assignment4_pca-alg

April 18, 2021

1 Objective: To develop a mathematical understanding of PCA

2 Test data

[1]:
```
import numpy as np
from scipy import linalg as LA
X = np.array([
        [0.387,4878, 5.42],
        [0.723,12104,5.25],
        [1,12756,5.52],
        [1.524,6787,3.94],
    ])
X = X - np.mean(X, axis=0)
print(X.shape)
```

```
(4, 3)
```

3 Non-Linear Iterative Partial Least-Squares (NIPALS) algorithm

Steps to compute PCA using NIPALS algorithm

- Step 1: Initialize an arbitrary column vector $\mathbf{t}_a$ either randomly or by just copying any column of X.
- Step 2: Take every column of $\mathbf{X}$, $\mathbf{X_k}$ and regress it onto the $\mathbf{t}_a$ vector and store the regression coefficeints as $\mathbf{p}_{ka}$. (Note: This simply means performing an ordinary least squares regression ($y = mx$) with $x = t_a$ and $y = X_k$ with $m = (\mathbf{x^T x})^{-1}\mathbf{x^T y}$). In the current notation we get

$$p_{ka} = \frac{\mathbf{t_a^T X}_k}{\mathbf{t_a^T t_a}}$$

Repeat it for each of the columns of X to get the entire vector $\mathbf{p}_k$. This is shown in the illustration above where each column from X is regressed, one at a time, on $\mathbf{t}_a$, to calculate the loading entry, $_{ka}$

In practice we don't do this one column at time; we can regress all columns in X in go:

$$\mathbf{p_a^T} = \frac{1}{\mathbf{t_a^T t_a}}.\mathbf{t_a^T X_a}$$

where $\mathbf{t_a}$ is an $N \times 1$ column vector, and $\mathbf{X}_a$ us an $N \times K$ matrix. * The loading vector $\mathbf{p_a^T}$ won't have unit length (magnitude) yet. So we simply rescale it to have magnitude of 1.0:

$$\mathbf{p_a^T} = \frac{\mathbf{p_a^T}}{\sqrt{\mathbf{p_a^T p_a}}}$$

* Step 4: Regress every row in X onto this normalized loadings vector. As illustrated below, in our linear regression the rows in X are our y-variable each time, while the loadings vector is our x-variable. The regression coefficient becomes the score value for that th row:

$$t_{i,a} = \frac{\mathbf{x}_i^T \mathbf{p}_a}{\mathbf{p}_a^T \mathbf{p}_a}$$

where x_i^T is a $K \times 1$ column vector. We can combine these N separate least-squares models and calculate them in one go to get the entire vector,

$$\mathbf{t}_a^T = \frac{1}{\mathbf{p}_a^T \mathbf{p}_a} \mathbf{X} \mathbf{p}_a^T$$

where p_a is a $K \times 1$ column vector. * Step 5: Continue looping over steps 2,3,4 until the change in vector t_a is below a chosen tolerance * Step 6: On convergence, the score vector and the loading vectors, $\mathbf{t}_a$ and $\mathbf{p}_a$ are stored as the a^{th} column in matrix $\mathbf{T}$ and $\mathbf{P}$. We then deflate the $\mathbf{X}$ matrix. This crucial step removes the variability captured in this component (t_a and p_a) from $\mathbf{X}$:

$$E_a = X_a - t_a p_a^T$$

$$X_{a+1} = E_a$$

For the first component, X_a is just the preprocessed raw data. So we can see that the second component is actually calculated on the residuals E_1, obtained after extracting the first component. This is called deflation, and nicely shows why each component is orthogonal to the others. Each subsequent component is only seeing variation remaining after removing all the others; there is no possibility that two components can explain the same type of variability. After deflation we go back to step 1 and repeat the entire process for the next component.

In short:

1. Guess a scores vector.
2. Find a loadings vector that fits this.
3. Normalize the found loadings.
4. Do regression on all rows of X to find the new scores.
5. Do step 2-4 over and over until change is small enough.
6. The scores and loadings left are the PCs.

3.1 IMPLEMENTATION IN PYTHON

[2]:
```
def PCA(X,no_components):
    tol = 0.0000001
    it=1000
    obsCount,varCount = X.shape
    Xa = X - np.mean(X, axis = 0)
    #Xh = X-np.tile(np.mean(X,axis=0).reshape(-1,1).T, obsCount).reshape(4,3)
    T = np.zeros((obsCount,no_components))
    P = np.zeros((varCount,no_components))
    pcvar = np.zeros((varCount,1))
    varTotal = np.sum(np.var(Xa,axis=0,ddof=1))
    currVar = varTotal
    nr=0
    for h in range(no_components):
        th = Xa[:,0].reshape(-1,1)
        ende = False
        while ende != True:
            nr = nr + 1
            ph = np.dot(Xa.T,th)/np.dot(th.T,th)
            ph = ph /np.linalg.norm(ph)
            thnew = np.dot(Xa,ph)/np.dot(ph.T,ph)
            prec = np.dot((thnew-th).T,(thnew-th))
            th = thnew
            if prec <= (tol*tol):
                ende = True
            elif it <=nr:
                ende = True
                print("Iternation stops without convergence")
        Ea = Xa - np.dot(th,ph.T)
        Xa = Ea
        T[:,h] = th.flatten()
        P[:,h] = ph.flatten()
        oldVar = currVar
        currVar = np.sum(np.var(Xa,axis=0,ddof=1))
        pcvar[h] = (oldVar - currVar) / varTotal
        nr = 0
    return T,P,pcvar
```

3.2 Advantages of the NIPALS algorithm

- The NIPALS algorithm computes one component at a time. The first component computed is equivalent to the t1 and p1 vectors that would have been found from an eigenvalue or singular value decomposition.
- The algorithm can handle missing data in X.
- The algorithm always converges, but the convergence can sometimes be slow.
- It is also known as the Power algorithm to calculate eigenvectors and eigenvalues.
- It works well for very large data sets.

- It is used by most software packages, especially those that handle missing data.
- Of interest: it is well known that Google used this algorithm for the early versions of their search engine, called PageRank148.

[3]:
```
no_components=3
T,P,pcvar = PCA(X,no_components)
print("T (Scores)")
print(T)
print(" ")
print("P (Loadings)")
print(P)
print(np.sqrt(pcvar)/np.sum(np.sqrt(pcvar)))
```

```
T (Scores)
[[-4.25324997e+03 -8.41288672e-01  8.37859036e-03]
 [ 2.97275001e+03 -1.25977272e-01 -1.82476780e-01]
 [ 3.62475003e+03 -1.56843494e-01  1.65224286e-01]
 [-2.34425007e+03  1.12410944e+00  8.87390330e-03]]

P (Loadings)
[[ 1.21901390e-05  5.66460728e-01  8.24088735e-01]
 [ 9.99999997e-01  5.32639787e-05 -5.14047689e-05]
 [ 7.30130279e-05 -8.24088733e-01  5.66460726e-01]]
[[9.99753412e-01]
 [2.10083377e-04]
 [3.65048880e-05]]
```

4 SVD

[4]:
```
from numpy.linalg import svd
U, S, PTrans = svd(X, full_matrices=False)
Sigma = np.diag(S)
T=np.dot(U,Sigma)
P=PTrans.T

print("T (Scores)")
print(T)
print(" ")
print("P (Loadings)")
print(P)
print("Sigma (Variance)")
print(S)
```

```
T (Scores)
[[-4.25324997e+03 -8.41288672e-01 -8.37858943e-03]
 [ 2.97275001e+03 -1.25977271e-01  1.82476780e-01]
 [ 3.62475003e+03 -1.56843494e-01 -1.65224286e-01]
```

```
 [-2.34425007e+03  1.12410944e+00 -8.87390454e-03]]

P (Loadings)
[[ 1.21901390e-05  5.66460727e-01 -8.24088736e-01]
 [ 9.99999997e-01  5.32639789e-05  5.14047691e-05]
 [ 7.30130279e-05 -8.24088734e-01 -5.66460725e-01]]
Sigma (Variance)
[6.74994067e+03 1.41840009e+00 2.46466604e-01]
```

5 SKLEARN PCA

[5]:
```
from sklearn.decomposition import PCA
pca = PCA()
T=pca.fit_transform(X)
Prans=pca.components_ #eigen vectors.T
latent = pca.explained_variance_
explained = pca.explained_variance_ratio_
P=PTrans.T
S=pca.singular_values_
Sigma=np.diag(S)
print("T (Scores)")
print(T)
print(" ")
print("P (Loadings)")
print(P)
print("Sigma (Variance)")
print(S)
#print(pca.singular_values_/np.sqrt(3))
```

```
T (Scores)
[[ 4.25324997e+03 -8.41288672e-01 -8.37858943e-03]
 [-2.97275001e+03 -1.25977271e-01  1.82476780e-01]
 [-3.62475003e+03 -1.56843494e-01 -1.65224286e-01]
 [ 2.34425007e+03  1.12410944e+00 -8.87390454e-03]]

P (Loadings)
[[ 1.21901390e-05  5.66460727e-01 -8.24088736e-01]
 [ 9.99999997e-01  5.32639789e-05  5.14047691e-05]
 [ 7.30130279e-05 -8.24088734e-01 -5.66460725e-01]]
Sigma (Variance)
[6.74994067e+03 1.41840009e+00 2.46466604e-01]
```

[6]:
```
pca.explained_variance_ratio_
```

[6]:
```
array([9.99999955e-01, 4.41567976e-08, 1.33326424e-09])
```

[7]:
```
explained_variance_2 = (S ** 2) / 4
explained_variance_ratio_2 = (explained_variance_2 / explained_variance_2.sum())
print(explained_variance_ratio_2)
```

```
[9.99999955e-01 4.41567976e-08 1.33326424e-09]
```

6 Eigenvalue decomposition approach

Recall that the latent variable directions (the loading vectors) were oriented so that the variance of the scores in that direction were maximal. We can cast this as an optimization problem. For the first component:

$$max(\phi) = \mathbf{t_1^T t_1} = \mathbf{p_1^T X^T X p_1}$$

subject to

$$\mathbf{p_1^T p_1} = 1$$

.

