Introduction-Numpy

November 26, 2020

1 Introduction to Numpy, Scipy, SciKit

In this tutorial we will look into the Numpy library: http://www.numpy.org/

Numpy is a very important library for numerical computations and matrix manipulation. It has a lot of the functionality of Matlab, and some of the functionality of Pandas

We will also use the Scipy library for scientific computation: http://docs.scipy.org/doc/numpy/reference/index.html

[56]:
```
import numpy as np
import scipy as sp
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns
%matplotlib inline
```

[57]:
```
import scipy.sparse as sp_sparse
import scipy.spatial.distance as sp_dist
import sklearn as sk
import sklearn.datasets as sk_data
import sklearn.metrics as metrics
from sklearn import preprocessing
import scipy.sparse.linalg as linalg
```

1.0.1 Why Numpy?

[58]:
```
import time

def trad_version():
    t1 = time.time()
    X = range(10000000)
    Y = range(10000000)
    Z = [x+y for x,y in zip(X,Y)]
    return time.time() - t1
```

```
def numpy_version():
    t1 = time.time()
    X = np.arange(10000000)
    Y = np.arange(10000000)
    Z = X + Y
    return time.time() - t1


traditional_time = trad_version()
numpy_time = numpy_version()
print ("Traditional time = "+ str(traditional_time))
print ("Numpy time       = "+ str(numpy_time))
```

```
Traditional time = 1.4342577457427979
Numpy time       = 0.060837507247924805
```

2 Arrays

2.0.1 Creating Arrays

In Numpy data is organized into arrays. There are many different ways to create a numpy array.

For the following we will use the random library of Numpy: http://docs.scipy.org/doc/numpy-1.10.0/reference/routines.random.html

Creating arrays from lists

[59]:
```
#1-dimensional arrays
x = np.array([2,5,18,14,4])
print ("\n Deterministic 1-dimensional array \n")
print (x)

#2-dimensional arrays
x = np.array([[2,5,18,14,4], [12,15,1,2,8]])
print ("\n Deterministic 2-dimensional array \n")
print (x)
```

```
 Deterministic 1-dimensional array

[ 2  5 18 14  4]

 Deterministic 2-dimensional array

[[ 2  5 18 14  4]
 [12 15  1  2  8]]
```

We can also create Numpy arrays from Pandas DataFrames

[60]:
```
d = {'A':[1., 2., 3., 4.],
     'B':[4., 3., 2., 1.]}
df = pd.DataFrame(d)
x = np.array(df)
print(x)
```

```
[[1. 4.]
 [2. 3.]
 [3. 2.]
 [4. 1.]]
```

Creating random arrays

[61]:
```
#1-dimensional arrays
x = np.random.rand(5)
print ("\n Random 1-dimensional array \n")
print (x)

#2-dimensional arrays

x = np.random.rand(5,5)
print ("\n Random 5x5 2-dimensional array \n")
print (x)

x = np.random.randint(10,size=(2,3))
print("\n Random 2x3 array with integers")
print(x)
```

```
 Random 1-dimensional array

[0.23818734 0.89756757 0.63236682 0.42353155 0.23689535]

 Random 5x5 2-dimensional array

[[0.90589428 0.19548478 0.06661911 0.8635276  0.10017688]
 [0.70481576 0.74408421 0.27654818 0.86778097 0.99395313]
 [0.1005293  0.1286749  0.12554055 0.20583548 0.08123416]
 [0.82224742 0.82900872 0.41467888 0.45541027 0.08015081]
 [0.85616401 0.82030885 0.06022059 0.9745874  0.11484953]]

 Random 2x3 array with integers
[[0 4 1]
 [6 1 4]]
```

Transpose and get array dimensions

[62]:
```
print("\n Matrix Dimensions \n")
print(x.shape)
print ("\n Transpose of the matrix \n")
print (x.T)
print (x.T.shape)
```

```
 Matrix Dimensions

(2, 3)

 Transpose of the matrix

[[0 6]
 [4 1]
 [1 4]]
(3, 2)
```

Special Arrays

[63]:
```
x = np.zeros((4,4))
print ("\n 4x4 array with zeros \n")
print(x)

x = np.ones((4,4))
print ("\n 4x4 array with ones \n")
print (x)

x = np.eye(4)
print ("\n Identity matrix of size 4\n")
print(x)

x = np.diag([1,2,3])
print ("\n Diagonal matrix\n")
print(x)
```

```
 4x4 array with zeros

[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

 4x4 array with ones

[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
```

```
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

 Identity matrix of size 4

[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]

 Diagonal matrix

[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

2.0.2 Operations on arrays.

These are very similar to what we did with Pandas

[127]:
```
x = np.random.randint(10, size = (2,4))
print (x)
print('\n mean value of all elements')
print (np.mean(x))
print('\n vector of mean values for columns')
print (np.mean(x,0)) #0 signifies the dimension meaning columns
print('\n vector of mean values for rows')
print (np.mean(x,1)) #1 signifies the dimension meaning rows
```

