))
models.append(("DT", DecisionTreeClassifier()))
models.append(("SVM", SVC()))

for name, model in models:
    clf=model
    clf.fit(X_train_pca, y_train)

    y_pred=clf.predict(X_test_pca)
    print(10*="#"+"{} Result".format(name).upper(),10*="#")
    print("Accuracy score:{:0.2f}".format(metrics.accuracy_score(y_test, y_pred)))
    print()

    ===== LDA RESULT =====
    Accuracy score:0.93

    ===== LR RESULT =====
    Accuracy score:0.93

    ===== NB RESULT =====
    Accuracy score:0.87

    ===== KNN RESULT =====
    Accuracy score:0.70

    ===== DT RESULT =====
    Accuracy score:0.61

    ===== SVM RESULT =====
    Accuracy score:0.91
```

According to the above results, Linear Discriminant Analysis and Logistic Regression seems to have the best performances.

[Go to Contents Menu](#) | Quick Links: [1.](#) | [3.](#) | [3.1.](#) | [3.2.](#) | [4.](#) | [4.1.](#) | [4.2.](#) | [4.3.](#) | [4.4.](#) | [4.5.](#) | [4.6.](#) | [4.7.](#) | [4.8.](#) | [4.9.](#) | [4.10.](#) | [4.11.](#) | [4.12.](#)

#### 4.9. Validated Results

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
pca=PCA(n_components=n_components, whiten=True)
pca.fit(X)
X_pca=pca.transform(X)
for name, model in models:
    kfold=KFold(n_splits=5, shuffle=True, random_state=0)

    cv_scores=cross_val_score(model, X_pca, target, cv=kfold)
    print("{} mean cross validations score:{:.2f}".format(name, cv_scores.mean()))

LDA mean cross validations score:0.98
LR mean cross validations score:0.95
NB mean cross validations score:0.77
KNN mean cross validations score:0.69
DT mean cross validations score:0.48
SVM mean cross validations score:0.88
```

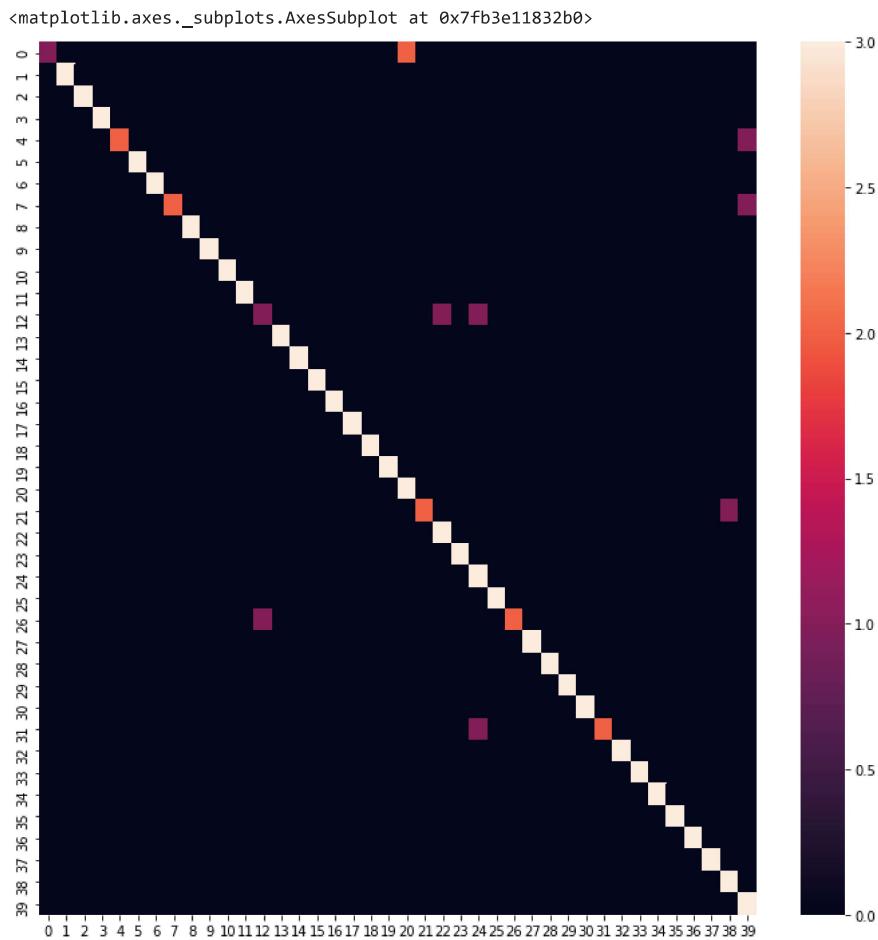
According to the cross validation scores Linear Discriminant Analysis and Logistic Regression still have best performance

```
lr=LinearDiscriminantAnalysis()
lr.fit(X_train_pca, y_train)
y_pred=lr.predict(X_test_pca)
print("Accuracy score:{:.2f}".format(metrics.accuracy_score(y_test, y_pred)))
```