This is equivalent to

$$max(\phi) = \mathbf{p_1^T X^T X p_1} - \lambda(\mathbf{p_1^T p_1} - 1)$$

because we can move the constraint into the objective function with a Lagrange multiplier, λ. The maximum value must occur when the partial derivatives with respect to $\mathbf{p_1}$,

our search variable, are zero:

$$\frac{\partial \phi}{\partial p_1} = \frac{\partial(\mathbf{p_1^T X^T X p_1} - \lambda(\mathbf{p}_1^T \mathbf{p}_1 - 1))}{\partial \mathbf{p}_1} = 0$$

$$2\mathbf{X^T X p_1} - 2\lambda_1 \mathbf{p_1} = 0$$

$$(\mathbf{X^T X} - \lambda_1 \mathbf{I})\mathbf{p_1} = 0$$

$$\mathbf{X^T X p_1} = \lambda_1 \mathbf{p_1}$$

which is just the eigenvalue equation, indicating that $\mathbf{p_1}$ is the eigenvector of $\mathbf{X^T X}$ and λ_1 is the eigenvalue. One can show that $\lambda_1 = \mathbf{t_1^T t_1}$, which is proportional to the variance of the first component. In a similar manner we can calculate the second eigenvalue, but this time we add the additional constraint that $\mathbf{p}_1 \perp \mathbf{p}_2$. Writing out this objective function and taking partial derivatives leads to showing that

$$\mathbf{X^T X p_2} = \lambda_2 \mathbf{p_2}$$

.

From this we learn that: * The loadings are the eigenvectors of $\mathbf{X^T X}$. * Sorting the eigenvalues in order from largest to smallest gives the order of the corresponding eigenvectors, the loadings. * We know from the theory of eigenvalues that if there are distinct eigenvalues, then their eigenvectors are

linearly independent (orthogonal). * We also know the eigenvalues of $\mathbf{X}^\mathbf{T}\mathbf{X}$ must be real values and positive; this matches with the interpretation that the eigenvalues are proportional to the variance of each score vector. * Also, the sum of the eigenvalues must add up to sum of the diagonal entries of $\mathbf{X}^\mathbf{T}\mathbf{X}$, which represents of the total variance of the $\mathbf{X}$ matrix, if all eigenvectors are extracted. So plotting the eigenvalues is equivalent to showing the proportion of variance explained in X by each component. This is not necessarily a good way to judge the number of components to use, but it is a rough guide: use a Pareto plot of the eigenvalues (though in the context of eigenvalue problems, this plot is called a scree plot).

[8]:
```
cov = np.cov(X, rowvar = False)
evals , P = LA.eigh(cov)
idx = np.argsort(evals)[::-1]
P = P[:,idx]
evals = evals[idx]
T = np.dot(X, P)
Sigma=LA.norm(T,axis=0)
print("T (Scores)")
print(T)
print("P (Loadings)")
print(P)
print("Sigma (Variance)")
print(Sigma)
```

```
T (Scores)
[[ 4.25324997e+03  8.41288672e-01  8.37858943e-03]
 [-2.97275001e+03  1.25977271e-01 -1.82476780e-01]
 [-3.62475003e+03  1.56843494e-01  1.65224286e-01]
 [ 2.34425007e+03 -1.12410944e+00  8.87390454e-03]]
P (Loadings)
[[-1.21901390e-05 -5.66460727e-01  8.24088736e-01]
 [-9.99999997e-01 -5.32639789e-05 -5.14047691e-05]
 [-7.30130279e-05  8.24088734e-01  5.66460725e-01]]
Sigma (Variance)
[6.74994067e+03 1.41840009e+00 2.46466604e-01]
```

6.1 Task 1: Test if the loading vectors are orthogonal and orthonormal or not

[9]:
```
from itertools import combinations

for i, j in combinations(range(P.shape[1]), 2):
    d = P[:,i].dot(P[:,j])
    print(f"P{i} · P{j} = {d}")
```

```
P0 · P1 = -1.8270014701286074e-19
P0 · P2 = -2.7070685671791457e-19
P1 · P2 = -1.0080824159617591e-16
```

[10]:
```
for i in range(P.shape[1]):
    l = np.linalg.norm(P[:,i])
    print(f"||P{i}|| = {l}")
```

```
||P0|| = 0.9999999999999999
||P1|| = 1.0
||P2|| = 1.0
```

The dot products are very small, so the loading vectors are more or less pairwise orthogonal. The norms are close to or exactly unit length. Hence, the loading vectors is approximately orthonormal.

6.2 Task 2: Test if the scores vectors are orthogonal and orthonormal or not

[11]:
```
from itertools import combinations

for i, j in combinations(range(T.shape[1]), 2):
    d = T[:,i].dot(T[:,j])
    print(f"T{i} · T{j} = {d}")
```

```
T0 · T1 = -1.8189894035458565e-11
T0 · T2 = -1.830358087318018e-11
T1 · T2 = -1.4030443473700416e-14
```

[12]:
```
for i in range(T.shape[1]):
    l = np.linalg.norm(T[:,i])
    print(f"||T{i}|| = {l}")
```

```
||T0|| = 6749.940666380364
||T1|| = 1.41840008831963
||T2|| = 0.24646660404039109
```

The dot products are very small, so the score vectors are very close to pairwise orthogonal. However, their norms are nowhere near unit length, so the score vectors are not orthonormal.

6.3 Task 3: Add more columns to the original data matrix by:

- Make some of the columns to be the linear combination of others
- Duplicate some columns
- Add noise as some columns
- Add a few columns of categorical values

Then apply PCA to the dataset and report your findings here

[13]:
```
# Add linear combination
x = X[:,0] - 2*X[:,1]
X = np.append(X, x[:,None], axis=1)

x = X[:,2] - X[:,0]
X = np.append(X, x[:,None], axis=1)
```

```
# Duplicate
x = X[:,1:3]
X = np.append(X, x, axis=1)

# Add noise
x = np.random.uniform(-1, 1, (4,3))
X = np.append(X, x, axis=1)

# Add categorical
x = np.array([[0, 0, 1, 0], [1, 0, 0, 0], [0, 0, 0, 1]]).T
X = np.append(X, x, axis=1)

print(X.shape)
print(X)
```

```
(4, 13)
[[-5.21500000e-01 -4.25325000e+03  3.87500000e-01  8.50597850e+03
   9.09000000e-01 -4.25325000e+03  3.87500000e-01 -3.71249390e-01
   4.00973840e-01  8.70698019e-01  0.00000000e+00  1.00000000e+00
   0.00000000e+00]
 [-1.85500000e-01  2.97275000e+03  2.17500000e-01 -5.94568550e+03
   4.03000000e-01  2.97275000e+03  2.17500000e-01  1.90831421e-01
   1.57254101e-01  6.13245861e-01  0.00000000e+00  0.00000000e+00
   0.00000000e+00]
 [ 9.15000000e-02  3.62475000e+03  4.87500000e-01 -7.24940850e+03
   3.96000000e-01  3.62475000e+03  4.87500000e-01  2.48591305e-01
   7.32978128e-02 -1.98154401e-01  1.00000000e+00  0.00000000e+00
   0.00000000e+00]
 [ 6.15500000e-01 -2.34425000e+03 -1.09250000e+00  4.68911550e+03
  -1.70800000e+00 -2.34425000e+03 -1.09250000e+00 -7.08481855e-01
   6.73947283e-01  8.57206582e-01  0.00000000e+00  0.00000000e+00
   1.00000000e+00]]
```

[14]:
```
U, S, V = np.linalg.svd(X)
P = V.T
T = U @ np.diag(S)

print(f"P (loadings), shape = {P.shape}")
print(P)
print()
print(f"T (scores), shape = {T.shape}")
print(T)
```

```
P (loadings), shape = (13, 13)
[[ 4.97457238e-06  2.69491315e-01  5.69687663e-02 -2.31899940e-01
   5.37466765e-01 -3.82501210e-01  1.94088570e-01  1.28411954e-01
  -9.49558583e-02  2.59691783e-01 -4.79036715e-01  8.83386713e-02
```

```
 -2.59378825e-01]
[ 4.08249947e-01  8.98505761e-02  1.89684910e-02 -7.72828796e-02
 -1.61249933e-01 -1.73764244e-01 -6.73061617e-02  1.45209969e-01
 -3.44164566e-01 -6.65705387e-01 -1.77733535e-01 -3.16148738e-01
 -2.09672733e-01]
[ 2.98102850e-05 -3.94586051e-01 -4.80694293e-02 -1.45031891e-01
 -3.66859239e-01 -3.98920922e-04 -3.61611919e-01 -3.23580039e-02
  2.37476277e-03  3.25049935e-01 -6.29919410e-01 -2.14582731e-01
  7.78044732e-02]
[-8.16494919e-01  8.97901629e-02  1.90317843e-02 -7.73341809e-02
  2.36643974e-02  3.34742242e-01  4.52241776e-03  9.83685216e-02
 -1.92273745e-01 -2.86681921e-01 -1.81145921e-01 -1.42092857e-01
 -1.56402871e-01]
[ 2.48357127e-05 -6.64077366e-01 -1.05038196e-01  8.68680487e-02
  6.59465043e-01 -1.32046331e-02 -1.64262292e-01 -3.14486914e-02
 -3.25667500e-02 -2.07791405e-01  6.86775157e-02 -1.46166466e-01
  8.40443168e-02]
[ 4.08249947e-01  8.98505761e-02  1.89684910e-02 -7.72828796e-02
  2.08573680e-01  8.43249451e-01  7.63363296e-02  5.14546012e-02
 -4.03809060e-02  9.22963272e-02 -1.84569819e-01  3.20494782e-02
 -1.03133718e-01]
[ 2.98102850e-05 -3.94586051e-01 -4.80694293e-02 -1.45031891e-01
 -1.96106215e-01 -5.27649678e-03  8.74300889e-01 -1.84163093e-02
 -1.12985730e-02 -2.53191833e-02 -7.81440888e-02 -9.29173682e-02
  4.71315200e-02]
[ 4.21886279e-05 -1.38212671e-01  1.83239173e-01  6.21868712e-02
 -5.64639906e-02 -2.59507288e-03 -2.19763493e-02  9.52601128e-01
  7.94509868e-02  9.09486097e-02  9.72522559e-02  4.32048346e-02
  8.01118002e-02]
[-2.28691285e-05  1.09573371e-01 -4.03981125e-01 -6.10750865e-02
  2.54621716e-02  1.27877394e-03  8.38665032e-03  8.84296364e-02
  8.27792924e-01 -2.60864068e-01 -1.54366945e-01 -1.24658140e-01
 -1.40935000e-01]
[-4.12897324e-05  7.49309180e-02 -7.17612148e-01  4.32541749e-01
 -2.84605491e-02 -6.78249955e-03  6.19789349e-02  1.39482629e-01
 -3.09090628e-01  2.79874797e-01  7.14258362e-02 -2.11713751e-01
 -2.10669690e-01]
[ 3.24789812e-05 -2.81105873e-02 -2.70269413e-01 -8.12318210e-01
 -1.32192123e-02  8.95153382e-03 -1.32764621e-01  6.35014997e-02
 -1.05458154e-01  1.54513196e-01  4.21411967e-01 -1.44222078e-01
 -1.04908515e-01]
[-3.81093158e-05 -1.86976867e-01 -3.45637434e-01 -1.02724394e-01
 -1.16534695e-01 -3.19855499e-03 -7.36484804e-02  5.50292629e-02
 -1.41451184e-01 -2.38987579e-01 -1.45490813e-01  8.42061829e-01
 -7.49668515e-02]
[-2.10072855e-05  2.79815285e-01 -2.75250887e-01 -1.16529329e-01
  1.21350199e-01  5.45083883e-03  4.85722982e-02  7.70746247e-02
 -1.20498864e-01 -1.16510027e-01 -1.65193107e-01 -5.37357510e-02
```

```
   8.66622794e-01]]

T (scores), shape = (4, 4)
[[-1.04178665e+04 -1.63720759e+00 -8.34437601e-01 -8.22533091e-02]
 [ 7.28187206e+03 -1.09921766e-01 -9.22281561e-01  6.01811262e-01]
 [ 8.87871328e+03 -2.46142037e-01 -6.52484189e-01 -6.50438111e-01]
 [-5.74271906e+03  2.45011972e+00 -6.64510459e-01 -9.33071740e-02]]
```