```
[[3 0 7 8]
 [8 4 2 8]]

 mean value of all elements
5.0

 vector of mean values for columns
[5.5 2.  4.5 8. ]

 vector of mean values for rows
[4.5 5.5]
```

[128]:
```
print('\n standard deviation of all elements')
print (np.std(x))
print('\n vector of std values for rows')
print (np.std(x,1)) #1 signifies the dimension meaning rows
print('\n median value of all elements')
print (np.median(x))
print('\n vector of median values for rows')
```

```
print (np.median(x,1))
print('\n sum of all elements')
print (np.sum(x))
print('\n vector of column sums')
print (np.sum(x,0))
print('\n product of all elements')
print (np.prod(x))
print('\n vector of row products')
print (np.prod(x,1))
```

```
 standard deviation of all elements
2.958039891549808

 vector of std values for rows
[3.20156212 2.59807621]

 median value of all elements
5.5

 vector of median values for rows
[5. 6.]

 sum of all elements
40

 vector of column sums
[11  4  9 16]

 product of all elements
0

 vector of row products
[  0 512]
```

2.0.3 Manipulating arrays

Accessing and Slicing

[72]:
```
x = np.random.rand(4,3)
print(x)
print("\n element\n")
print(x[1,2])
print("\n row zero \n")
print(x[0,:])
print("\n column 2 \n")
print(x[:,2])
print("\n submatrix \n")
```

```
print(x[1:3,0:2])
print("\n entries > 0.5 \n")
print(x[x>0.5])
```

```
[[0.77661791 0.25334135 0.44022965]
 [0.44183799 0.77399097 0.25294422]
 [0.28345437 0.75754562 0.24618864]
 [0.78535173 0.34794355 0.06980467]]

 element

0.25294422169492525

 row zero

[0.77661791 0.25334135 0.44022965]

 column 2

[0.44022965 0.25294422 0.24618864 0.06980467]

 submatrix

[[0.44183799 0.77399097]
 [0.28345437 0.75754562]]

 entries > 0.5

[0.77661791 0.77399097 0.75754562 0.78535173]
```

Changing entries

[129]:
```
x = np.random.rand(4,3)
print(x)

x[1,2] = -5 #change an entry
x[0:2,:] += 1 #change a set of rows
x[2:4,1:3] = 0.5 #change a block
print(x)

print('\n Set entries > 0.5 to zero')
x[x>0.5] = 0
print(x)
```

```
[[0.95199099 0.84173482 0.32252688]
 [0.83841305 0.88070548 0.45831285]
 [0.25807667 0.11751366 0.50504308]
 [0.21017    0.76246724 0.7135389 ]]
```

```
[[ 1.95199099  1.84173482  1.32252688]
 [ 1.83841305  1.88070548 -4.        ]
 [ 0.25807667  0.5         0.5       ]
 [ 0.21017     0.5         0.5       ]]

 Set entries > 0.5 to zero
[[ 0.          0.          0.        ]
 [ 0.          0.         -4.        ]
 [ 0.25807667  0.5         0.5       ]
 [ 0.21017     0.5         0.5       ]]
```

[130]:
```
print('\n Diagonal \n')
x = np.random.rand(4,4)
print(x)
print('\n Read Diagonal \n')
print(x.diagonal())
print('\n Fill Diagonal with 1s \n')
np.fill_diagonal(x,1)
print(x)
print('\n Fill Diagonal with vector \n')
x[np.diag_indices_from(x)] = [1,2,3,4]
print(x)
```

```
 Diagonal

[[0.64781232 0.95392589 0.79933131 0.53432797]
 [0.03261657 0.25841772 0.3285714  0.14270102]
 [0.32608876 0.72774296 0.62959577 0.87942258]
 [0.08192375 0.19954746 0.30350918 0.06990768]]

 Read Diagonal

[0.64781232 0.25841772 0.62959577 0.06990768]

 Fill Diagonal with 1s

[[1.         0.95392589 0.79933131 0.53432797]
 [0.03261657 1.         0.3285714  0.14270102]
 [0.32608876 0.72774296 1.         0.87942258]
 [0.08192375 0.19954746 0.30350918 1.        ]]

 Fill Diagonal with vector

[[1.         0.95392589 0.79933131 0.53432797]
 [0.03261657 2.         0.3285714  0.14270102]
 [0.32608876 0.72774296 3.         0.87942258]
 [0.08192375 0.19954746 0.30350918 4.        ]]
```

2.0.4 Quiz

We want to create a dataset of 10 users and 5 items, where each user i has selects an item j with probability 0.3.

How can we do this with matrix operations?

[78]:
```
D = np.random.rand(10,5)
print(D)
D[D <= 0.3] = 1
D[D != 1] = 0
D
```

