Lecture 23 Part 1 - PCA Application & Performance Measures for Classification Tasks

In [1]:
```
import pandas as pd
from scipy import stats
import numpy as np
import numpy.random as npr
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('bmh')
plt.rcParams['axes.grid'] = False

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

In [2]:
```
df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/wine/

df_wine.columns = ['Class label', 'Alcohol',
                   'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium',
                   'Total phenols', 'Flavanoids',
                   'Nonflavanoid phenols',
                   'Proanthocyanins',
                   'Color intensity', 'Hue',
                   'OD280/OD315 of diluted wines',
                   'Proline']

df_wine
```

Out[2]:

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanth
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	
...	...	...	...	...	...	...	...	...	...	
173	3	13.71	5.65	2.45	20.5	95	1.68	0.61	0.52	
174	3	13.40	3.91	2.48	23.0	102	1.80	0.75	0.43	
175	3	13.27	4.28	2.26	20.0	120	1.59	0.69	0.43	
176	3	13.17	2.59	2.37	20.0	120	1.65	0.68	0.53	
177	3	14.13	4.10	2.74	24.5	96	2.05	0.76	0.56	

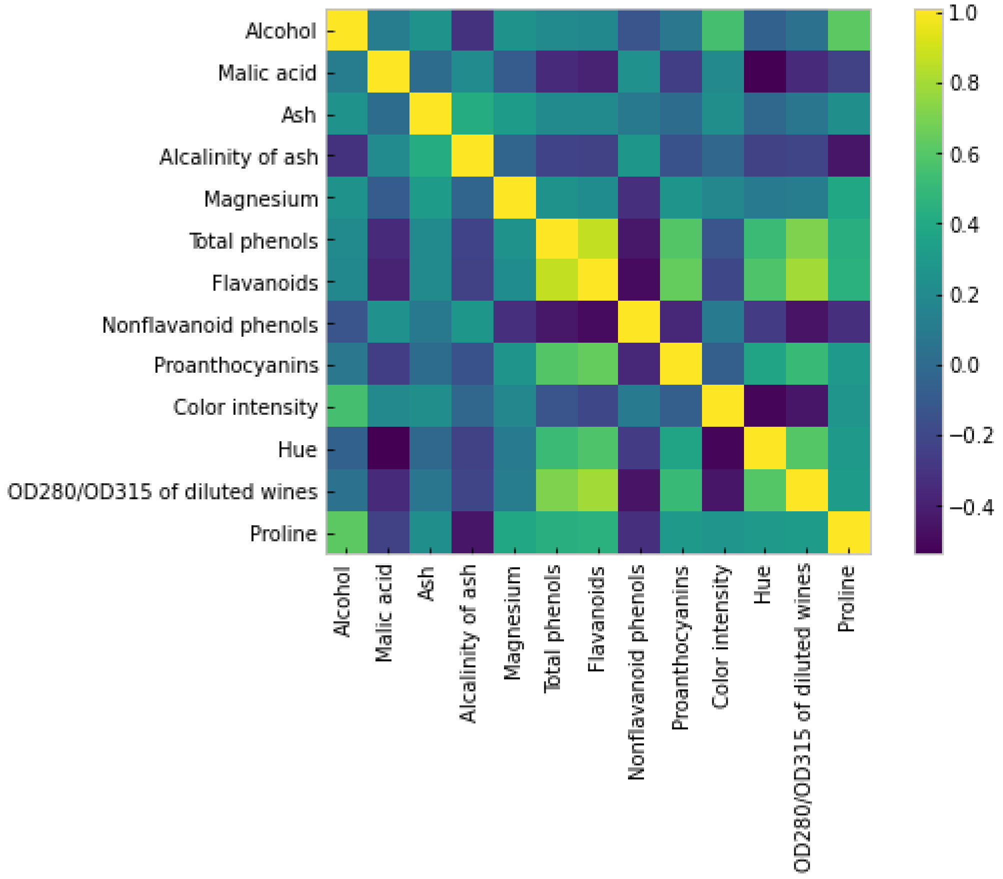
178 rows × 14 columns

In [3]:
```python
# Target Labels
t = df_wine['Class label'].values

# Feature Matrix
X = df_wine.drop(['Class label'], axis=1).values
print(X.shape)

# Stratified partition of the data into training/test sets
X_train, X_test, t_train, t_test = train_test_split(X, t,
                                                    test_size=0.3,
                                                    stratify=t)
# Scaling data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```
```
(178, 13)
```

Coming back to the wine dataset:

In [4]:
```python
X_train.shape
```
Out[4]:
```
(124, 13)
```

In [5]:
```python
cov_train = np.cov(X_train.T)
# np.cov expects the input to be D-by-N matrix

plt.figure(figsize=(8,5))
plt.imshow(cov_train)
plt.colorbar()
plt.xticks(range(13),df_wine.columns[1:],rotation=90)
plt.yticks(range(13),df_wine.columns[1:]);
```



Building a function to implement PCA from scratch:

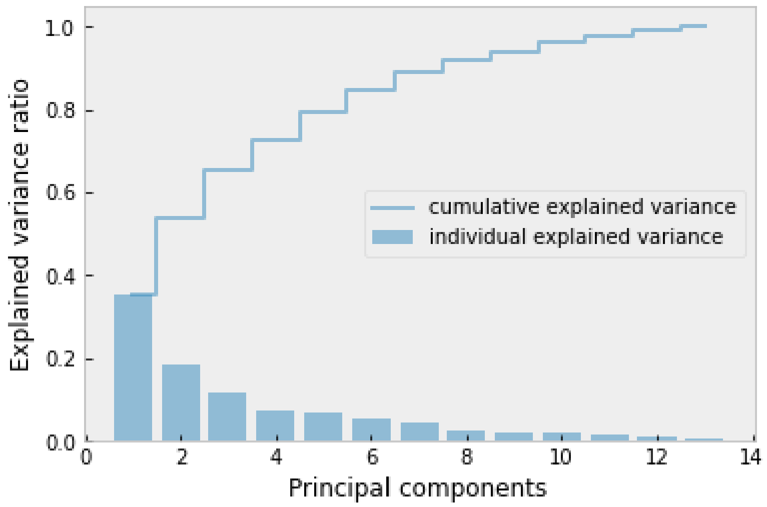
In [6]:
```
def myPCA(X, m, display=1):
    '''This function implements PCA. The data matrix X is DxN matrix,
    where D is the dimension and N the number of points'''

    D, N = X.shape

    # Demean the Data
    data = X - X.mean(axis=1).reshape(-1, 1)

    # Covariance of the input  X
    cov_mat = np.cov(data)

    # Find eigenvectors and eigenvalues
    eigen_vals, eigen_vecs = np.linalg.eigh(cov_mat)
    # by default np.linalg.eigh will store eigenvectors/eigenvalues in increasing orde

    # Sort eigenvectors by magnitude of eigenvalues
    L = eigen_vals[::-1]
    U = eigen_vecs[:,::-1]

    # Linear transformation
    A = U[:,:m].T

    #compute explained variance and visualize it
    cumulative_var_exp=0
    total = sum(L)
    var_explained = [(i/total) for i in L]
    cumulative_var_exp = np.cumsum(var_explained)

    if display:
        plt.bar(range(1,D+1), var_explained, alpha=0.5, align='center', label='indivic
        plt.step(range(1,D+1), cumulative_var_exp, alpha=0.5, where='mid', label='cumu
        plt.ylabel('Explained variance ratio')
        plt.xlabel('Principal components')
        plt.legend(loc='best');

    return A, var_explained
```