[15]:
```python
from itertools import combinations

for i, j in combinations(range(T.shape[1]), 2):
    d = T[:,i].dot(T[:,j])
    print(f"T{i} · T{j} = {d}")

print()
for i, j in combinations(range(P.shape[1]), 2):
    d = P[:,i].dot(P[:,j])
    print(f"P{i} · P{j} = {d}")
```

```
T0 · T1 = 9.094947017729282e-12
T0 · T2 = -2.2737367544323206e-12
T0 · T3 = -9.094947017729282e-13
T1 · T2 = 8.881784197001252e-16
T1 · T3 = 5.551115123125783e-17
T2 · T3 = -2.220446049250313e-16

P0 · P1 = 1.384013559693673e-16
P0 · P2 = 7.216395708389416e-17
P0 · P3 = -8.644985931640519e-17
P0 · P4 = 2.1707457081280605e-16
P0 · P5 = -3.1800102400132374e-16
P0 · P6 = 9.345517369284571e-17
P0 · P7 = 5.4187959304530263e-17
P0 · P8 = 2.16882686219331e-17
P0 · P9 = 2.0352378161700734e-16
P0 · P10 = -1.941238493990129e-16
P0 · P11 = 3.121419890920022e-17
P0 · P12 = -7.230134958062171e-17
P1 · P2 = -1.6943945182504454e-16
P1 · P3 = 3.4677675398328587e-18
P1 · P4 = 3.4365798506155196e-17
P1 · P5 = -7.178535808211429e-17
P1 · P6 = 7.946291903491407e-17
P1 · P7 = 5.2915442496445343e-17
P1 · P8 = -8.693792135521295e-17
P1 · P9 = -2.229740455894779e-16
P1 · P10 = -7.463562591587836e-17
P1 · P11 = -7.420054514741911e-17
```

P1 · P12 = -1.6774842002576154e-16
P2 · P3 = -5.264798818083943e-17
P2 · P4 = -4.476358919762213e-17
P2 · P5 = -5.337878344580473e-17
P2 · P6 = -3.607788478450433e-17
P2 · P7 = -3.6153541969312435e-18
P2 · P8 = 6.343514527216511e-17
P2 · P9 = -1.171332606214875e-16
P2 · P10 = -1.796046355541906e-17
P2 · P11 = 7.445941700604253e-17
P2 · P12 = 6.47440602198726e-17
P3 · P4 = -9.068985160277184e-17
P3 · P5 = 6.075185049435661e-17
P3 · P6 = -1.3516213501847547e-17
P3 · P7 = -4.8023812158227115e-17
P3 · P8 = 4.341408914543455e-17
P3 · P9 = 9.24963381819216e-17
P3 · P10 = 9.46857087121735e-17
P3 · P11 = -1.801412842585164e-18
P3 · P12 = 6.043846026337365e-17
P4 · P5 = -1.283212746986339e-16
P4 · P6 = 3.7653172725892016e-17
P4 · P7 = 6.884725347537703e-17
P4 · P8 = -5.807004046427742e-17
P4 · P9 = -1.0245201538544e-17
P4 · P10 = -1.4200096104820289e-16
P4 · P11 = 2.2784942186198993e-18
P4 · P12 = -8.820169359109847e-17
P5 · P6 = -5.602753301274504e-17
P5 · P7 = -2.644481957567093e-17
P5 · P8 = 1.5021166009683772e-17
P5 · P9 = -8.253296084796184e-17
P5 · P10 = 1.3134178117865412e-16
P5 · P11 = -1.7215581833550074e-17
P5 · P12 = 3.937200462670998e-17
P6 · P7 = 2.017834044567053e-17
P6 · P8 = -1.1958546474854027e-17
P6 · P9 = -6.137041311752852e-18
P6 · P10 = -5.87431147716743e-17
P6 · P11 = -1.3400254042492703e-17
P6 · P12 = -3.038739669016131e-17
P7 · P8 = -2.240028682526449e-17
P7 · P9 = 2.770258213986145e-17
P7 · P10 = -5.898927391719644e-17
P7 · P11 = -1.5965072638333783e-17
P7 · P12 = -2.3676637291834002e-17
P8 · P9 = -4.192528668136294e-17
P8 · P10 = 5.337171609308266e-18

```
P8 · P11 = 2.2260957243856615e-17
P8 · P12 = 2.609454506777097e-17
P9 · P10 = -9.343916425271724e-17
P9 · P11 = 6.615547931562843e-18
P9 · P12 = -6.53442784702703e-17
P10 · P11 = 1.6628303646904707e-17
P10 · P12 = 4.1058242877917034e-17
P11 · P12 = 3.0034668508669726e-17
```

[16]:
```python
for i in range(T.shape[1]):
    l = np.linalg.norm(T[:,i])
    print(f"||T{i}|| = {l}")

print()
for i in range(P.shape[1]):
    l = np.linalg.norm(P[:,i])
    print(f"||P{i}|| = {l}")
```

```
||T0|| = 16533.84328925055
||T1|| = 2.959088375039947
||T2|| = 1.5537693375924309
||T3|| = 0.8948286799484996

||P0|| = 1.0000000000000002
||P1|| = 1.0
||P2|| = 0.9999999999999999
||P3|| = 1.0
||P4|| = 1.0
||P5|| = 0.9999999999999999
||P6|| = 1.0
||P7|| = 1.0
||P8|| = 0.9999999999999999
||P9|| = 0.9999999999999999
||P10|| = 1.0
||P11|| = 1.0
||P12|| = 1.0
```

[17]:
```python
explained_variance = S/np.sum(S)
for i, ev in enumerate(explained_variance):
    print(f"Explained variance by PC-{i}: {100*ev:.5f}%")
```

```
Explained variance by PC-0: 99.96730%
Explained variance by PC-1: 0.01789%
Explained variance by PC-2: 0.00939%
Explained variance by PC-3: 0.00541%
```

All loadings and score vectors are pairwise orthogonal as they should be, and all the loadings are also unit length. This is what is expected from the PCA, and we get this even though we have

appended all different types of data.

We also see that almost all of the explained variance lies in the first PC. We also see that the magnitude of the first scores vector is significantly larger than the others. Perhaps it is that the scores vector explaining the largest variance is the one with largest norm?

assignment4_pca-cluster

April 18, 2021

1 Objective: Learn to do clustering and noise reduction in data using PCA

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
from numpy.linalg import svd
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

[1]:
```
(1797, 64)
```

1.1 PCA using SVD

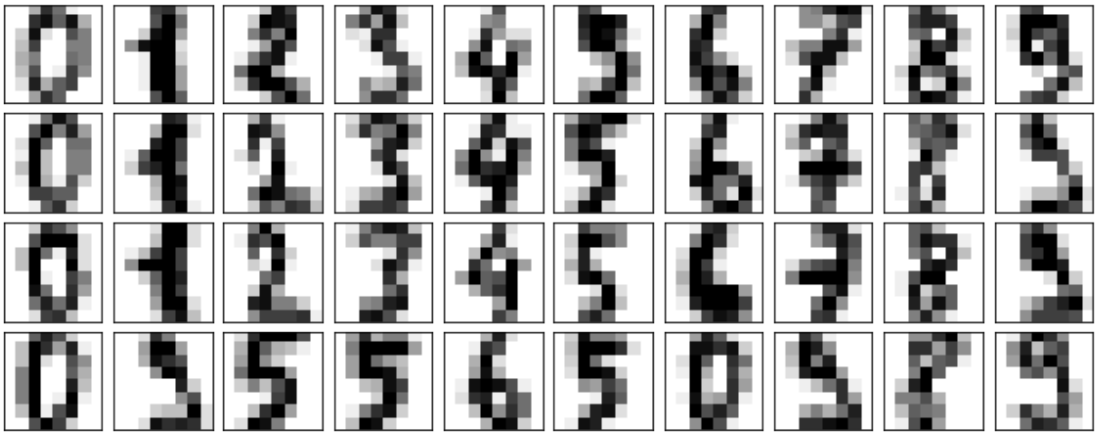
[2]:
```
def pca(X):
    U, S, PTrans = svd(X, full_matrices=False)
    Sigma = np.diag(S)
    T=np.dot(U,Sigma)
    P=PTrans.T
    return T, Sigma, P #Score, Variace, Loadings
```

[3]:
```
def plot_digits(data):
    fig, axes = plt.subplots(4, 10, figsize=(10, 4),
                             subplot_kw={'xticks':[], 'yticks':[]},
                             gridspec_kw=dict(hspace=0.1, wspace=0.1))
    for i, ax in enumerate(axes.flat):
        ax.imshow(data[i].reshape(8, 8),
                  cmap='binary', interpolation='nearest',
                  clim=(0, 16))
```

[4]:
```
# Find out the original dimension of the data
X = digits.data
y = digits.target
print("Shape of X",X.shape)
print("Shape of y", y.shape)
```

```
Shape of X (1797, 64)
Shape of y (1797,)
```