```
[[0.79578463 0.4337277  0.2125876  0.63199767 0.56537184]
 [0.74221258 0.26979647 0.21679366 0.79863161 0.48003212]
 [0.05831527 0.37878698 0.31203118 0.53469186 0.04043251]
 [0.93659677 0.9269729  0.4075207  0.20622661 0.98978848]
 [0.27186714 0.20639595 0.65672342 0.81012499 0.38410343]
 [0.57687903 0.67319821 0.78508931 0.70485438 0.63244049]
 [0.33973492 0.03815656 0.17120652 0.07160217 0.22028054]
 [0.32591091 0.66164666 0.94444681 0.1655961  0.31215205]
 [0.41527979 0.02526784 0.00150974 0.15672606 0.9645682 ]
 [0.69831531 0.44613275 0.50443458 0.47337582 0.95979451]]
```

[78]:
```
array([[0., 0., 1., 0., 0.],
       [0., 1., 1., 0., 0.],
       [1., 0., 0., 0., 1.],
       [0., 0., 0., 1., 0.],
       [1., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 1., 1., 1., 1.],
       [0., 0., 0., 1., 0.],
       [0., 1., 1., 1., 0.],
       [0., 0., 0., 0., 0.]])
```

2.0.5 Operations with Arrays

Multiplication and addition with scalar

[79]:
```
x = np.random.rand(4,3)
print(x)

#multiplication and addition with scalar value
print("\n Matrix 2x+1 \n")
print(2*x+1)
```

```
[[0.26655923 0.4976653  0.04086411]
 [0.37654774 0.72029154 0.19286209]
 [0.94842235 0.45388593 0.331548  ]
```

```
 [0.93141818 0.46236218 0.67553373]]

 Matrix 2x+1

[[1.53311846 1.99533059 1.08172822]
 [1.75309547 2.44058308 1.38572418]
 [2.8968447  1.90777185 1.66309599]
 [2.86283635 1.92472436 2.35106747]]
```

Vector-vector dot product

There are three ways to get the dot product of two vectors:

Using the method .dot of an array

Using the method dot of the numpy library

Using the '@' operator

[131]:
```
y = np.array([2,-1,3])
z = np.array([-1,2,2])
print('\n y:',y)
print(' z:',z)
print('\n vector-vector dot product')
print(y.dot(z))
print(np.dot(y,z))
print(y@z)
```

```
 y: [ 2 -1  3]
 z: [-1  2  2]

 vector-vector dot product
2
2
2
```

External product

The external product between two vectors x,y of size (n,1) and (m,1) results in a matrix M of size (n,m) with entries M(i,j) = x(i)*y(j)

[132]:
```
print('\n y:',y)
print(' z:',z)
print('\n vector-vector external product')
print(np.outer(y,z))
```

```
 vector-vector external product
[[-2  4  4]
 [ 1 -2 -2]
 [-3  6  6]]
```

Element-wise operations

[134]:
```
print('\n y:',y)
print(' z:',z)
print('\n element-wise addition')
print(y+z)
print('\n element-wise product')
print(y*z)
print('\n element-wise division')
print(y/z)
```

```
 y: [ 2 -1  3]
 z: [-1  2  2]

 element-wise addition
[1 1 5]

 element-wise product
[-2 -2  6]

 element-wise division
[-2.  -0.5  1.5]
```

Matrix-Vector multiplication

Again we can do the multiplication either using the dot method or the '@' operator

[139]:
```
X = np.random.randint(10, size = (4,3))
print('Matrix X:\n',X)
y = np.array([1,0,0])
print("\n Matrix-vector right multiplication with",y,"\n")
print(X.dot(y))
print(np.dot(X,y))
print(X@y)
y = np.array([1,0,1,0])
print("\n Matrix-vector left multiplication with",y,"\n")
print(y.dot(X))
print(np.dot(y,X))
print(y@X)
```

```
Matrix X:
 [[0 6 2]
 [4 1 7]
 [2 9 0]
 [0 5 2]]

 Matrix-vector right multiplication with [1 0 0]
```

```
[0 4 2 0]
[0 4 2 0]
[0 4 2 0]

 Matrix-vector left multiplication with [1 0 1 0]

[ 2 15  2]
[ 2 15  2]
[ 2 15  2]
```

Matrix-Matrix multiplication

Same for the matrix-matrix operation

[142]:
```
Y = np.random.randint(10, size=(3,2))
print("\n Matrix-matrix multiplication\n")
print('Matrix X:\n',X)
print('Matrix Y:\n',Y)
print('Product:\n',X.dot(Y))
print('Product:\n',X@Y)
```

```
 Matrix-matrix multiplication

Matrix X:
 [[0 6 2]
 [4 1 7]
 [2 9 0]
 [0 5 2]]
Matrix Y:
 [[8 8]
 [3 6]
 [6 6]]
Product:
 [[30 48]
 [77 80]
 [43 70]
 [27 42]]
Product:
 [[30 48]
 [77 80]
 [43 70]
 [27 42]]
```

Matrix-Matrix element-wise operations

[143]:
```
Z = np.random.randint(10, size=(3,2))+1
print('Matrix Y:\n',Y)
print('Matrix Z:\n',Z)
```

```
print("\n Matrix-matrix element-wise addition\n")
print(Y+Z)
print("\n Matrix-matrix element-wise multiplication\n")
print(Y*Z)
print("\n Matrix-matrix element-wise division\n")
print(Y/Z)
```