Accuracy score:0.93

```
cm=metrics.confusion_matrix(y_test, y_pred)
```

```
plt.subplots(1, figsize=(12,12))
sns.heatmap(cm)
```



```
print("Classification Results:\n{}\n".format(metrics.classification_report(y_test, y_pred)))
```

Classification Results:		precision	recall	f1-score	support
0	1.00	0.33	0.50	3	
1	1.00	1.00	1.00	3	
2	1.00	1.00	1.00	3	
3	1.00	1.00	1.00	3	
4	1.00	0.67	0.80	3	
5	1.00	1.00	1.00	3	
6	1.00	1.00	1.00	3	
7	1.00	0.67	0.80	3	
8	1.00	1.00	1.00	3	
9	1.00	1.00	1.00	3	
10	1.00	1.00	1.00	3	
11	1.00	1.00	1.00	3	
12	0.50	0.33	0.40	3	
13	1.00	1.00	1.00	3	
14	1.00	1.00	1.00	3	
15	1.00	1.00	1.00	3	
16	1.00	1.00	1.00	3	
17	1.00	1.00	1.00	3	
18	1.00	1.00	1.00	3	

19	1.00	1.00	1.00	3
20	0.60	1.00	0.75	3
21	1.00	0.67	0.80	3
22	0.75	1.00	0.86	3
23	1.00	1.00	1.00	3
24	0.60	1.00	0.75	3
25	1.00	1.00	1.00	3
26	1.00	0.67	0.80	3
27	1.00	1.00	1.00	3
28	1.00	1.00	1.00	3
29	1.00	1.00	1.00	3
30	1.00	1.00	1.00	3
31	1.00	0.67	0.80	3
32	1.00	1.00	1.00	3
33	1.00	1.00	1.00	3
34	1.00	1.00	1.00	3
35	1.00	1.00	1.00	3
36	1.00	1.00	1.00	3
37	1.00	1.00	1.00	3
38	0.75	1.00	0.86	3
39	0.60	1.00	0.75	3
micro avg		0.93	0.93	120
macro avg		0.94	0.93	120
weighted avg		0.95	0.93	120

[Go to Contents Menu](#) | Quick Links: [1.](#) | [3.](#) | [3.1.](#) | [3.2.](#) | [4.](#) | [4.1.](#) | [4.2.](#) | [4.3.](#) | [4.4.](#) | [4.5.](#) | [4.6.](#) | [4.7.](#) | [4.8.](#) | [4.9.](#) | [4.10.](#) | [4.11.](#) | [4.12.](#)

#### 4.10. More Validated Results: Leave One Out cross-validation

The Olivetti dataset contains 10 face images for each subject. This is a small number for training and testing machine learning models. There is a recommended cross-validation method for better assessment of machine learning models where there are few examples of classes: Leave One Out cross validation. In the LOO approach, only one of the samples of a class is used for testing. Others are used for training. This procedure is repeated until each sample is used for testing.

```
from sklearn.model_selection import LeaveOneOut
loo_cv=LeaveOneOut()
clf=LogisticRegression()
cv_scores=cross_val_score(clf,
                         X_pca,
                         target,
                         cv=loo_cv)
print("{} Leave One Out cross-validation mean accuracy score:{:.2f}".format(clf.__class__.__name__,
                                                                           cv_scores.mean()))
```

LogisticRegression Leave One Out cross-validation mean accuracy score:0.96

```
from sklearn.model_selection import LeaveOneOut
loo_cv=LeaveOneOut()
clf=LinearDiscriminantAnalysis()
cv_scores=cross_val_score(clf,
                         X_pca,
                         target,
                         cv=loo_cv)
print("{} Leave One Out cross-validation mean accuracy score:{:.2f}".format(clf.__class__.__name__,
                                                                           cv_scores.mean()))
```

LinearDiscriminantAnalysis Leave One Out cross-validation mean accuracy score:0.98

[Go to Contents Menu](#) | Quick Links: [1.](#) | [3.](#) | [3.1.](#) | [3.2.](#) | [4.](#) | [4.1.](#) | [4.2.](#) | [4.3.](#) | [4.4.](#) | [4.5.](#) | [4.6.](#) | [4.7.](#) | [4.8.](#) | [4.9.](#) | [4.10.](#) | [4.11.](#) | [4.12.](#)