[5]:
```
#Visualize the original data
plot_digits(X)
```

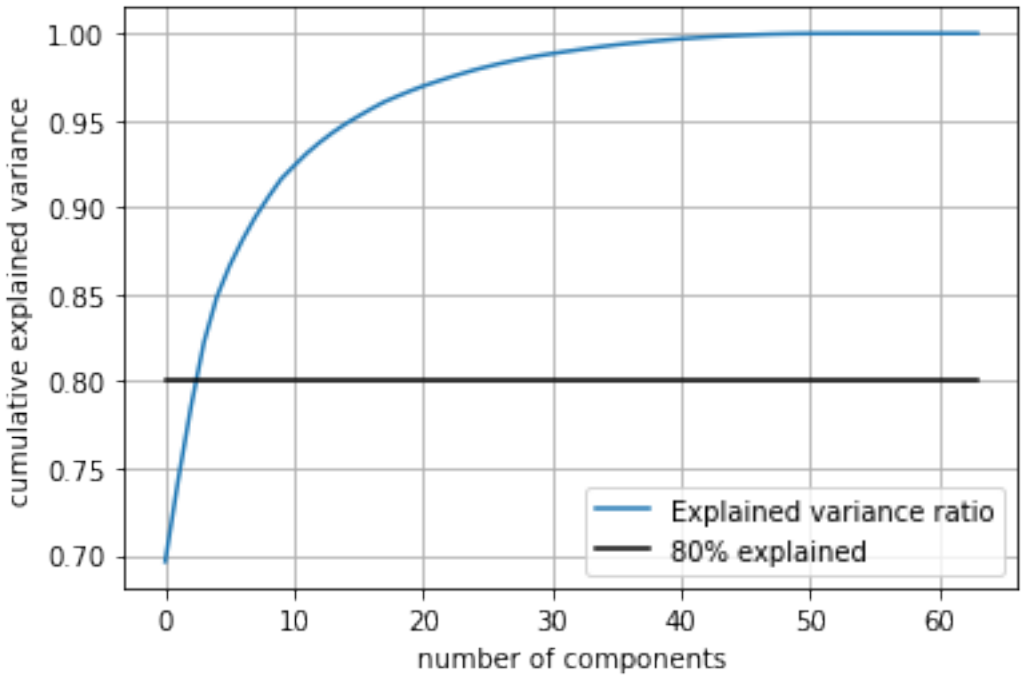


[6]:
```
# Mean-center the data
Xm = np.mean(X, axis=0)
#X = X - Xm
```

1.1.1 Task 1: Dimensionality reduction: Conduct PCA on the the matrix X to find out the dimension required to capture 80% of the variance

[7]:
```
T, Sigma, P = pca(X)

def plot_explained_variance(Sigma, variance_threshold=0.8):
    SS = np.diag(Sigma)
    explained_variance = (SS ** 2) / 4  # TODO: why dividing by 4?
    explained_variance_ratio = (explained_variance / explained_variance.sum())
    cumsum = np.cumsum(explained_variance_ratio)
    plt.plot(cumsum, label="Explained variance ratio")
    plt.plot(variance_threshold * np.ones(explained_variance_ratio.shape), "k",␣
 ↪label=f"{100*variance_threshold:.0f}% explained")
    plt.grid()
    plt.xlabel('number of components')
    plt.ylabel('cumulative explained variance')
    plt.legend()
    plt.show()

    for i, variance in enumerate(cumsum):
        if variance > variance_threshold:
```

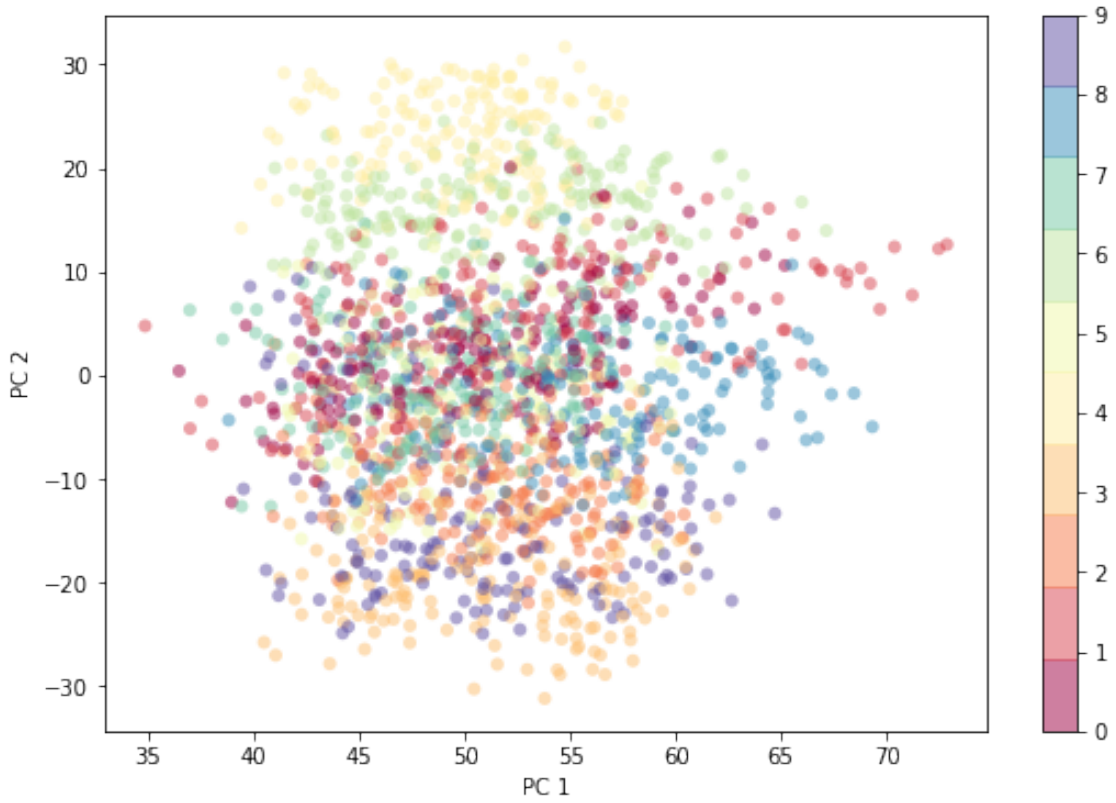
```
            break
    print(f"{100*variance_threshold:.0f}% explained variance after {i+1} PCs.␣
↪({100*variance:.2f}%)")

    return i+1


plot_explained_variance(Sigma);
```




```
80% explained variance after 4 PCs. (82.22%)
```

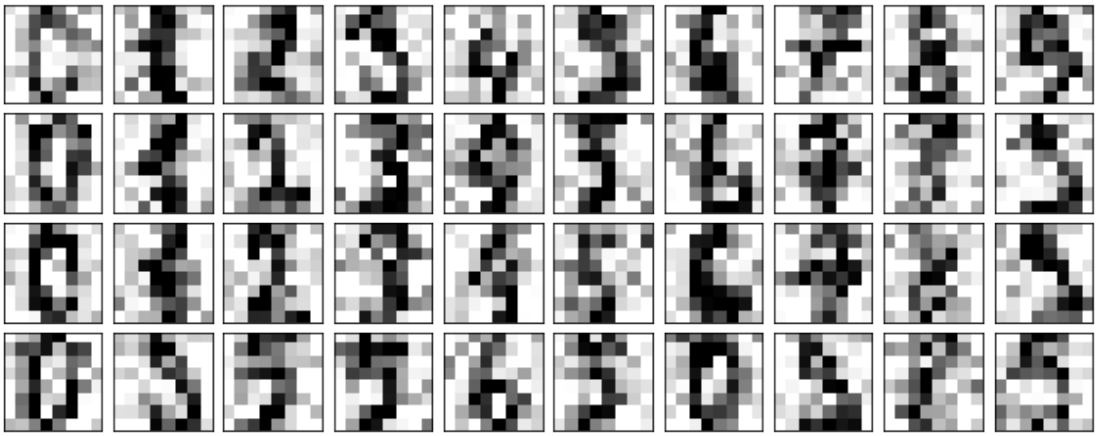
1.1.2 Task 2: Clustering: Project the original data matrix X on the first two PCs and draw the scalar plot

[8]:
```
t1 = T[:,0]
t2 = T[:,1]
plt.figure(figsize=(9,6))
plt.scatter(t1, t2,
            c=digits.target, edgecolor='none', alpha=0.5,
            cmap=plt.cm.get_cmap('Spectral', 10))
plt.xlabel("PC 1")
plt.ylabel("PC 2")
plt.colorbar()
plt.show()
```



1.1.3 Task 3: Denoising: Remove noise from the noisy data

[9]:
```
# Adding noise to the original data
X = digits.data
y = digits.target

np.random.seed(42)
noisy = np.random.normal(X, 4)
plot_digits(noisy)
```



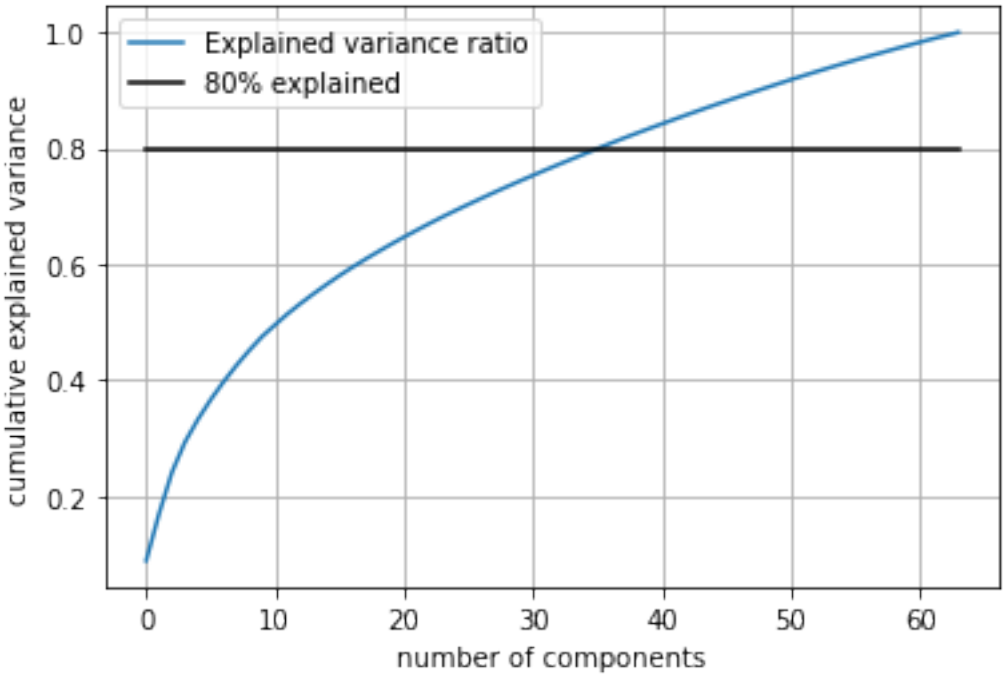
Tips:

- Decompose the noisy data using PCA
- Reconstruct the data using just a few dominant components.For eg. check the variance plot

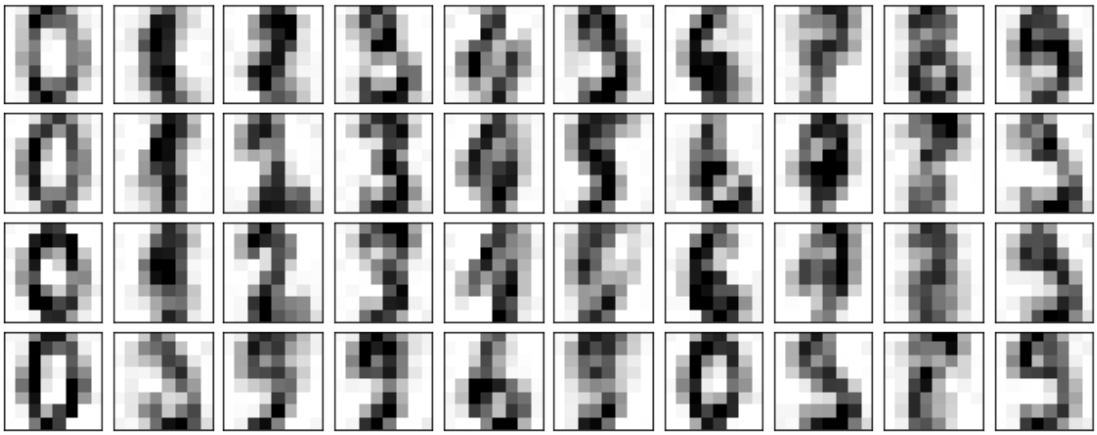
Since the nature of the noise is more or less similar across all the digits, they are not the fearues with enough variance to discriminate between the digits.