In [7]:
```
X_train.shape # N-by-D
```

Out[7]:
```
(124, 13)
```

In [8]:
```
A, var_explained = myPCA(X_train.T, 13, display=1)


A.shape
```

Out[8]:
```
(13, 13)
```



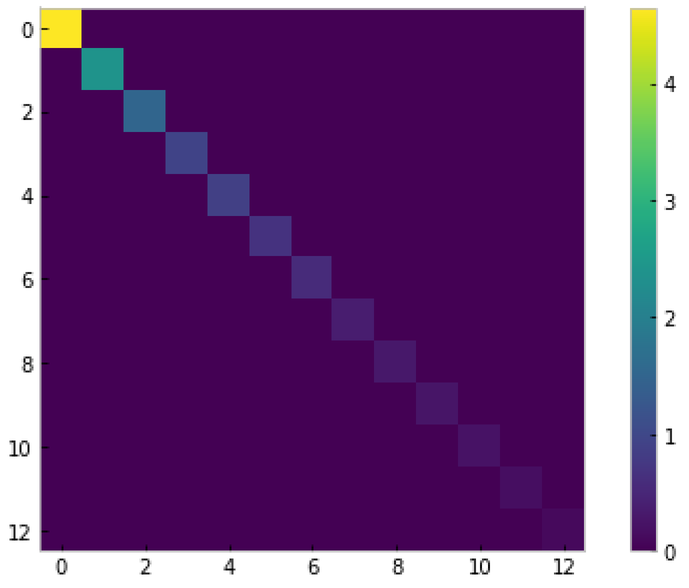
The resulting plot indicates that the first principal component alone accounts for 40 percent of the variance. Also, we can see that the first two principal components combined explain almost 60 percent of the variance in the data.

Although the explained variance plot reminds us of the feature importance, we shall remind ourselves that PCA is an unsupervised method, which means that information about the class labels is ignored.

In [10]:
```
# PCA transformation

X_train_pca = (A@X_train.T).T # N-by-D matrix


X_train_pca.shape
```

Out[10]:
```
(124, 13)
```

In [11]:
```
cov_mat = np.cov(X_train_pca.T)

plt.figure(figsize=(8,5))
plt.imshow(cov_mat)
plt.colorbar();
```



In [18]:
```
cov_mat
```

```
Out[18]: array([[ 4.65055604e+00, -1.24854195e-15, -1.64548025e-16,
                  1.11407024e-15, -8.14164841e-16,  8.50353368e-17,
                  7.35628817e-18,  1.99737724e-16, -7.62439885e-17,
                  1.67168792e-16,  2.47048415e-16,  2.05877572e-16,
                 -2.61759900e-16],
                [-1.24854195e-15,  2.39666980e+00,  5.76236166e-17,
                  1.47797532e-16, -3.06880064e-16, -4.25094454e-17,
                  4.02120608e-16,  2.46441830e-17,  5.26859396e-16,
                  3.62132692e-16, -1.27537693e-16,  2.90599077e-16,
                 -9.92882380e-17],
                [-1.64548025e-16,  5.76236166e-17,  1.51899223e+00,
                 -1.10165031e-16, -4.80949358e-16, -1.39132128e-18,
                 -4.93311390e-16,  1.70341541e-16,  6.13211889e-16,
                  2.05518679e-16,  6.86414533e-16,  2.58194118e-16,
                 -4.24231562e-17],
                [ 1.11407024e-15,  1.47797532e-16, -1.10165031e-16,
                  9.40848534e-01, -9.57186993e-16,  2.19090182e-16,
                  2.51079625e-16, -2.74324210e-16,  1.83409821e-16,
                  2.54558548e-16, -1.41566282e-16,  1.79383245e-16,
                  9.02620345e-17],
                [-8.14164841e-16, -3.06880064e-16, -4.80949358e-16,
                 -9.57186993e-16,  8.83711106e-01,  3.78123287e-16,
                  1.61527154e-16,  1.28391599e-16, -8.70432766e-17,
                 -1.34652399e-16, -1.72709162e-16, -1.61065653e-16,
                 -9.02620345e-17],
                [ 8.50353368e-17, -4.25094454e-17, -1.39132128e-18,
                  2.19090182e-16,  3.78123287e-16,  6.86506737e-01,
                 -2.38848006e-16, -2.49301130e-16,  6.30778672e-17,
                 -1.48406164e-16,  1.40893854e-16, -8.35309548e-17,
                  2.32876049e-16],
                [ 7.35628817e-18,  4.02120608e-16, -4.93311390e-16,
                  2.51079625e-16,  1.61527154e-16, -2.38848006e-16,
                  5.72667544e-01, -2.19812065e-16, -2.46991696e-16,
                 -3.40523191e-17, -4.92544054e-17,  3.96433118e-17,
                 -3.38708285e-16],
                [ 1.99737724e-16,  2.46441830e-17,  1.70341541e-16,
                 -2.74324210e-16,  1.28391599e-16, -2.49301130e-16,
                 -2.19812065e-16,  3.66142041e-01,  2.91131786e-16,
                  2.52234342e-16, -4.90794398e-18, -2.43526306e-16,
                 -7.22096276e-18],
                [-7.62439885e-17,  5.26859396e-16,  6.13211889e-16,
                  1.83409821e-16, -8.70432766e-17,  6.30778672e-17,
                 -2.46991696e-16,  2.91131786e-16,  2.97011180e-01,
                  6.50240297e-16, -2.70531008e-16, -3.42386527e-16,
                  1.43967945e-16],
                [ 1.67168792e-16,  3.62132692e-16,  2.05518679e-16,
                  2.54558548e-16, -1.34652399e-16, -1.48406164e-16,
                 -3.40523191e-17,  2.52234342e-16,  6.50240297e-16,
                  2.68432043e-01, -3.72166931e-17, -5.22407061e-16,
                 -1.69466970e-16],
                [ 2.47048415e-16, -1.27537693e-16,  6.86414533e-16,
                 -1.41566282e-16, -1.72709162e-16,  1.40893854e-16,
                 -4.92544054e-17, -4.90794398e-18, -2.70531008e-16,
                 -3.72166931e-17,  2.35176183e-01,  1.02193402e-16,
                 -1.44419255e-16],
                [ 2.05877572e-16,  2.90599077e-16,  2.58194118e-16,
                  1.79383245e-16, -1.61065653e-16, -8.35309548e-17,
                  3.96433118e-17, -2.43526306e-16, -3.42386527e-16,
                 -5.22407061e-16,  1.02193402e-16,  1.76308574e-01,
                  6.67939055e-17],
                [-2.61759900e-16, -9.92882380e-17, -4.24231562e-17,
                  9.02620345e-17, -9.02620345e-17,  2.32876049e-16,
                 -3.38708285e-16, -7.22096276e-18,  1.43967945e-16,
                 -1.69466970e-16, -1.44419255e-16,  6.67939055e-17,
                  1.12669051e-01]])
```

PCA with `scikit-learn`

In [12]:
```
from sklearn.decomposition import PCA
```

In [14]:
```
PCA?
```

In [15]:
```
pca = PCA(n_components=13)

pca
```

Out[15]:
```
PCA(n_components=13)
```

In [16]:
```
pca.fit(X_train)

# training the pca object
```

Out[16]:
```
PCA(n_components=13)
```

In [17]:
```
pca.explained_variance_
```

Out[17]:
```
array([4.65055604, 2.3966698 , 1.51899223, 0.94084853, 0.88371111,
       0.68650674, 0.57266754, 0.36614204, 0.29701118, 0.26843204,
       0.23517618, 0.17630857, 0.11266905])
```