```
Matrix Y:
 [[8 8]
 [3 6]
 [6 6]]
Matrix Z:
 [[ 2  5]
 [ 9  7]
 [ 4 10]]

 Matrix-matrix element-wise addition

[[10 13]
 [12 13]
 [10 16]]

 Matrix-matrix element-wise multiplication

[[16 40]
 [27 42]
 [24 60]]

 Matrix-matrix element-wise division

[[4.         1.6       ]
 [0.33333333 0.85714286]
 [1.5        0.6       ]]
```

2.0.6 Creating Sparse Arrays

For sparse arrays we need to use the sp_sparse library from SciPy: http://docs.scipy.org/doc/scipy/reference/sparse.html

There are three types of sparse matrices:

csr: compressed row format

csc: compressed column format

lil: list of lists format

coo: coordinates format

These different types have to do with the matrix implementation and the data structures used.

The csr and csc formats are fast for arithmetic operations, but slow for slicing and incremental changes.

The lil format is fast for slicing and incremental construction, but slow for arithmetic operations.

The coo format does not support arithmetic operations and slicing, but it is very fast for constructing a matrix incrementally. You should then transform it to some other format for operations.

Creation of matrix from triplets

Triplets are of the form (row, column, value)

[60]:
```
import scipy.sparse as sp_sparse

d = np.array([[0, 0, 12],
              [0, 1, 1],
              [0, 5, 34],
              [1, 3, 12],
              [1, 2, 6],
              [2, 0, 23],
              [3, 4, 14],
             ])
row = d[:,0]
col = d[:,1]
data = d[:,2]
# a matrix M with M[row[i],col[i]] = data[i] will be created
M = sp_sparse.csr_matrix((data,(row,col)), shape=(5,6))
print(M)
print(M.toarray()) #transforms back to full matrix
```

```
  (0, 0)        12
  (0, 1)        1
  (0, 5)        34
  (1, 2)        6
  (1, 3)        12
  (2, 0)        23
  (3, 4)        14
[[12  1  0  0  0 34]
 [ 0  0  6 12  0  0]
 [23  0  0  0  0  0]
 [ 0  0  0  0 14  0]]
```

Making a full matrix sparse

[87]:
```
x = np.random.randint(2,size = (3,4))
print(x)
print('\n make x sparce')
A = sp_sparse.csr_matrix(x)
print(A)
```

```
[[1 0 0 0]
```

```
[0 1 0 1]
[1 0 1 0]]

make x sparce
  (0, 0)        1
  (1, 1)        1
  (1, 3)        1
  (2, 0)        1
  (2, 2)        1
```

Creating a sparse matrix incrementally

[88]:
```
# Use lil (list of lists) representation
A = sp_sparse.lil_matrix((10, 10))
A[0, :5] = np.random.randint(10,size = 5)
A[1, 5:10] = A[0, :5]
A.setdiag(np.random.randint(10,size = 10))
A[9,9] = 99
A[9,0]=1
print(A)
print(A.toarray())
print(A.diagonal())
A = A.tocsr() # makes it a compressed column format. better for dot product.
B = A.dot(np.ones(10))
print(B)
```

```
  (0, 0)        2.0
  (0, 1)        4.0
  (0, 2)        9.0
  (0, 3)        9.0
  (0, 4)        7.0
  (1, 1)        3.0
  (1, 5)        8.0
  (1, 6)        4.0
  (1, 7)        9.0
  (1, 8)        9.0
  (1, 9)        7.0
  (2, 2)        1.0
  (3, 3)        1.0
  (4, 4)        5.0
  (6, 6)        2.0
  (7, 7)        6.0
  (8, 8)        8.0
  (9, 0)        1.0
  (9, 9)        99.0
[[ 2.  4.  9.  9.  7.  0.  0.  0.  0.  0.]
 [ 0.  3.  0.  0.  0.  8.  4.  9.  9.  7.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.]
```

```
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  5.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  2.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  6.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  8.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0. 99.]]
[ 2.  3.  1.  1.  5.  0.  2.  6.  8. 99.]
[ 31.  40.   1.   1.   5.   0.   2.   6.   8. 100.]
```

All operations work like before

[90]:
```
print(A.dot(A.T))
```