[10]:
```
# Mean-center noisy data
noisy_mean = np.mean(noisy, axis=0)
noisy = noisy - noisy_mean
```

[11]:
```
T, Sigma, P = pca(noisy)

pc_count = plot_explained_variance(Sigma, variance_threshold=0.8)

pc_count = 10
X_denoised = T[:,:pc_count] @ P[:,:pc_count].T
plot_digits(X_denoised + noisy_mean)
```




```
80% explained variance after 37 PCs. (80.83%)
```



Using the PC count from setting a threshold of e.g. 80% yielded results that still looked very noisy. Manually reducing the number of PCs gave the above results which looks much less noisy, but are still legible.

1.1.4 Task 4: Study the impact of normalization of the dataset before conducting PCA. Discuss if it is critical to normalize this particular data compared to the dataset in other notebooks

[12]:
```
X = digits.data
y = digits.target

# Calculate mean and std of each variable
Xm = np.mean(X, axis=0)
Xs = np.std(X, axis=0)

print(f"Data: min: {X.min():.2f}, max: {X.max():.2f}, average: {X.mean():.2f}")
print(f"Mean: min: {Xm.min():.2f}, max: {Xm.max():.2f}, average: {Xm.mean():.
↪2f}")
print(f"Std:  min: {Xs.min():.2f}, max: {Xs.max():.2f}, average: {Xs.mean():.
↪2f}")

# Standard deviation of 0 does not work, set it to 1 to not get any errors.
# TODO: will this give correct results?
Xs[Xs == 0] = 1
```

```
Data: min: 0.00, max: 16.00, average: 4.88
Mean: min: 0.00, max: 12.09, average: 4.88
Std:  min: 0.00, max: 6.54, average: 3.68
```

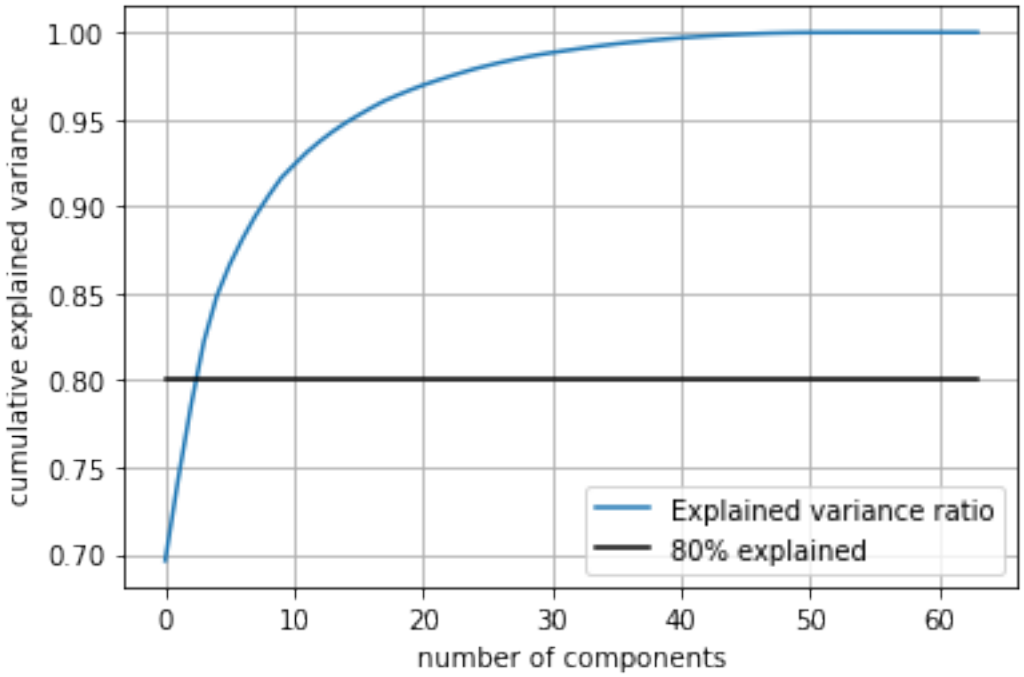
Comparing the order of magnitude for the mean and the data, it is possible that mean-centering could give better results. It is also likely that normalizing using the standard deviation would yield better results. Let us compare:

1. No augmentations
2. Only mean centering
3. Only normalizing
4. Mean centering and normalizing

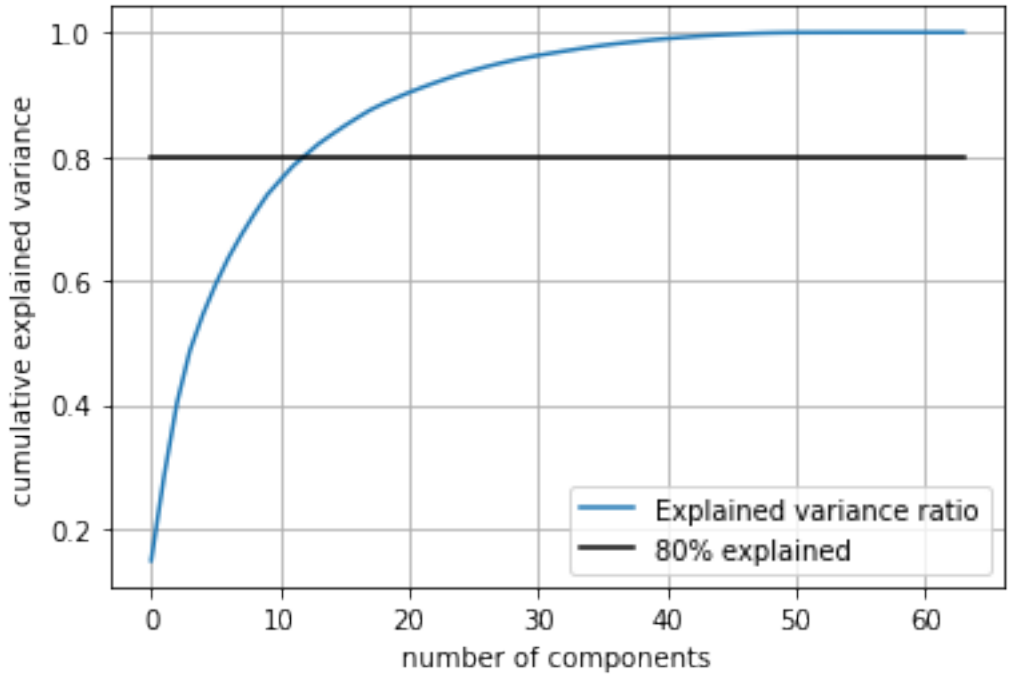
[13]:
```
print("--- 1. No augmentation ---")
T1, Sigma1, P1 = pca(X)
pc_count1 = plot_explained_variance(Sigma1)
print()

print("--- 2. Only mean centering ---")
T2, Sigma2, P2 = pca(X - Xm)
pc_count2 = plot_explained_variance(Sigma2)
print()

print("--- 3. Only normalizing ---")
T3, Sigma3, P3 = pca(X / Xs)
pc_count3 = plot_explained_variance(Sigma3)
print()
```

```
print("--- 4. Mean centering and normalizing ---")
T4, Sigma4, P4 = pca((X - Xm) / Xs)
pc_count4 = plot_explained_variance(Sigma4)
print()
```

```
--- 1. No augmentation ---
```

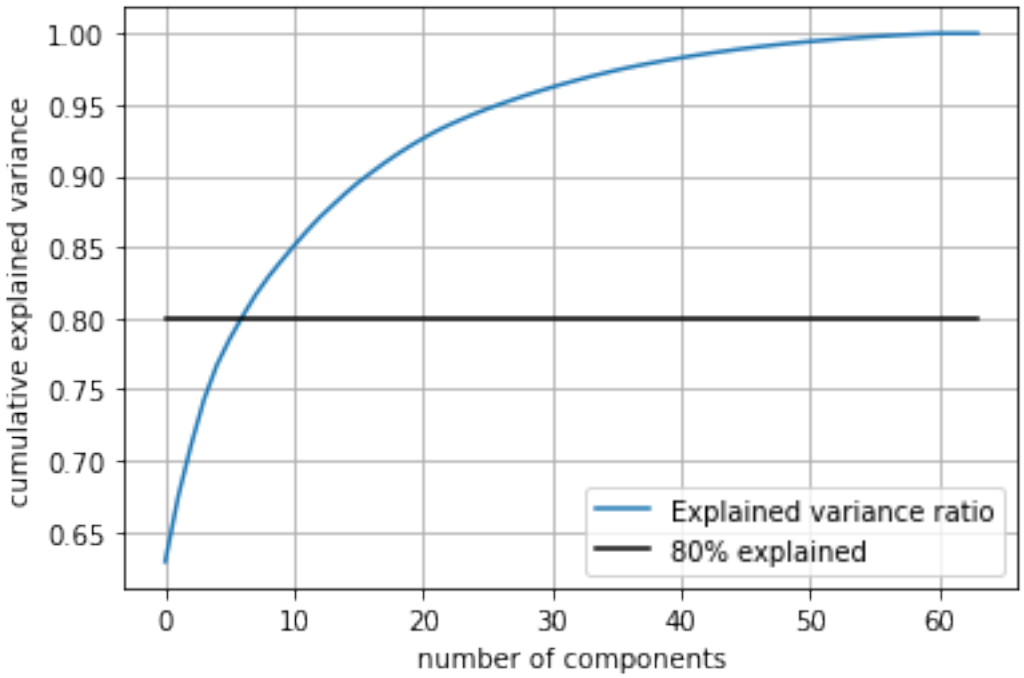



```
80% explained variance after 4 PCs. (82.22%)

--- 2. Only mean centering ---
```