#### 4.11. Hyperparameter Tuning: GridSearchCV

We can do GridSearchCV to improve model generalization performance. To that we will tune the hyperparameters of Logistic Regression classifier.

```
from sklearn.model_selection import GridSearchCV
```

```

from sklearn.model_selection import LeaveOneOut

#This process takes long time. You can use parameter:{'C': 1.0, 'penalty': 'l2'}
#grid search cross validation score:0.93
"""

params={'penalty':['l1', 'l2'],
         'C':np.logspace(0, 4, 10)
        }
clf=LogisticRegression()
#kfold=KFold(n_splits=3, shuffle=True, random_state=0)
loo_cv=LeaveOneOut()
gridSearchCV=GridSearchCV(clf, params, cv=loo_cv)
gridSearchCV.fit(X_train_pca, y_train)
print("Grid search fitted..")
print(gridSearchCV.best_params_)
print(gridSearchCV.best_score_)
print("grid search cross validation score:{:.2f}".format(gridSearchCV.score(X_test_pca, y_test)))
"""

```
\`nparams={'\`penalty\`:[\`l1\`, \`l2\`]}, \`C\`:=np.logspace(0, 4, 10)\n`nclf=LogisticRegression()\`nkfold=KFold(n_splits=3, shuffle=True,\nrandom_state=0)\`nloo_cv=LeaveOneOut()\`ngridSearchCV=GridSearchCV(clf, params, cv=loo_cv)\`ngridSearchCV.fit(X_train_pca,\ny_train)\`nprint("Grid search fitted..")\`nprint(gridSearchCV.best_params_)\`nprint(gridSearchCV.best_score_)\`nprint("grid search cross\nvalidation score:{:.2f}\``.format(gridSearchCV.score(X_test_pca, y_test)))\`n`n
```
lr=LogisticRegression(C=1.0, penalty="l2")
lr.fit(X_train_pca, y_train)
print("lr score:{:.2f}`.format(lr.score(X_test_pca, y_test)))`n
```
lr score:0.93
```

```

[Go to Contents Menu](#) | Quick Links: [1.](#) | [3.](#) | [3.1.](#) | [3.2.](#) | [4.](#) | [4.1.](#) | [4.2.](#) | [4.3.](#) | [4.4.](#) | [4.5.](#) | [4.6.](#) | [4.7.](#) | [4.8.](#) | [4.9.](#) | [4.10.](#) | [4.11.](#) | [4.12.](#)

#### 4.12. Precision-Recall-ROC Curves

Precision-recall curves are for binary classification. In Olivetti dataset, there are 40 distinct classes. Dont worry, sklearn allows us to illustrate the precision-recall in multi-label settings

```

from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier

Target=label_binarize(target, classes=range(40))
print(Target.shape)
print(Target[0])

n_classes=Target.shape[1]

(400, 40)
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0]

X_train_multiclass, X_test_multiclass, y_train_multiclass, y_test_multiclass=train_test_split(X,
                                            Target,
                                            test_size=0.3,
                                            stratify=Target,
                                            random_state=0)

pca=PCA(n_components=n_components, whiten=True)
pca.fit(X_train_multiclass)

X_train_multiclass_pca=pca.transform(X_train_multiclass)
X_test_multiclass_pca=pca.transform(X_test_multiclass)

oneRestClassifier=OneVsRestClassifier(lr)

oneRestClassifier.fit(X_train_multiclass_pca, y_train_multiclass)
y_score=oneRestClassifier.decision_function(X_test_multiclass_pca)
```

```

```
# For each class
precision = dict()
recall = dict()
average_precision = dict()
for i in range(n_classes):
    precision[i], recall[i], _ = metrics.precision_recall_curve(y_test_multiclass[:, i],
  y_score[:, i])
    average_precision[i] = metrics.average_precision_score(y_test_multiclass[:, i], y_score[:, i])