```
  (0, 4)        35.0
  (0, 3)        9.0
  (0, 2)        9.0
  (0, 1)        12.0
  (0, 9)        2.0
  (0, 0)        231.0
  (1, 9)        693.0
  (1, 8)        72.0
  (1, 7)        54.0
  (1, 6)        8.0
  (1, 1)        300.0
  (1, 0)        12.0
  (2, 2)        1.0
  (2, 0)        9.0
  (3, 3)        1.0
  (3, 0)        9.0
  (4, 4)        25.0
  (4, 0)        35.0
  (6, 6)        4.0
  (6, 1)        8.0
  (7, 7)        36.0
  (7, 1)        54.0
  (8, 8)        64.0
  (8, 1)        72.0
  (9, 1)        693.0
  (9, 9)        9802.0
  (9, 0)        2.0
```

2.1 Singluar Value Decomposition

For the singular value decomposition we will use the libraries from Numpy and SciPy and SciKit Learn

We use sklearn to create a low-rank matrix (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_low_rank_matrix.html). We will

create a matrix with *effective rank* 2.

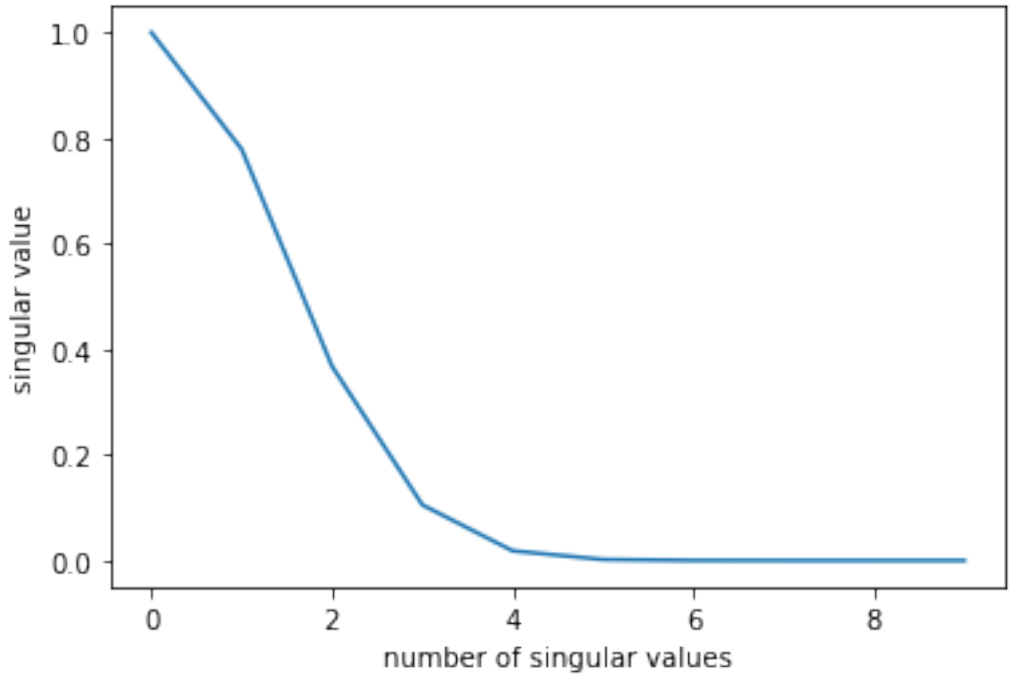
[145]:
```
import sklearn.datasets as sk_data

data = sk_data.make_low_rank_matrix(n_samples=100, n_features=50,␣
 ↪effective_rank=2, tail_strength=0.0, random_state=None)
#sns.heatmap(data, xticklabels=False, yticklabels=False, linewidths=0)
```

We will use the numpy.linalg.svd function to compute the Singular Value Decomposition of the matrix we created (http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.svd.html).

[146]:
```
U, s, V = np.linalg.svd(data,full_matrices = False)
print (U.shape, s.shape, V.shape)
print(s)
plt.plot(s[0:10])
plt.ylabel('singular value')
plt.xlabel('number of singular values')
```