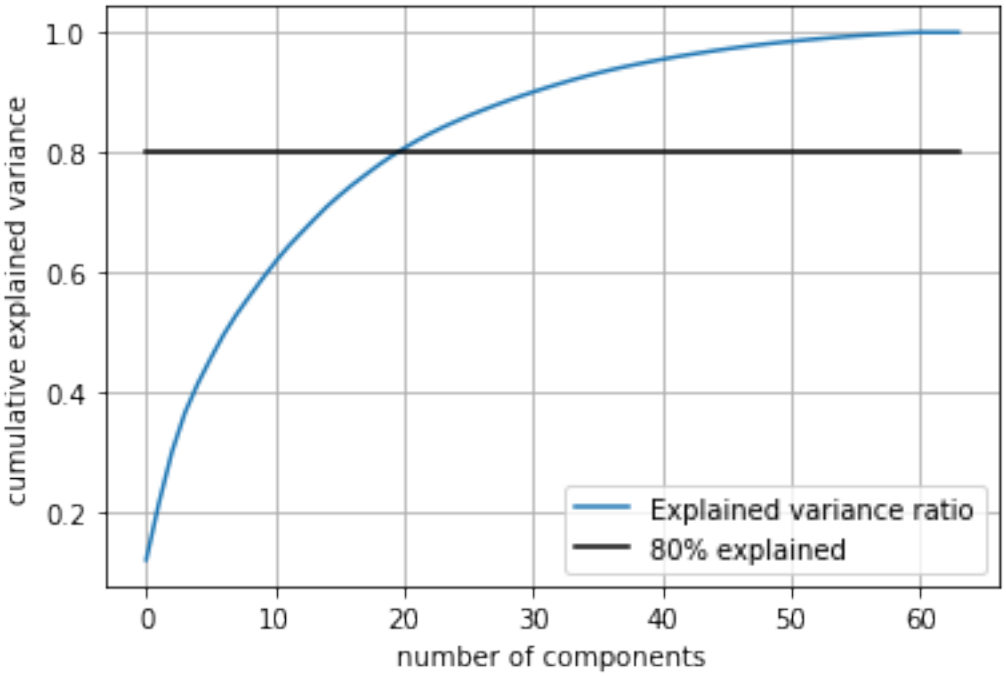


80% explained variance after 13 PCs. (80.29%)

--- 3. Only normalizing ---



```
80% explained variance after 7 PCs. (80.12%)

--- 4. Mean centering and normalizing ---
```




```
80% explained variance after 21 PCs. (80.66%)
```

Is more or fewer PCs good? I think fewer, but unsure.

Normalizing does not make much sense since all variables are pixel values and are of the same order of magnitude.

The numbers above makes sense as when you normalize using std the distributions become "rounder" in all dimensions, so the distribution of variance should be more even.

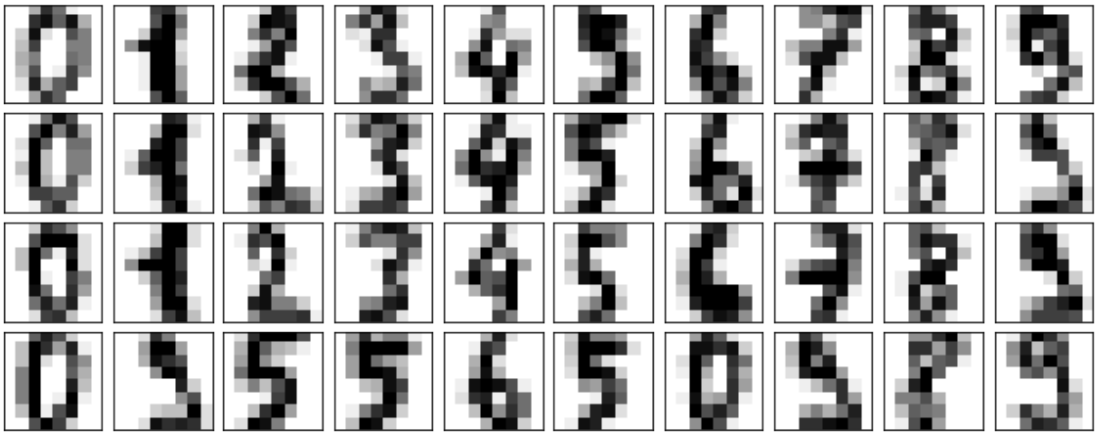
2 All the above excercise can be done using the SKLEAR library as follows

[14]:
```
from sklearn.decomposition import PCA
X=digits.data
y=digits.target
```

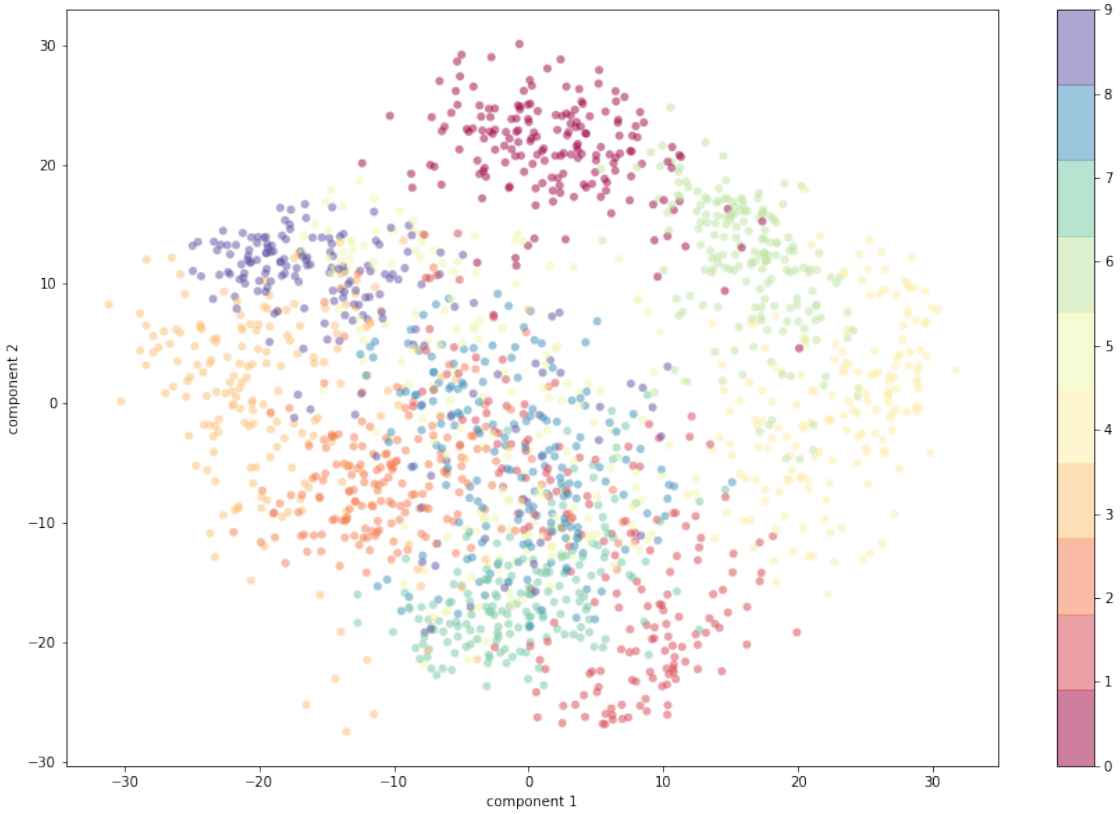
[15]:
```
pca = PCA(2)  # project from 64 to 2 dimensions
projected = pca.fit_transform(digits.data)
```

```
print(digits.data.shape)
print(projected.shape)
plot_digits(digits.data)
```

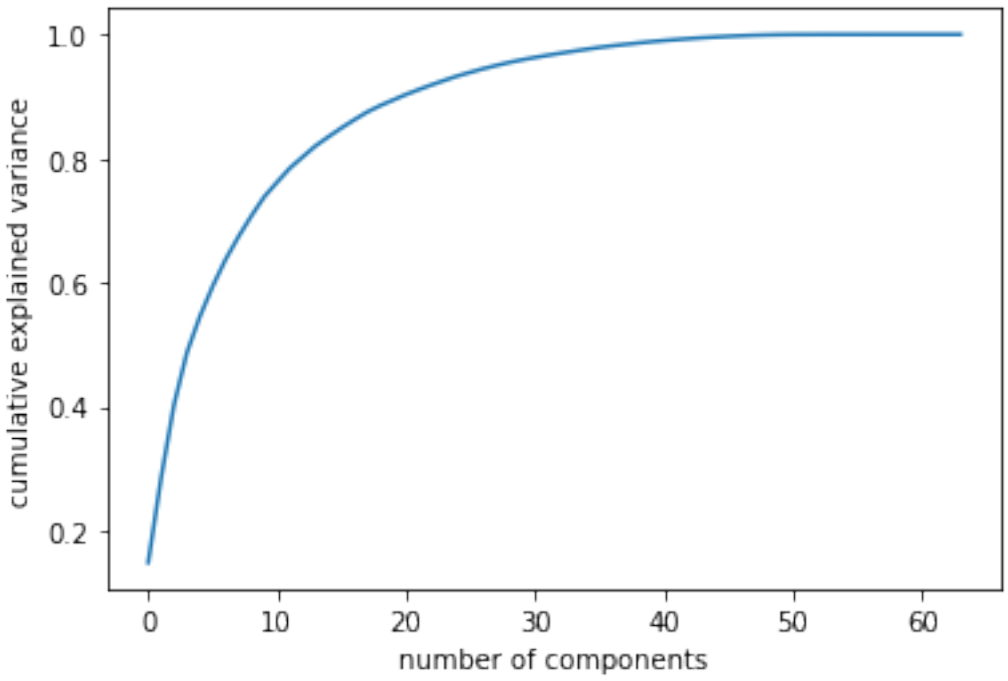
```
(1797, 64)
(1797, 2)
```



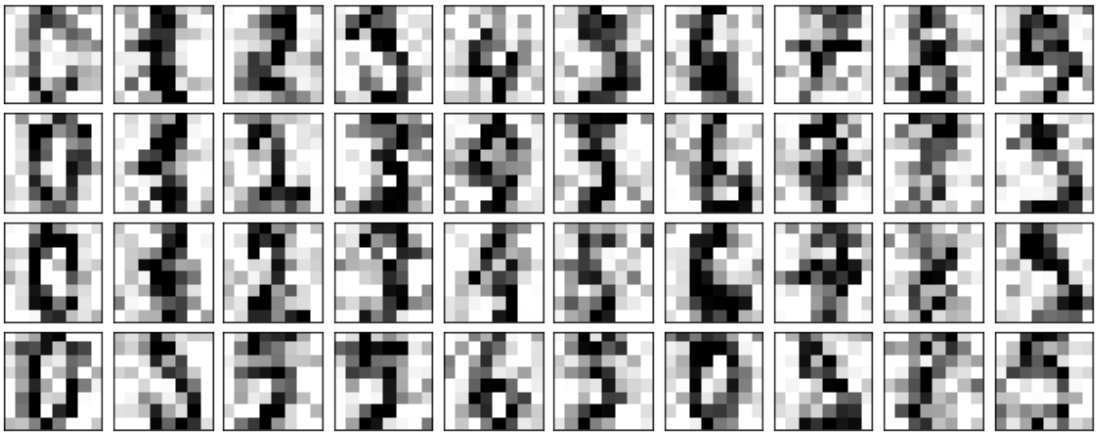
[16]:
```
plt.figure(figsize=(15,10))
plt.scatter(projected[:, 0], projected[:, 1],
            c=digits.target, edgecolor='none', alpha=0.5,
            cmap=plt.cm.get_cmap('Spectral', 10))
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.colorbar();
```