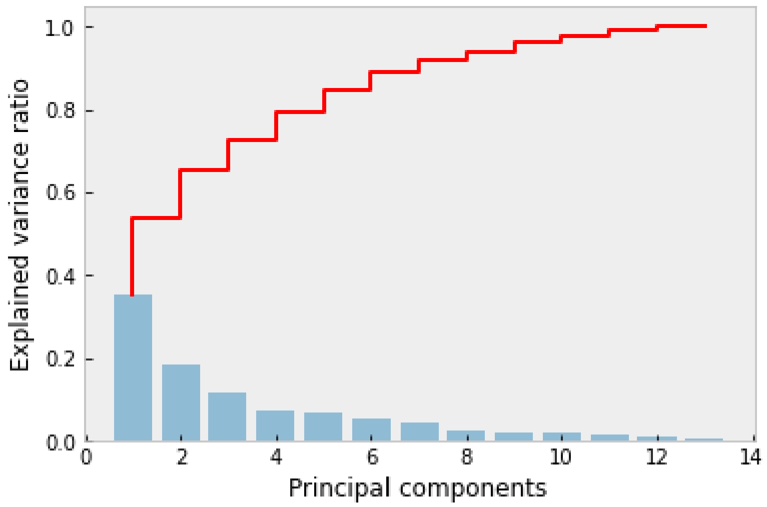
In [19]:
```
np.cumsum(pca.explained_variance_ratio_)
```

Out[19]:
```
array([0.35485012, 0.53772257, 0.65362582, 0.72541513, 0.7928447 ,
       0.84522704, 0.88892314, 0.91686077, 0.93952354, 0.96000563,
       0.97795022, 0.99140304, 1.        ])
```

In [20]:
```
# The matrix A = U.T is

pca.components_.shape
```

Out[20]:
```
(13, 13)
```

In [21]:
```
plt.step(range(1,14),np.cumsum(pca.explained_variance_ratio_),c='r')
plt.bar(range(1,14),pca.explained_variance_ratio_, alpha=0.5)
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal components');
```

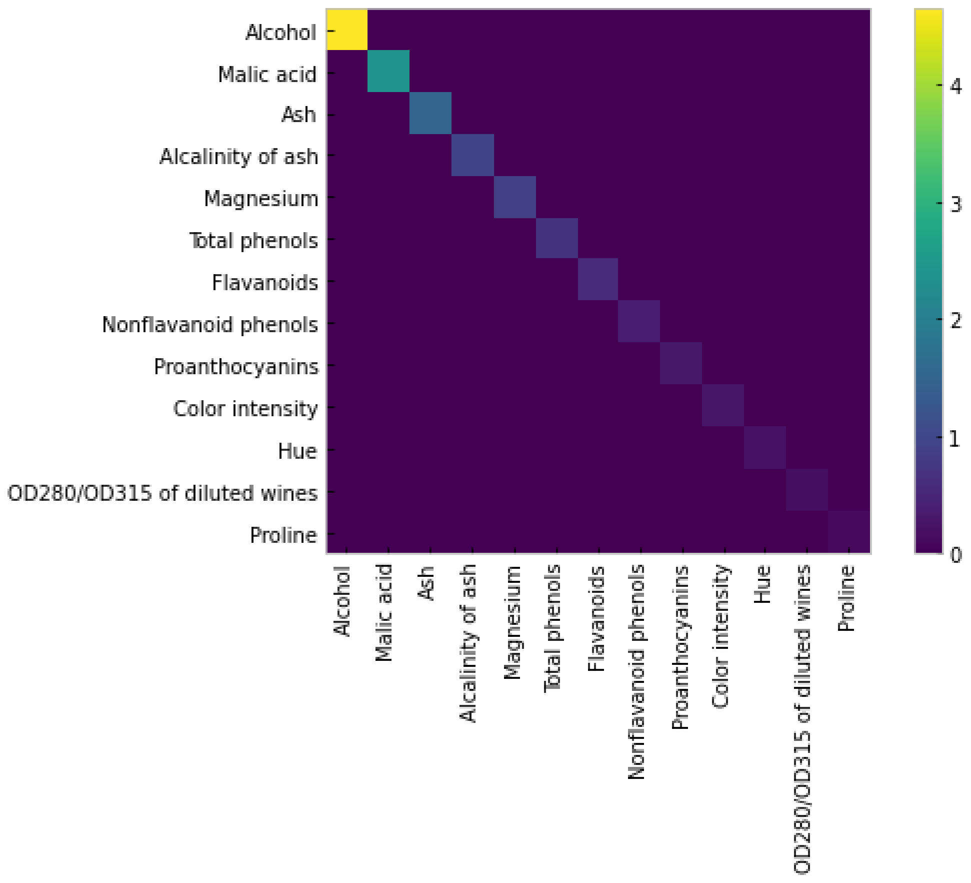


```
In [22]: np.where(np.cumsum(pca.explained_variance_ratio_) >=0.9)

Out[22]: (array([ 7,  8,  9, 10, 11, 12], dtype=int64),)
```

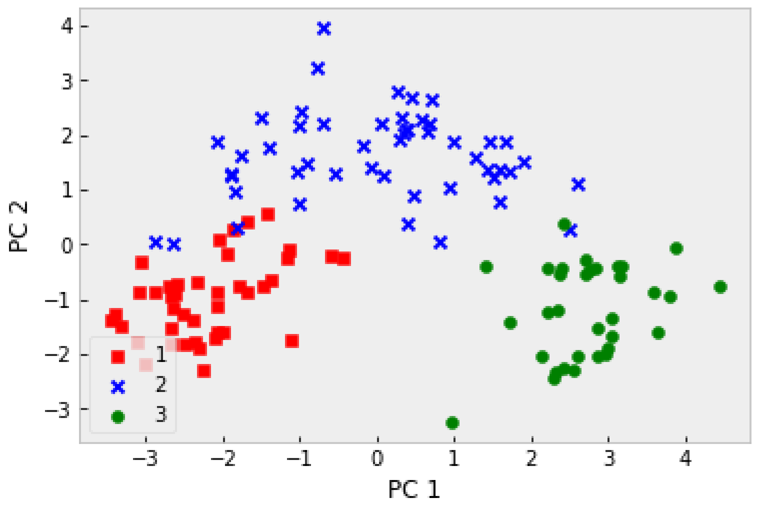
```
In [23]: np.cumsum(pca.explained_variance_ratio_)[7]

Out[23]: 0.9168607724765693
```

```
In [24]: y_train_pca = pca.transform(X_train) #A@X_train.T

         y_train_pca.shape

Out[24]: (124, 13)
```

```
In [25]: cov_mat = np.cov(y_train_pca.T)

         plt.figure(figsize=(8,5))
         plt.imshow(cov_mat)
         plt.colorbar()
         plt.xticks(range(13),df_wine.columns[1:],rotation=90)
         plt.yticks(range(13),df_wine.columns[1:]);
```



```
In [26]: colors = ['r', 'b', 'g']
         markers = ['s', 'x', 'o']

         for l, c, m in zip(np.unique(t_train), colors, markers):
             plt.scatter(y_train_pca[t_train==l, 0], y_train_pca[t_train==l, 1],c=c, label=l, m

         plt.xlabel('PC 1')
         plt.ylabel('PC 2')
         plt.legend(loc='lower left')
         plt.show()
```



The training data is used to find the new features (eigenvectors). We can then represent the test set in this new feature space:

```
In [27]: y_test_pca = pca.transform(X_test) #A@X_test.T

         y_test_pca.shape
```

```
Out[27]: (54, 13)
```

Example: Eigenfaces - Dataset Decomposition

```
In [28]: from sklearn.datasets import fetch_olivetti_faces

         faces = fetch_olivetti_faces(return_X_y=False)

         # print(faces.DESCR)
```

```
In [29]: X = faces.data # data matrix

         t = faces.target # target label

         X.shape, t.shape # 400 images, each of size 64x64=4096 pixels
```

```
Out[29]: ((400, 4096), (400,))
```

```
In [30]: fig = plt.figure(figsize=(10,10))
         for i in range(40):
             fig.add_subplot(7,6,i+1)
             idx = np.random.choice(np.where(t==i)[0])
             plt.imshow(X[idx,:].reshape(64,64), cmap='gray')
             plt.axis('off')
```