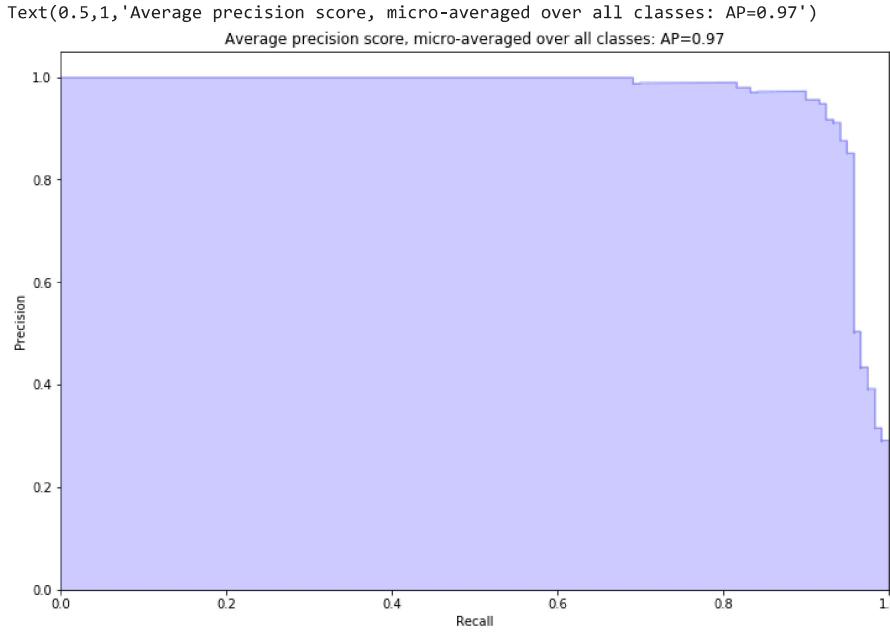
# A "micro-average": quantifying score on all classes jointly
precision["micro"], recall["micro"], _ = metrics.precision_recall_curve(y_test_multiclass.ravel(),
    y_score.ravel())
average_precision["micro"] = metrics.average_precision_score(y_test_multiclass, y_score,
    average="micro")
print('Average precision score, micro-averaged over all classes: {0:0.2f}\n'.
    .format(average_precision["micro"]))

Average precision score, micro-averaged over all classes: 0.97
```

```
from sklearn.utils.fixes import signature

step_kwargs = ({'step': 'post'}
              if 'step' in signature(plt.fill_between).parameters
              else {})
plt.figure(1, figsize=(12,8))
plt.step(recall['micro'], precision['micro'], color='b', alpha=0.2,
         where='post')
plt.fill_between(recall["micro"], precision["micro"], alpha=0.2, color='b',
                 **step_kwargs)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title(
    'Average precision score, micro-averaged over all classes: AP={0:0.2f}\n'.
    .format(average_precision["micro"]))
```



[Go to Contents Menu](#) | Quick Links: [1.](#) [3.](#) [3.1.](#) [3.2.](#) [4.](#) [4.1.](#) [4.2.](#) [4.3.](#) [4.4.](#) [4.5.](#) [4.6.](#) [4.7.](#) [4.8.](#) [4.9.](#) [4.10.](#) [4.11.](#) [4.12.](#)

## 5. Linear Discriminant Analysis ile Boyut Azaltma

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

```

lda = LinearDiscriminantAnalysis(n_components=n_components)
X_train_lda = lda.fit(X_train, y_train).transform(X_train)
X_test_lda=lda.transform(X_test)

lr=LogisticRegression(C=1.0, penalty="l2")
lr.fit(X_train_lda,y_train)
y_pred=lr.predict(X_test_lda)