[17]:
```
pca = PCA().fit(digits.data)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```

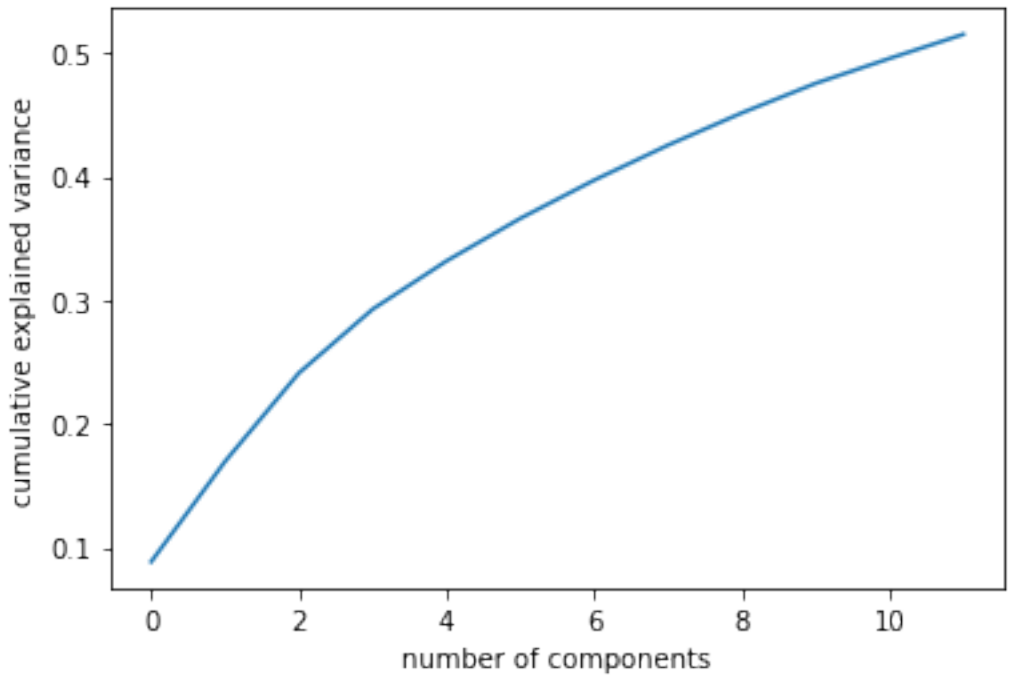


[18]:
```
np.random.seed(42)
noisy = np.random.normal(digits.data, 4)
plot_digits(noisy)
```

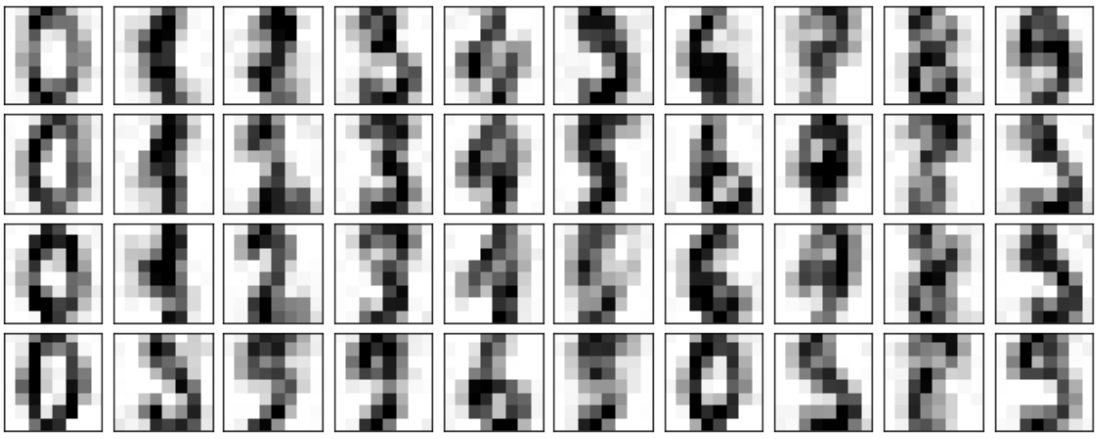


[19]:
```
pca = PCA(0.50).fit(noisy) # 50% of the variance amounts to 12 principal␣
↪components.
pca.n_components_
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
```

```
plt.ylabel('cumulative explained variance');
```



[20]:
```
components = pca.transform(noisy)
filtered = pca.inverse_transform(components)
plot_digits(filtered)
```



assignment4_pca-scaling

April 18, 2021

1 Objective: To understand the importance of scaling on PCA

[23]:
```
from sklearn.decomposition import PCA
from sklearn import preprocessing
from sklearn import metrics
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine

import warnings
#warnings.filterwarnings(action="error", category=np.ComplexWarning)

np.seterr(all="raise")
```

[23]:
```
{'divide': 'raise', 'over': 'raise', 'under': 'raise', 'invalid': 'raise'}
```

2 Task 0: Write the function to compute the pca using Eigenvector approach

[33]:
```
from numpy.linalg import svd
def pca(X):
    XTX = X.T @ X
    Sigma2, P = np.linalg.eig(XTX)
    assert np.all(Sigma2 >= 0)
    order = np.argsort(np.abs(Sigma2))[::-1]
    P = P[:,order]
    Sigma = np.sqrt(Sigma2[order])

    XXT = X @ X.T
    Sigma_, U = np.linalg.eig(XXT)
    r = len(order)
    T = U[:,:r][:,order] @ np.diag(Sigma)

    return T, Sigma, P #Score, Variace, Loadings

def pca_svd(X):
```

```
    U, S, PTrans = svd(X, full_matrices=False)
    Sigma = np.diag(S)
    T=np.dot(U,Sigma)
    P=PTrans.T
    return T, Sigma, P #Score, Variace, Loadings

features, target = load_wine(return_X_y=True)
X = features
T, Sigma, P = pca(X)

T_svd, Sigma_svd, P_svd = pca_svd(X)

pca = pca_svd
```

```
(178, 178)
(178,)
```

[ ]:
```
X: m x n
XTX: n x n
XXT: m x m

P: n x n
U: m x m
```

Note: My own calculations give complex numbers with complex part 0 (possibly some numerical errors with giving -0). Also note that my calculations of the eigenvalues and eigenvectors give opposite sign of that coming from SVD. This is no problem since the eigenvectors are defined up to a scaling factor with arbitrary sign.

[25]:
```
features, target = load_wine(return_X_y=True)
X = features
y = target
```

3 Three different ways of scaling

- Scaling by removing the mean and divising by the standard deviation so the standard deviation is 1 for each features (any assumptions about gaussianity?)

  ```
  standard_scaling = preprocessing.StandardScaler()

  X_standard = standard_scaling.fit_transform(X)
  ```

- Scaling to minimum and maximum values of each feature so all samples for a feature lies between 0 and 1

  ```
  minmax_scaling = preprocessing.MinMaxScaler()

  X_minmax = minmax_scaling.fit_transform(X)
  ```

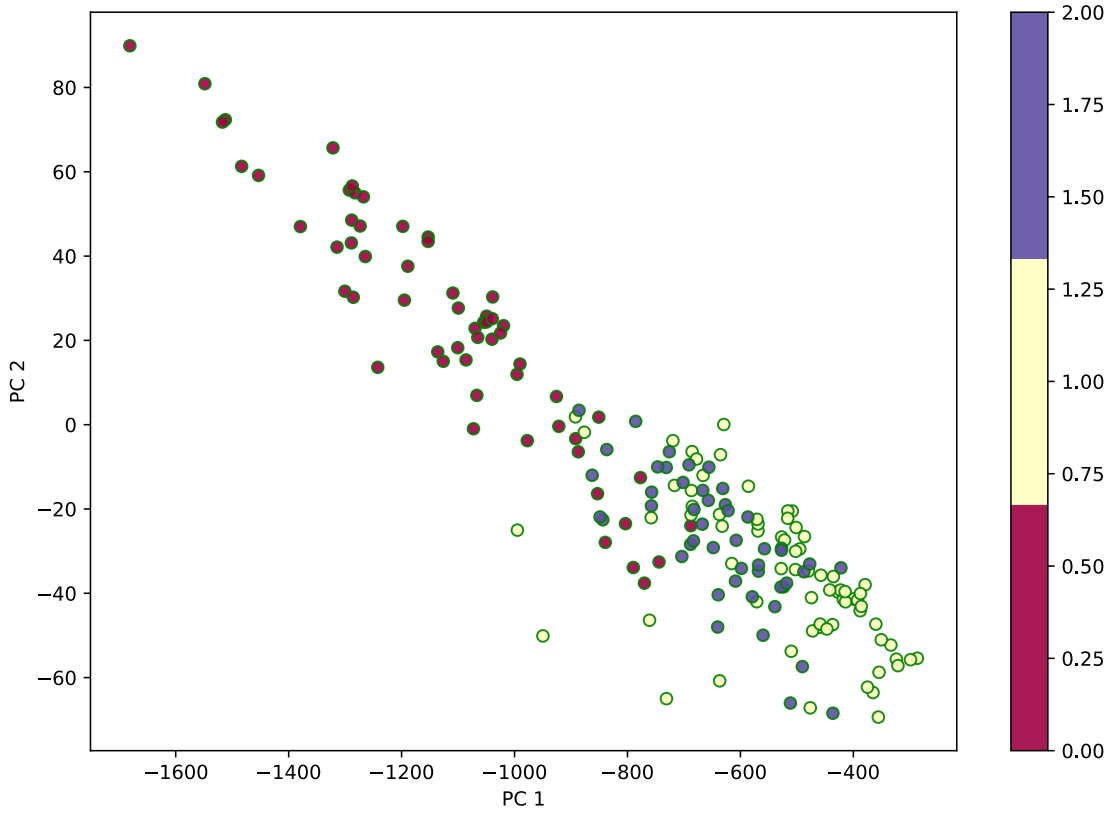
- Scaling by diving by the absolute value of the maximum value of each feature