In [31]:
```
np.unique(t, return_counts=True)
```

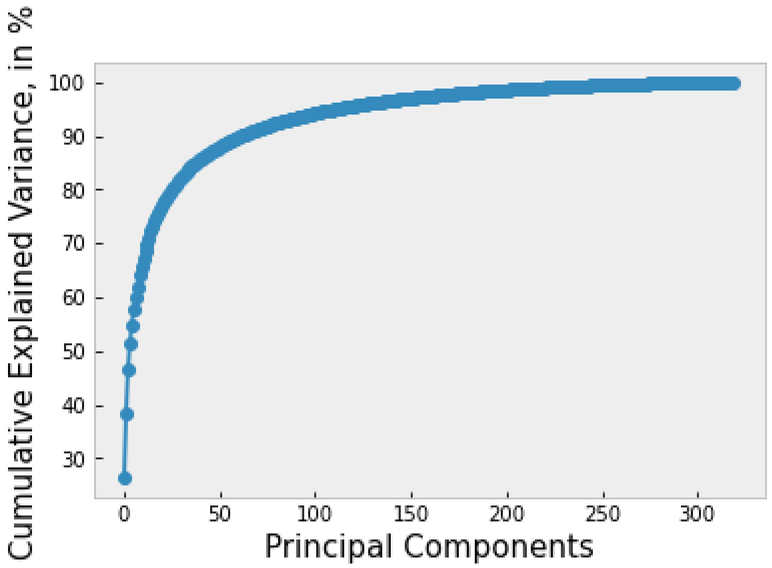
Out[31]:
```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
        34, 35, 36, 37, 38, 39]),
 array([10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
        10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
        10, 10, 10, 10, 10, 10], dtype=int64))
```

In [32]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, t_train, t_test = train_test_split(X, t,
                                                    test_size=0.2,
                                                    random_state=42)

X_train.shape, t_train.shape, X_test.shape, t_test.shape
```

Out[32]:
```
((320, 4096), (320,), (80, 4096), (80,))
```

In [33]:
```
scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)

X_test_scaled = scaler.transform(X_test)
```

In [34]:
```
pca = PCA(n_components=320)
pca.fit_transform(X_train_scaled);

plt.plot(100*np.cumsum(pca.explained_variance_ratio_), '-o')
plt.xlabel('Principal Components',size=15)
plt.ylabel('Cumulative Explained Variance, in %', size=15);
```



In [35]:
```
np.cumsum(pca.explained_variance_ratio_)
```

```
Out[35]:  array([0.26222053, 0.3837615 , 0.46618384, 0.51325184, 0.54706043,
                 0.57680434, 0.5999966 , 0.6201292 , 0.6393128 , 0.6561432 ,
                 0.6717103 , 0.6851146 , 0.6977544 , 0.7097174 , 0.7209915 ,
                 0.73112214, 0.73983663, 0.74830997, 0.75650233, 0.76390773,
                 0.7709245 , 0.7774063 , 0.78363425, 0.78951514, 0.7950188 ,
                 0.80026454, 0.80534273, 0.81023884, 0.8148485 , 0.81911826,
                 0.8231793 , 0.82722455, 0.83095217, 0.8344677 , 0.83793795,
                 0.84133005, 0.84446174, 0.84752935, 0.8505306 , 0.85339916,
                 0.85620475, 0.85882545, 0.8613985 , 0.86393654, 0.8664041 ,
                 0.86880726, 0.871115  , 0.87336886, 0.8755738 , 0.8777162 ,
                 0.879808  , 0.88180697, 0.8837133 , 0.8855643 , 0.88734275,
                 0.8891022 , 0.89083225, 0.8925449 , 0.8942304 , 0.8958646 ,
                 0.89746374, 0.8990333 , 0.900588  , 0.90208685, 0.90355617,
                 0.9050007 , 0.90642554, 0.90781415, 0.90916413, 0.91049826,
                 0.9118218 , 0.9131064 , 0.9143806 , 0.91561925, 0.91684157,
                 0.9180372 , 0.9191994 , 0.9203577 , 0.9214798 , 0.9225966 ,
                 0.9236907 , 0.92476964, 0.92581487, 0.9268568 , 0.9278625 ,
                 0.9288573 , 0.9298277 , 0.9307933 , 0.9317423 , 0.9326831 ,
                 0.9336107 , 0.9345241 , 0.93542236, 0.9363115 , 0.93717617,
                 0.9380158 , 0.93884397, 0.9396655 , 0.9404735 , 0.9412716 ,
                 0.9420559 , 0.94283026, 0.9435857 , 0.94432163, 0.9450492 ,
                 0.945764  , 0.9464769 , 0.9471851 , 0.94788563, 0.9485679 ,
                 0.9492469 , 0.9499102 , 0.95056844, 0.9512204 , 0.9518594 ,
                 0.9524891 , 0.9531107 , 0.9537238 , 0.9543294 , 0.9549234 ,
                 0.9555149 , 0.95609474, 0.9566704 , 0.95723253, 0.95779127,
                 0.95833766, 0.9588759 , 0.95940655, 0.95993596, 0.96045864,
                 0.9609728 , 0.9614824 , 0.9619807 , 0.9624758 , 0.9629634 ,
                 0.9634466 , 0.96392417, 0.9643954 , 0.96486306, 0.9653263 ,
                 0.965784  , 0.96623707, 0.96668744, 0.9671319 , 0.96756744,
                 0.96800137, 0.9684333 , 0.9688568 , 0.9692769 , 0.96968806,
                 0.9700944 , 0.97049797, 0.9708992 , 0.9712961 , 0.9716905 ,
                 0.97207665, 0.9724578 , 0.97283393, 0.97320133, 0.9735671 ,
                 0.97393054, 0.97428936, 0.97464466, 0.9749953 , 0.9753452 ,
                 0.9756902 , 0.97603256, 0.9763676 , 0.9766972 , 0.97702444,
                 0.9773465 , 0.9776642 , 0.97797984, 0.9782928 , 0.978604  ,
                 0.97890866, 0.9792104 , 0.9795071 , 0.979803  , 0.9800942 ,
                 0.98038435, 0.98067147, 0.9809546 , 0.9812348 , 0.9815126 ,
                 0.98178935, 0.98206455, 0.982336  , 0.9826016 , 0.9828646 ,
                 0.98312134, 0.9833772 , 0.9836309 , 0.9838804 , 0.98412836,
                 0.98437184, 0.98461163, 0.98485065, 0.9850862 , 0.9853194 ,
                 0.9855509 , 0.98577756, 0.986003  , 0.9862259 , 0.9864452 ,
                 0.9866629 , 0.98687845, 0.9870935 , 0.98730534, 0.98751235,
                 0.98771775, 0.98792243, 0.9881257 , 0.9883261 , 0.98852456,
                 0.9887204 , 0.9889133 , 0.9891039 , 0.9892923 , 0.98948   ,
                 0.98966587, 0.9898465 , 0.9900258 , 0.9902047 , 0.9903818 ,
                 0.99055773, 0.99073213, 0.9909046 , 0.9910746 , 0.9912419 ,
                 0.9914075 , 0.99157214, 0.991735  , 0.991896  , 0.9920551 ,
                 0.99221265, 0.9923682 , 0.99252164, 0.9926743 , 0.9928232 ,
                 0.9929707 , 0.99311715, 0.9932629 , 0.99340755, 0.99354863,
                 0.99368846, 0.99382734, 0.9939636 , 0.9940988 , 0.9942326 ,
                 0.9943629 , 0.9944916 , 0.9946191 , 0.99474585, 0.9948704 ,
                 0.99499416, 0.9951169 , 0.99523884, 0.99535984, 0.99547875,
                 0.9955956 , 0.9957106 , 0.99582446, 0.99593765, 0.99604845,
                 0.9961583 , 0.9962678 , 0.996374  , 0.99647915, 0.99658257,
                 0.9966852 , 0.9967871 , 0.9968869 , 0.99698544, 0.9970827 ,
                 0.99717844, 0.9972733 , 0.99736696, 0.997459  , 0.9975497 ,
                 0.9976381 , 0.9977256 , 0.9978119 , 0.99789715, 0.9979811 ,
                 0.9980647 , 0.99814606, 0.9982261 , 0.9983046 , 0.9983822 ,
                 0.9984583 , 0.9985338 , 0.9986061 , 0.9986777 , 0.99874854,
                 0.99881846, 0.99888754, 0.9989558 , 0.99902177, 0.99908715,
                 0.999151  , 0.9992136 , 0.99927413, 0.9993329 , 0.99939   ,
                 0.99944687, 0.9995025 , 0.9995562 , 0.9996075 , 0.9996563 ,
                 0.99970424, 0.9997502 , 0.9997943 , 0.9998359 , 0.9998743 ,
                 0.99991095, 0.9999459 , 0.99997455, 0.99999976, 0.99999976],
                dtype=float32)
```

In [36]:
```
np.where(np.cumsum(pca.explained_variance_ratio_) >= 0.9)
```

Out[36]:
```
(array([ 62,  63,  64,  65,  66,  67,  68,  69,  70,  71,  72,  73,  74,
         75,  76,  77,  78,  79,  80,  81,  82,  83,  84,  85,  86,  87,
         88,  89,  90,  91,  92,  93,  94,  95,  96,  97,  98,  99, 100,
        101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113,
        114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126,
        127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139,
        140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152,
        153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165,
        166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178,
        179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191,
        192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204,
        205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217,
        218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230,
        231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243,
        244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256,
        257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269,
        270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282,
        283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295,
        296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308,
        309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319], dtype=int64),)
```

In [37]:
```
np.cumsum(pca.explained_variance_ratio_)[62]
```

Out[37]:
```
0.900588
```

In order to explain 90% of the variance in the data, we need to preserve 63 principal components.

Let's project to 2-D so we can plot it:

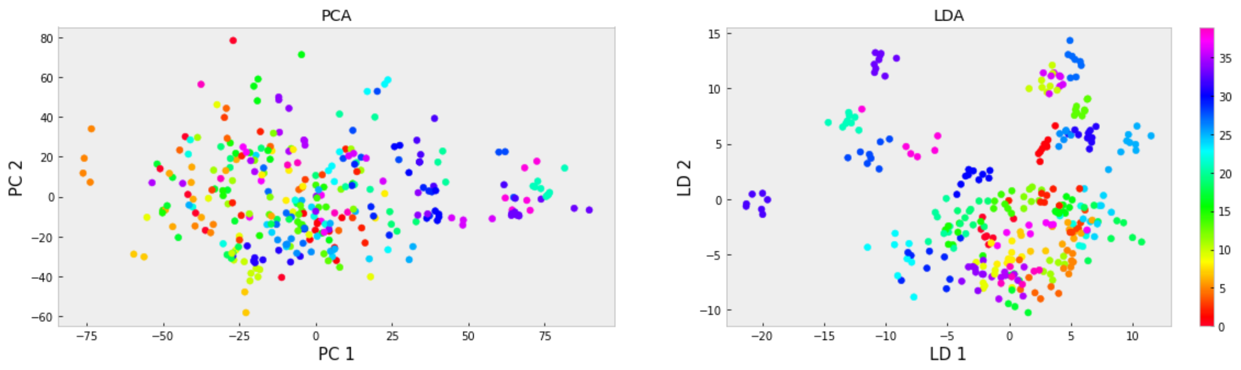
In [41]:
```
# PCA (unsupervised)