```
(100, 50) (50,) (50, 50)
[1.00000000e+00 7.78800783e-01 3.67879441e-01 1.05399225e-01
 1.83156389e-02 1.93045414e-03 1.23409804e-04 4.78511739e-06
 1.12535175e-07 1.60522805e-09 1.38879449e-11 7.28769945e-14
 2.34726867e-16 7.93002943e-17 7.93002943e-17 7.93002943e-17
 7.93002943e-17 7.93002943e-17 7.93002943e-17 7.93002943e-17
 7.93002943e-17 7.93002943e-17 7.93002943e-17 7.93002943e-17
 7.93002943e-17 7.93002943e-17 7.93002943e-17 7.93002943e-17
 7.93002943e-17 7.93002943e-17 7.93002943e-17 7.93002943e-17
 7.93002943e-17 7.93002943e-17 7.93002943e-17 7.93002943e-17
 7.93002943e-17 7.93002943e-17 7.93002943e-17 7.93002943e-17
 7.93002943e-17 7.93002943e-17 7.93002943e-17 7.93002943e-17
 7.93002943e-17 7.93002943e-17 7.93002943e-17 7.93002943e-17
 7.93002943e-17 2.16266804e-17]
```

[146]:
```
Text(0.5, 0, 'number of singular values')
```



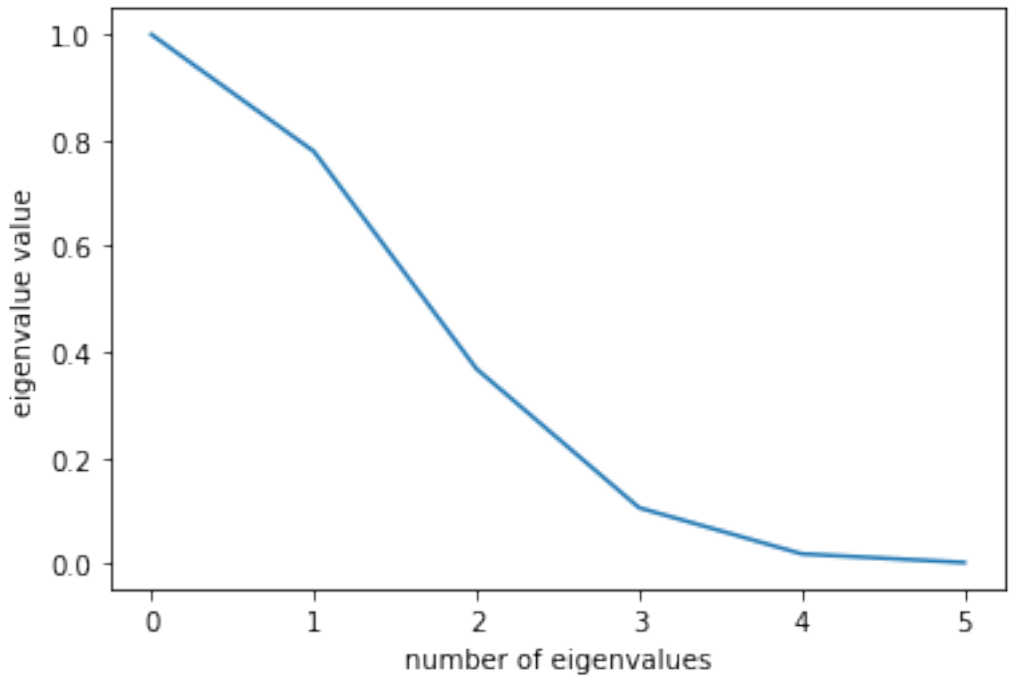
We can also use the scipy.sparse.linalg libary to compute the SVD for sparse matrices (http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.sparse.linalg.svds.html)

We need to specify the number of components, otherwise it is by default k = 6. The singular values are in increasing order.

[147]:
```
import scipy.sparse.linalg as sp_linalg

data2 = sp_sparse.csc_matrix(data)
print(data2.shape)
U,s,V = sp_linalg.svds(data2) #by default returns k=6 singular values
print (U.shape, s.shape, V.shape)
print(s)
plt.plot(s[::-1])
plt.ylabel('eigenvalue value')
plt.xlabel('number of eigenvalues')
```

```
(100, 50)
(100, 6) (6,) (6, 50)
[0.00193045 0.01831564 0.10539922 0.36787944 0.77880078 1.        ]
```

[147]:
```
Text(0.5, 0, 'number of eigenvalues')
```



We can also compute SVD using the library of SciKit Learn (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html)

[154]:
```
from sklearn.decomposition import TruncatedSVD

K = 6
svd = TruncatedSVD(n_components=K)
svd.fit(data2)
print(svd.components_.shape) # the V vectors
print(svd.transform(data2).shape) # the U vectors
print(svd.singular_values_)
```

```
(6, 50)
(100, 6)
[1.         0.77880078 0.36787944 0.10539922 0.01831564 0.00193045]
```

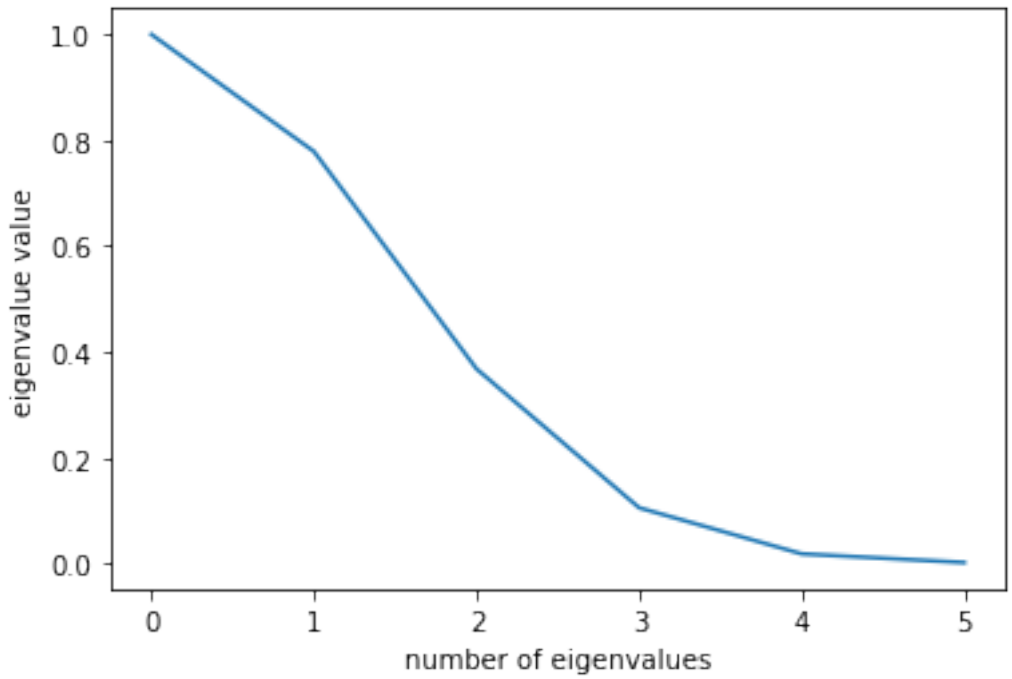
2.1.1 Onbtaining a low rank approximation of the data

To obtain a rank-k approximation of the matrix we multiplty the k first columns of U, with the diagonal matrix with the k first (largest) singular values, with the matrix with the first k rows of V transpose

[155]:
```
K = 6
U_k,s_k,V_k = sp_linalg.svds(data2, K, which = 'LM')
print (U_k.shape, s_k.shape, V_k.shape)
print(s_k)
```