print("Accuracy score:{:.2f}".format(metrics.accuracy_score(y_test, y_pred)))
print("Classification Results:\n{}".format(metrics.classification_report(y_test, y_pred)))

```

Accuracy score:0.94  
Classification Results:

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 0.50      | 0.33   | 0.40     | 3       |
| 1            | 1.00      | 1.00   | 1.00     | 3       |
| 2            | 0.60      | 1.00   | 0.75     | 3       |
| 3            | 1.00      | 1.00   | 1.00     | 3       |
| 4            | 1.00      | 1.00   | 1.00     | 3       |
| 5            | 1.00      | 1.00   | 1.00     | 3       |
| 6            | 1.00      | 1.00   | 1.00     | 3       |
| 7            | 1.00      | 0.67   | 0.80     | 3       |
| 8            | 0.75      | 1.00   | 0.86     | 3       |
| 9            | 1.00      | 1.00   | 1.00     | 3       |
| 10           | 1.00      | 1.00   | 1.00     | 3       |
| 11           | 1.00      | 1.00   | 1.00     | 3       |
| 12           | 1.00      | 0.67   | 0.80     | 3       |
| 13           | 1.00      | 1.00   | 1.00     | 3       |
| 14           | 1.00      | 1.00   | 1.00     | 3       |
| 15           | 1.00      | 1.00   | 1.00     | 3       |
| 16           | 1.00      | 1.00   | 1.00     | 3       |
| 17           | 1.00      | 1.00   | 1.00     | 3       |
| 18           | 1.00      | 1.00   | 1.00     | 3       |
| 19           | 1.00      | 1.00   | 1.00     | 3       |
| 20           | 1.00      | 1.00   | 1.00     | 3       |
| 21           | 1.00      | 0.67   | 0.80     | 3       |
| 22           | 1.00      | 1.00   | 1.00     | 3       |
| 23           | 0.75      | 1.00   | 0.86     | 3       |
| 24           | 1.00      | 1.00   | 1.00     | 3       |
| 25           | 1.00      | 0.67   | 0.80     | 3       |
| 26           | 1.00      | 0.67   | 0.80     | 3       |
| 27           | 1.00      | 1.00   | 1.00     | 3       |
| 28           | 1.00      | 1.00   | 1.00     | 3       |
| 29           | 1.00      | 1.00   | 1.00     | 3       |
| 30           | 1.00      | 1.00   | 1.00     | 3       |
| 31           | 1.00      | 1.00   | 1.00     | 3       |
| 32           | 1.00      | 1.00   | 1.00     | 3       |
| 33           | 1.00      | 1.00   | 1.00     | 3       |
| 34           | 1.00      | 1.00   | 1.00     | 3       |
| 35           | 1.00      | 1.00   | 1.00     | 3       |
| 36           | 1.00      | 1.00   | 1.00     | 3       |
| 37           | 1.00      | 1.00   | 1.00     | 3       |
| 38           | 0.75      | 1.00   | 0.86     | 3       |
| 39           | 0.75      | 1.00   | 0.86     | 3       |
| micro avg    | 0.94      | 0.94   | 0.94     | 120     |
| macro avg    | 0.95      | 0.94   | 0.94     | 120     |
| weighted avg | 0.95      | 0.94   | 0.94     | 120     |

[Go to Contents Menu](#) | Quick Links: [1](#).[3](#).[3.1](#).[3.2](#).[4](#).[4.1](#).[4.2](#).[4.3](#).[4.4](#).[4.5](#).[4.6](#).[4.7](#).[4.8](#).[4.9](#).[4.10](#).[4.11](#).[4.12](#).

## 6. Machine Learning Automated Workflow: Pipeline

Application of machine learning on data sets has a standard workflow. Sklearn offers the Pipeline object to automate this workflow. Pipeline allows standard work flows for performing machine learning operations such as scaling, feature extraction and modeling. The Pipeline guarantees the same operation in the entire data set, ensuring that the training and test data are consistent.

```

from sklearn.pipeline import Pipeline

work_flows_std = list()
work_flows_std.append(('lda', LinearDiscriminantAnalysis(n_components=n_components)))
work_flows_std.append(('logReg', LogisticRegression(C=1.0, penalty="l2")))
model_std = Pipeline(work_flows_std)
model_std.fit(X_train, y_train)
y_pred=model_std.predict(X_test)

```

```
print("Accuracy score:{:.2f}" .format(metrics.accuracy_score(y_test, y_pred)))
print("Classification Results:\n{}".format(metrics.classification_report(y_test, y_pred)))

Accuracy score:0.94
Classification Results:
      precision    recall  f1-score   support

       0         0.50     0.33     0.40       3
       1         1.00     1.00     1.00       3
       2         0.60     1.00     0.75       3
       3         1.00     1.00     1.00       3
       4         1.00     1.00     1.00       3
       5         1.00     1.00     1.00       3
       6         1.00     1.00     1.00       3
       7         1.00     0.67     0.80       3
       8         0.75     1.00     0.86       3
       9         1.00     1.00     1.00       3
      10        1.00     1.00     1.00       3
      11        1.00     1.00     1.00       3
      12        1.00     0.67     0.80       3
      13        1.00     1.00     1.00       3
      14        1.00     1.00     1.00       3
      15        1.00     1.00     1.00       3
      16        1.00     1.00     1.00       3
      17        1.00     1.00     1.00       3
      18        1.00     1.00     1.00       3
      19        1.00     1.00     1.00       3
      20        1.00     1.00     1.00       3
      21        1.00     0.67     0.80       3
      22        1.00     1.00     1.00       3
      23        0.75     1.00     0.86       3
      24        1.00     1.00     1.00       3
      25        1.00     0.67     0.80       3
      26        1.00     0.67     0.80       3
      27        1.00     1.00     1.00       3
      28        1.00     1.00     1.00       3
      29        1.00     1.00     1.00       3
      30        1.00     1.00     1.00       3
      31        1.00     1.00     1.00       3
      32        1.00     1.00     1.00       3
      33        1.00     1.00     1.00       3
      34        1.00     1.00     1.00       3
      35        1.00     1.00     1.00       3
      36        1.00     1.00     1.00       3
      37        1.00     1.00     1.00       3
```