pca = PCA(n_components=2)
ypca = pca.fit_transform(X_train_scaled)

ypca.shape
```

Out[41]:
```
(320, 2)
```

In [42]:
```
# LDA (supervised)

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

lda = LDA(n_components=2)
ylda = lda.fit_transform(X_train_scaled, t_train)

ylda.shape
```

```
Out[42]: (320, 2)
```

```
In [43]: plt.figure(figsize=(20,5))
plt.subplot(1,2,1)
plt.scatter(ypca[:,0], ypca[:,1], c=t_train, cmap=plt.cm.gist_rainbow)
plt.xlabel('PC 1', size=15)
plt.ylabel('PC 2', size=15)
plt.title('PCA')

plt.subplot(1,2,2)
plt.scatter(ylda[:,0], ylda[:,1], c=t_train, cmap=plt.cm.gist_rainbow)
plt.xlabel('LD 1', size=15)
plt.ylabel('LD 2', size=15)
plt.title('LDA')
plt.rcParams['axes.grid'] = False
plt.colorbar();
```



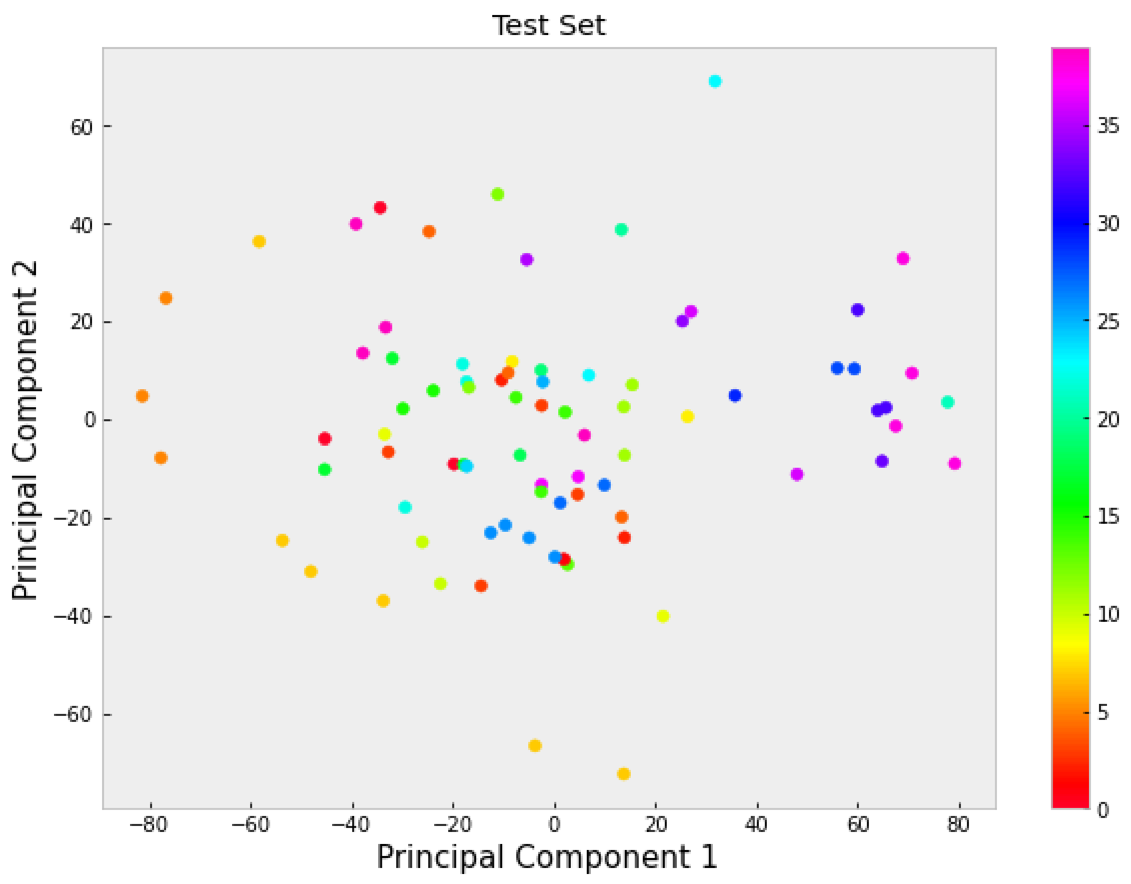
Not that the 40 classes are overlapping in the linear projection space. This is because PCA is **unsupervised**, it does use the class labels *anywhere* in finding the matrix for linear projection.

To apply this transformation in the test set, simply multiply the resultant modal matrix with the scaled test set:

```
In [44]: # Transform the test set using the linear transformation found with the training data
ypca_test = pca.transform(X_test_scaled)

ypca_test.shape
```

```
Out[44]: (80, 2)
```

```
In [45]: plt.figure(figsize=(10,7))
plt.scatter(ypca_test[:,0], ypca_test[:,1], c=t_test, cmap=plt.cm.gist_rainbow)
plt.xlabel('Principal Component 1', size=15)
plt.ylabel('Principal Component 2', size=15)
plt.title('Test Set')
plt.colorbar();
```



You can access the linear transformation $\mathbf{A} = \mathbf{U}^T$ using the method `components_`:

```
In [46]: A = pca.components_

         A.shape
```

```
Out[46]: (2, 4096)
```

Note that the eigenvectors are described in the original space, that is, they are 4096-dimensional!

Since we are working with images, we can reshape them back to a 64×64 image and see what are the regions in the image with maximum explained variance! This is called the **eigenfaces**.

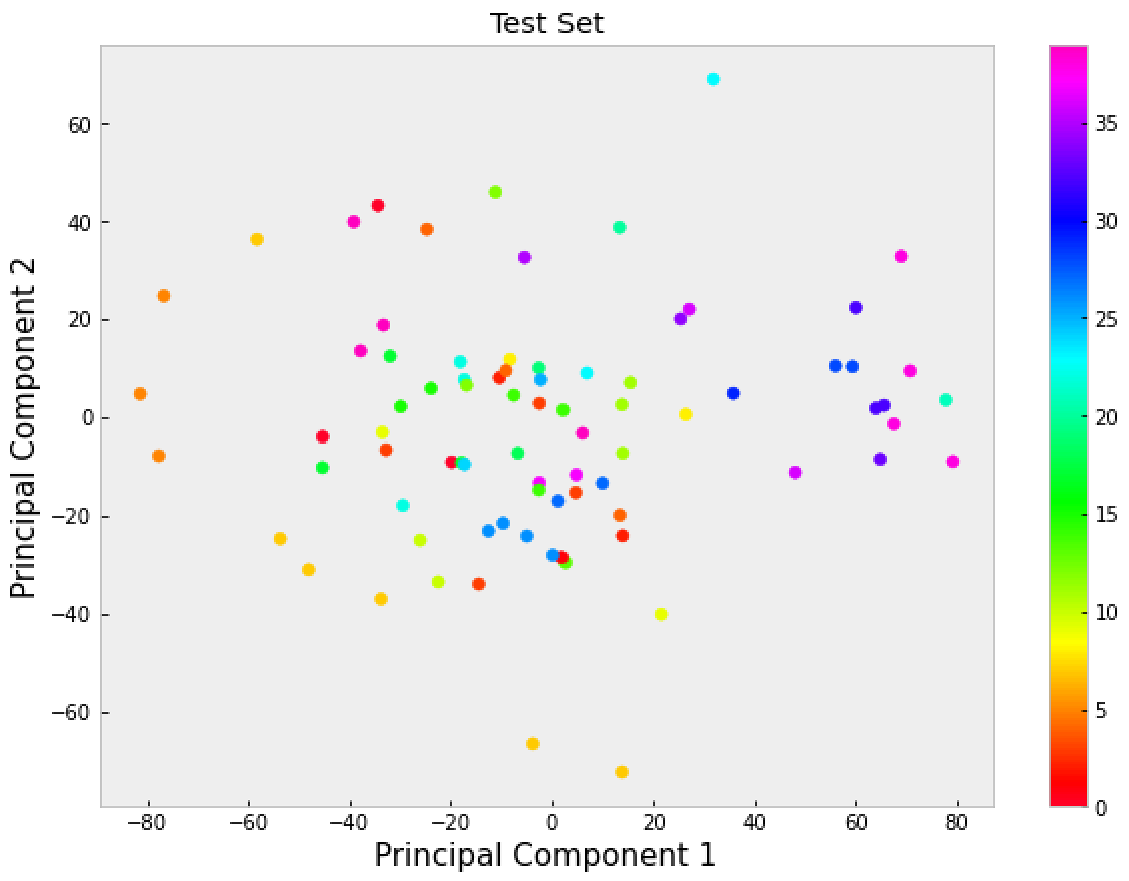
```
In [48]: X_test_scaled.shape
```

```
Out[48]: (80, 4096)
```

```
In [50]: yyy = (A@X_test_scaled.T).T

         yyy.shape
```

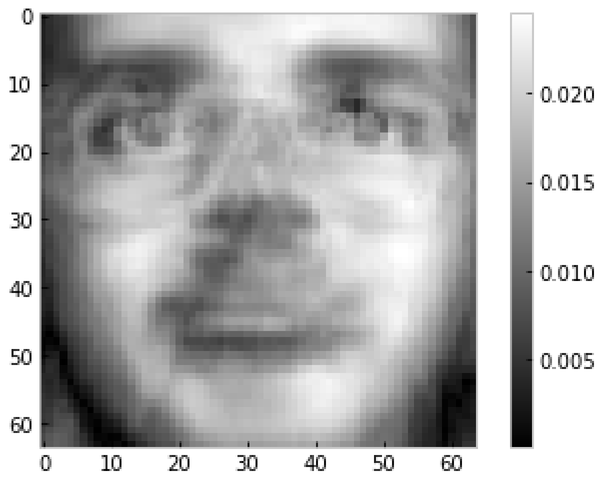
```
Out[50]: (80, 2)
```

```
In [51]: plt.figure(figsize=(10,7))
         plt.scatter(yyy[:,0], yyy[:,1], c=t_test, cmap=plt.cm.gist_rainbow)
         plt.xlabel('Principal Component 1', size=15)
         plt.ylabel('Principal Component 2', size=15)
         plt.title('Test Set')
         plt.colorbar();
```

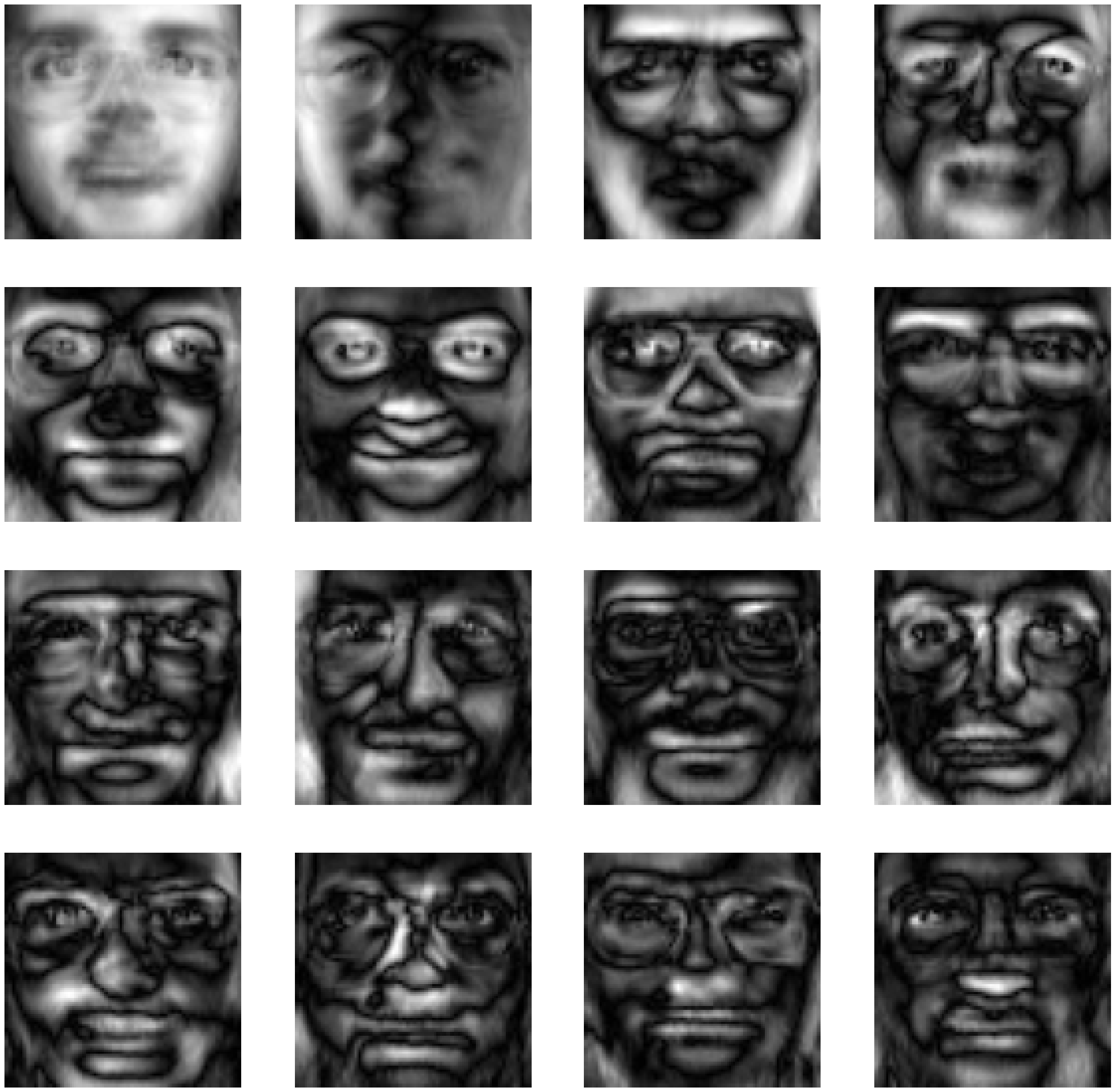


Let's now recover 16 eigenvectors and plot them as images:

In [56]:
```
plt.imshow(abs(pca.components_[0,:].reshape(64,64)), cmap='gray')
plt.colorbar();
```



In [57]:
```
n_components = 16