```
plt.plot(s_k[::-1])
plt.ylabel('eigenvalue value')
plt.xlabel('number of eigenvalues')
S_k = np.diag(s_k)
```

```
(100, 6) (6,) (6, 50)
[0.00193045 0.01831564 0.10539922 0.36787944 0.77880078 1.        ]
```

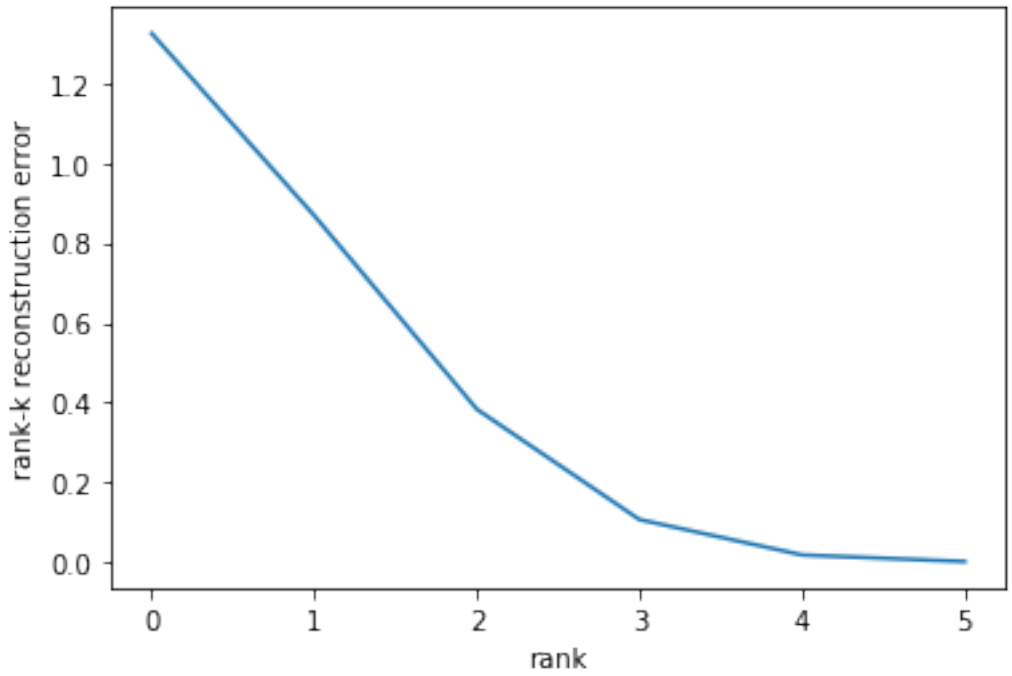


[157]:
```
reconstruction_error = []
for k in range(K,0,-1):
    data_k = U_k[:,k:].dot(S_k[k:,k:]).dot(V_k[k:,:])
    error = np.linalg.norm(data_k-data2,ord='fro')
    reconstruction_error.append(error)
    print(error)
data_k = U_k.dot(S_k).dot(V_k)
print(np.linalg.norm(data_k-data2,ord='fro'))

plt.plot(reconstruction_error)
plt.ylabel('rank-k reconstruction error')
plt.xlabel('rank')
```

```
1.3241276917357527
0.8679367165994609
0.3831233278055767
0.10699626662742329
```

```
0.018417506182009293
0.0019344006983659258
0.0001235025900939376
```

[157]:
```
Text(0.5, 0, 'rank')
```

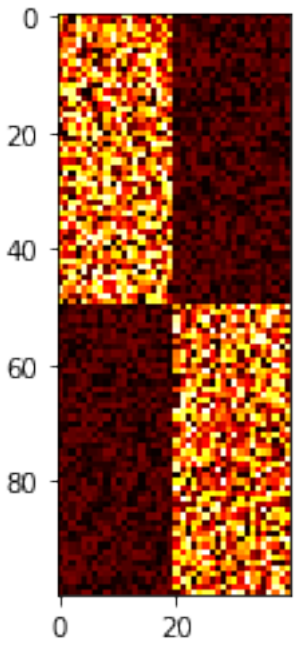


2.2 An example

We will create a block diagonal matrix, with blocks of different "intensity" of values

[159]:
```
import numpy as np