```
max_abs_scaler = preprocessing.MaxAbsScaler()

X_maxabs = max_abs_scaler.fit_transform(X)
```

3.1 Task 1: Create the scores plot without any scaling

[26]:
```
def draw_scores_plot(T):
    global y

    plt.figure(figsize=(10,7))
    plt.scatter(T[:, 0], T[:, 1],
                c=y, edgecolor='green', alpha=0.9,
                cmap=plt.cm.get_cmap('Spectral', 3))
    plt.xlabel("PC 1")
    plt.ylabel("PC 2")
    plt.colorbar()
    plt.show()
```

[27]:
```
T, S, P = pca(X)

draw_scores_plot(T)
```

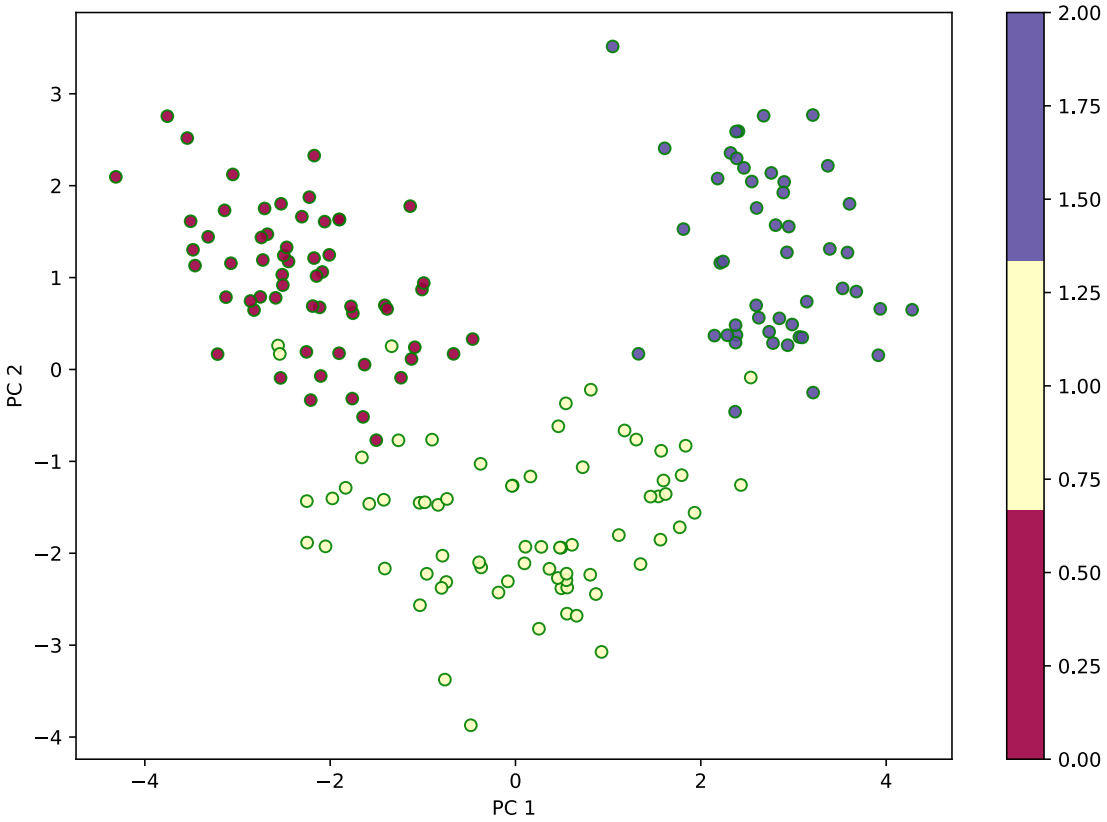


Notice that PC1 has order of magnitude 1000 while PC2 has order of magnitude 10. We also see that there are no good clusters, they are more or less all in each others way, except for some of the red data points.

3.2 Task 2: Create the scores plot with standard scaling

Standard scaling mean centers and scales to unit variance. Docs

[28]:
```
standard_scaler = preprocessing.StandardScaler()
X_standard = standard_scaler.fit_transform(X)
T, S, P = pca(X_standard)

draw_scores_plot(T)
```

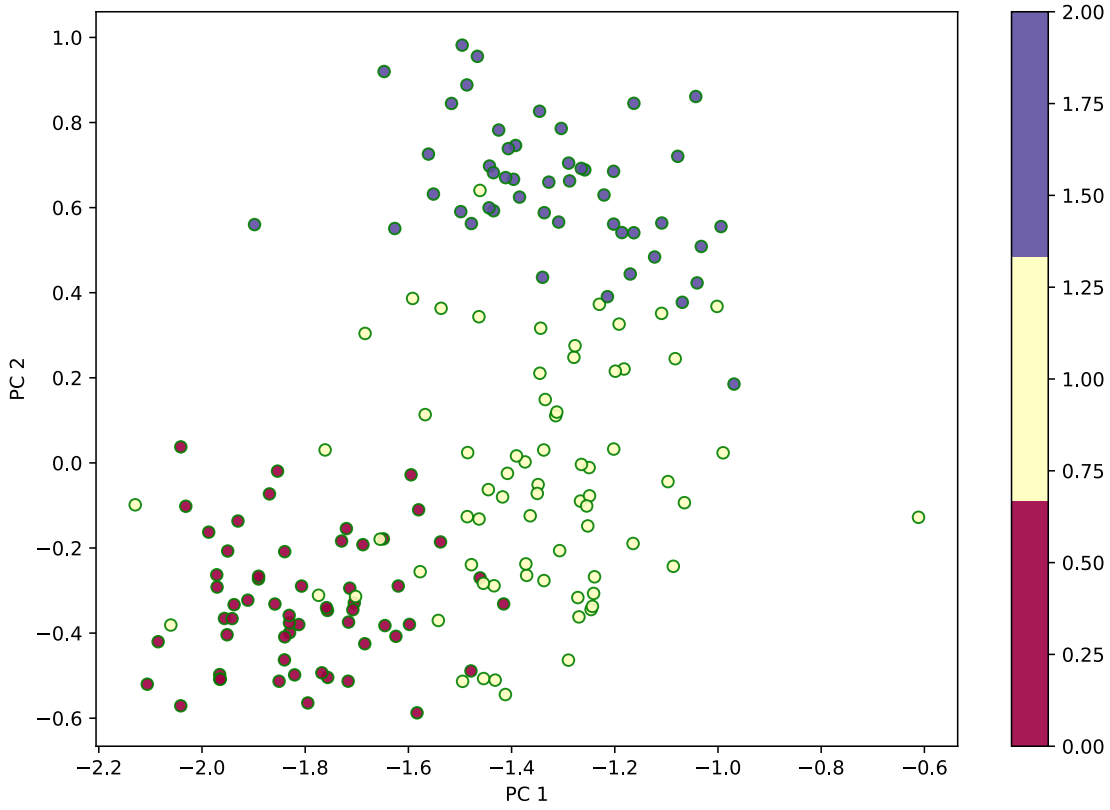


We see much better separation now, and both axis are of the same order of magnitude.

3.3 Task 3: Create the scores plot with min max scaling

Min max scaling transforms all samples in a feature lies between 0 and 1. Docs

[29]:
```
minmax_scaler = preprocessing.MinMaxScaler()
X_minmax = minmax_scaler.fit_transform(X)
T, S, P = pca(X_minmax)

draw_scores_plot(T)
```



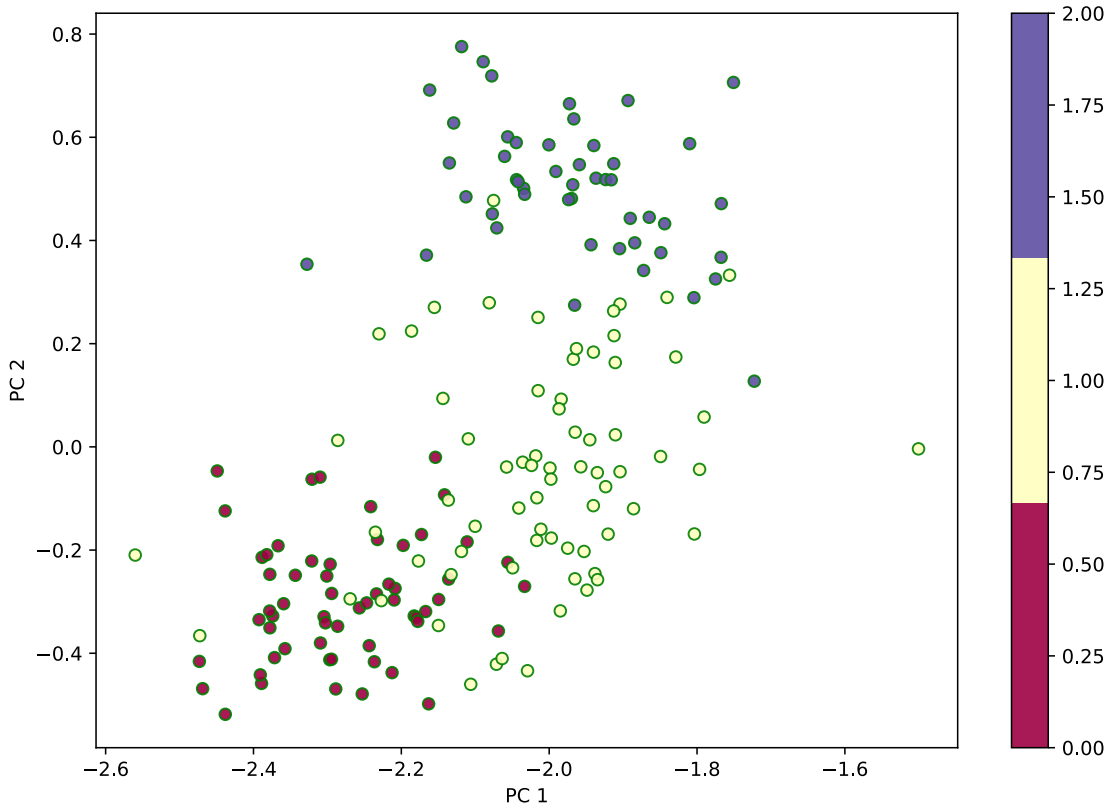
Good separation, same order of magnitude on the axes.

Note: Minmax scaling only gives values between 0 and 1 in the X matrix, not in the scores.

3.4 Task 4: Create the scores plot with max abs scaling scaling

Max abs scaling transforms the features so that the maximum value has unit size. Docs

[30]:
```
maxabs_scaler = preprocessing.MaxAbsScaler()
X_maxabs = maxabs_scaler.fit_transform(X)
T, S, P = pca(X_maxabs)

draw_scores_plot(T)
```



3.5 Comments on scaling

When standard scaling notice that the classes become more separated than with no or the other scalings. Standard scaling is also the only scaling with mean centering.