pca = PCA(n_components=n_components)
ypca = pca.fit_transform(X_train_scaled)

fig=plt.figure(figsize=(10,10))
for i in range(n_components):
    fig.add_subplot(4,4,i+1)
    plt.imshow(abs(pca.components_[i,:].reshape(64,64)),cmap='gray')
    plt.axis('off')
```



The eigenvectors are describing the regions in the 64x64 image that explain the most variance. the more eigenvectors are kept, the better a reconstruction image will be produced.

For example, let's reconstruct the images in the dataset using the top 16 eigenvectors:

In [58]:
```
K = np.cov(X_train_scaled.T)

L, U = np.linalg.eigh(K)

L
```

Out[58]:
```
array([-2.26986066e-13, -1.48546929e-13, -1.48270146e-13, ...,
        3.38660292e+02,  4.99392055e+02,  1.07742245e+03])
```

In [59]:
```
L = L[::-1]
U = U[:, ::-1] # eigenvectors are placed in columns
```

In [72]:
```
L
```

Out[72]:
```
array([ 1.07742245e+03,  4.99392055e+02,  3.38660292e+02, ...,
       -1.48270146e-13, -1.48546929e-13, -2.26986066e-13])
```

In [60]:
```
U.shape
```

Out[60]:
```
(4096, 4096)
```

In [63]:
```
N_eigenvectors = 9

P = U[:, :N_eigenvectors]

X_proj = X_train_scaled@P # (P@X_train_scaled.T).T
X_reconstruct = X_proj@np.linalg.pinv(P)

X_reconstruct.shape
```

Out[63]:
```
(320, 4096)
```

Since the projection is given by:

$$\mathbf{Y} = \mathbf{A}\mathbf{X}$$

In order to recover $\mathbf{X}$, we need to left-multiply by the pseudo-inverse of $\mathbf{A}$:

$$\hat{\mathbf{X}} = \mathbf{A}^{\dagger}\mathbf{Y}$$

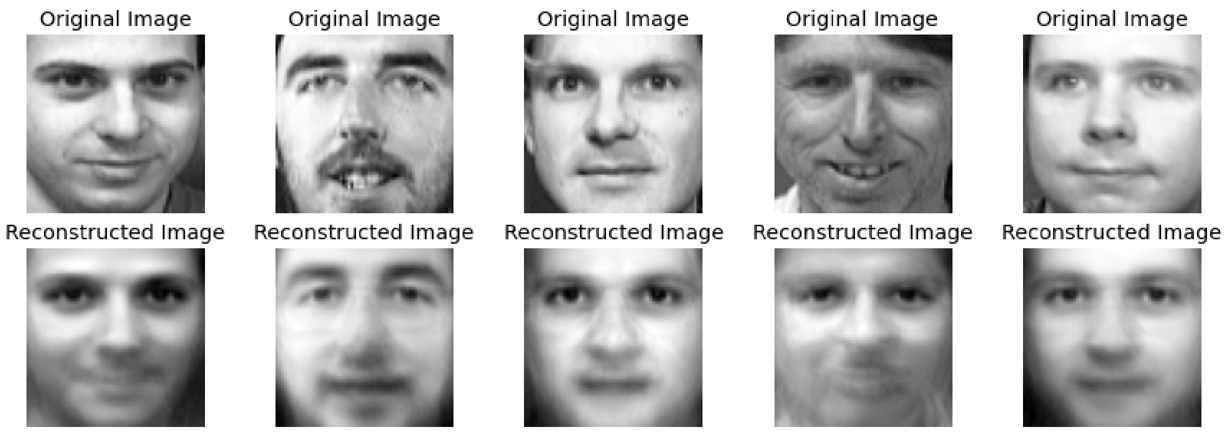
In [64]:
```
# Alternatively

ypca = pca.transform(X_train_scaled)
X_reconstruct_skl = pca.inverse_transform(ypca)

X_reconstruct_skl.shape
```

Out[64]:
```
(320, 4096)
```

We also need to bringing back to the original scaling: multiplying by the standard deviation and adding the sample mean value:

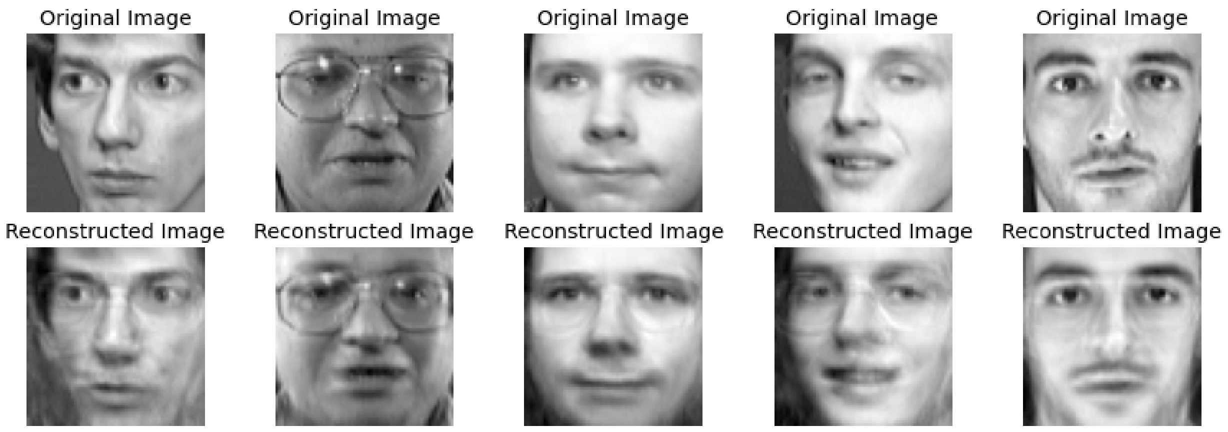
In [65]:
```
X_reconstructed = scaler.inverse_transform(X_reconstruct)

X_reconstructed_skl = scaler.inverse_transform(X_reconstruct_skl)
```

In [66]:
```
N = 5
idx = np.random.choice(range(X_reconstructed.shape[0]),replace=False,size=N)

fig = plt.figure(figsize=(15,5))

j=1
for i in range(N):
    fig.add_subplot(2,N,j)
    plt.imshow(X_train[idx[i],:].reshape(64,64), cmap='gray')
    plt.axis('off')
    plt.title('Original Image');

    fig.add_subplot(2,N,j+N)
    plt.imshow(X_reconstructed[idx[i],:].reshape(64,64), cmap='gray')
    plt.axis('off')
    plt.title('Reconstructed Image');
    j+=1
```



Putting it all together:

In [71]:
```
N_eigenvectors = 63

pca = PCA(n_components=N_eigenvectors)
ypca = pca.fit_transform(X_train_scaled)
X_reconstruct = pca.inverse_transform(ypca)
X_reconstructed = scaler.inverse_transform(X_reconstruct)

N = 5

fig = plt.figure(figsize=(15,5))
idx = np.random.choice(range(X_reconstructed.shape[0]),replace=False,size=N)
j=1
for i in range(N):
    fig.add_subplot(2,N,j)
    plt.imshow(X_train[idx[i],:].reshape(64,64), cmap='gray')
    plt.axis('off')
    plt.title('Original Image');

    fig.add_subplot(2,N,j+N)
    plt.imshow(X_reconstructed[idx[i],:].reshape(64,64), cmap='gray')
    plt.axis('off')
    plt.title('Reconstructed Image');
    j+=1
```



Still some compression loss but much better representation!

Performance Metrics

A key step in machine learning algorithm development and testing is determining a good error and evaluation metric.

Evaluation metrics help us to estimate how well our model is trained and it is important to pick a metric that matches our overall goal for the system.

Some common evaluation metrics include precision, recall, receiver operating curves, and confusion matrices.

Classification Accuracy and Error

Classification accuracy and e the number of correct predictions made as a ratio of all predictions made.

- **Classification accuracy** is defined as the number of correctly classified samples divided by all samples:

$$\text{accuracy} = \frac{N_{\text{corr}}}{N}$$

where N_{corr} is the number of correct classified samples and N is the total number of samples.

- **Classification error** is defined as the number of incorrectly classified samples divided by all samples:

$$\text{error} = \frac{N_{\text{miss}}}{N}$$

where N_{miss} is the number of misclassified samples and N is the total number of samples.

- Classification accuracy is the most common evaluation metric for classification problems, it is also the most misused. It is really only suitable when there are an equal number of observations in each class (which is rarely the case) and that all predictions and prediction errors are equally important, which is often not the case.

Example 1: Fish Dataset

Suppose there is a 3-class classification problem, in which we would like to classify each training sample (a fish) to one of the three classes (A = salmon or B = sea bass or C = cod).

Let's assume there are 150 samples, including 30 salmon, 40 sea bass and 80 cod. Suppose our model misclassifies 4 salmon, 2 sea bass and 5 cod.

- The classification accuracy (ACC) of our binary classification model is calculated as:

$$\mathrm{ACC} = \frac{26+38+75}{30+40+80} = \frac{139}{150} \approx 92.7\%$$

- The prediction error is calculated as:

$$\mathrm{error} = \frac{4+2+5}{30+40+80} = \frac{11}{150} \approx 7.3\%$$

- The classification accuracy doesn't really gives an insight on which class is being misclassified the most.

Confusion Matrix

A confusion matrix summarizes the classification accuracy across several classes. It shows the ways in which the classification model is confused when it makes predictions, allowing visualization of the performance of our algorithm.

Generally, each row represents the instances of a actual class while each column represents the instances in an predicted class.

If the classifier is trained to distinguish between salmon, sea bass and cod. We can summarize the prediction result in the confusion matrix as follows:

actual/predict	salmon	sea bass	cod
salmon	26	2	2
sea bass	2	38	0
cod	2	3	75

In this confusion matrix, of the 30 salmons (row 1), the classifier predicted that 26 are labeled salmon correctly, 2 are wrongly labeled as sea bass, and another 2 are wrongly labeled as cod.

All correct predictions are located in the diagonal of the table. So it is easy to visually inspect the table for prediction errors, as they will be represented by values outside the diagonal.

Precision, Recall & Fall-Out

We are often looking to discriminate between observations with a specific binary outcome, for example, event or no event. In our example, the fish company would like to produce salmon can but the harvest contains all three species. In this way, we can assign the event (salmon) as "positive" and no-event (not salmon) as "negative".

The confusion matrix for this two-class classification problem is:

actual/predict	salmon	non-salmon

actual/predict	salmon	non-salmon
salmon	26	4
non-salmon	4	116

- **True positive (TP):** correctly predicting positive events
- **False positive (FP):** incorrectly calling positive to a negative event
- **True negative (TN):** correctly predicting negative events
- **False negative (FN):** incorrectly labeling negative to a positive event

In this salmon/non-salmon classification problem, what are the TP, FP, TN, FN values?

actual/predict	Positive	Negative
Positive	TP	FN
Negative	FP	TN

- **Precision**, also called Positive Predictive Value (PPV), is the performance of detection

$$\text{Precision} = \text{PPV} = \frac{TP}{TP + FP}$$

- **Recall**, also called True Positive Rate (TPR) or Sensitivity, is the probability of detection

$$\text{Recall} = \text{TPR} = \text{Sensitivity} = \frac{TP}{TP + FN}$$

- **Fall-out**, also called False Positive Rate (FPR), is the probability of false alarm

$$\text{Fall-out} = \text{FPR} = \frac{FP}{FP + TN}$$

- **Specificity**, also called True Negative Rate (TNR), is the probability of negative events detection

$$\text{Specificity} = \frac{TN}{TN + FP}$$

- **F1-score**, also called F-score or F-measure, is a measure of a model's accuracy. It considers both the precision and the recall.

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- Learn about many other measures on this Wikipedia page and Scikit-Learn's Classification Metrics Module.

ROC Curves

Receiver Operating Characteristic (ROC) curve is the plot between the true positive rate (TPR) and the false positive rate (FPR), where the TPR is defined as the y-axis and FPR is defined as the x-axis.

- ROC curves were first developed for RADAR systems, hence the name.
- Given a binary classifier and its threshold, the (x,y) coordinates of ROC space can be calculated from all the prediction result. You trace out a ROC curve by varying the threshold to get all of the points on the ROC.
- The diagonal between (0,0) and (1,1) separates the ROC space into two areas, which are left up area and right bottom area. The points above the diagonal represent good classification (better than random guess) which below the diagonal represent bad classification (worse than random guess).
- *What is the perfect prediction point in a ROC curve?*

Area Under the Curve (AUC)

Area Under Curve (AUC) is a common measure of how good a test is. It is simply the area under the ROC curve. Random guessing can achieve the diagonal line, so the minimum AUC is 1/2. The maximum AUC is 1, which is achieved by a test that is always right; the ROC curve is along the left and top axes.

Example

1. Suppose you have a target detection task that you would like to evaluate using ROC curve analysis. You emplaced 5 targets and collected aerial hyperspectral imagery over 10 km^2. Then, suppose you ran a set of alarm generation and target detection algorithms over the collected data. Your algorithms produced the following list of alarm confidence values. You have already matched each of these alarms to a location on the ground and compared them with your ground truth. True targets, based on your ground truth, are marked with a "T" in the second column. Draw the associated ROC cure for these results.

Alarm confidence values	**0.91**	**0.90**	**0.80**	**0.79**	**0.77**	**0.75**	**0.50**	**0.40**	**0.39**	**0.38**	**0.37**	**0.25**	**0.10**	**0.09**	**0.01**
Ground truth	T	T		T				T						T	

1. Suppose you were segmenting a data set into three classes (e.g., vegetation, man-made materials, sand) and wanted to evaluate your results. Would using a ROC curve be an appropriate method for evaluation? Why or why not?