M1 = np.random.randint(1,50,(50,20))
M2 = np.random.randint(1,10,(50,20))
M3 = np.random.randint(1,10,(50,20))
M4 = np.random.randint(1,50,(50,20))

T = np.concatenate((M1,M2),axis=1)
B = np.concatenate((M3,M4),axis=1)
M = np.concatenate([T,B],axis = 0)
```

[160]:
```
plt.imshow(M, cmap='hot')
plt.show()
```



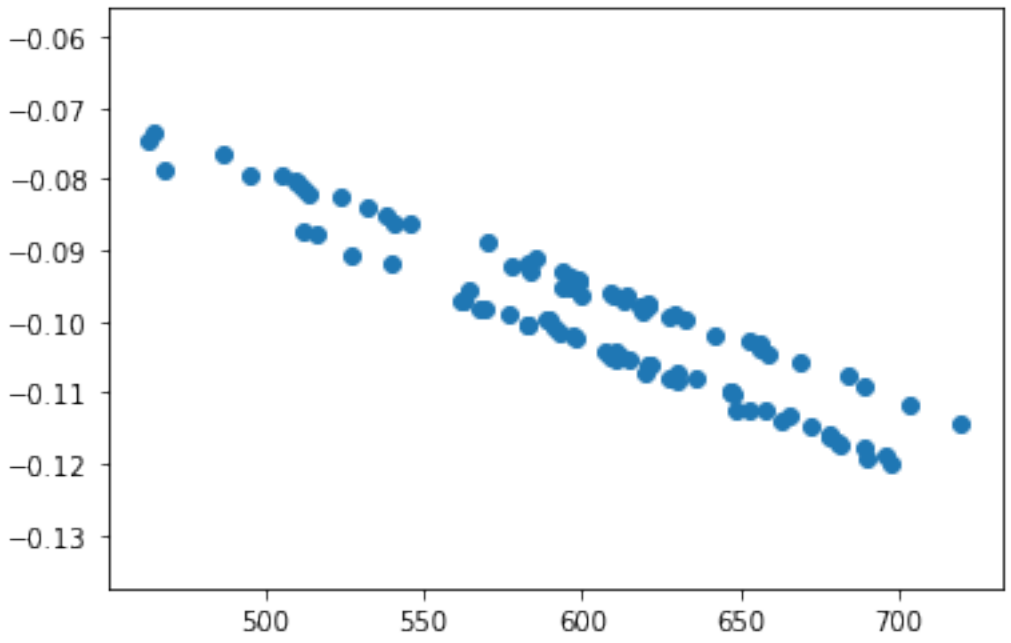
We observe that there is a correlation between the column and row sums and the left and right singular vectors

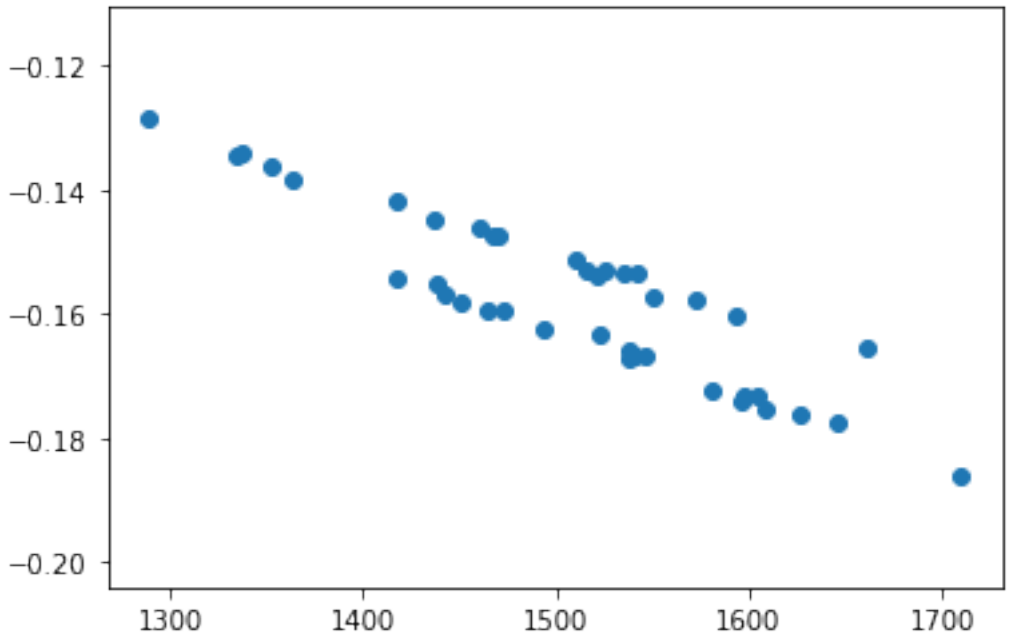
Note: The values of the vectors are negative. We would get the same result if we make them positive.

[161]:
```
import scipy.stats as stats
import matplotlib.pyplot as plt

(U,S,V) = np.linalg.svd(M,full_matrices = False)
#print(S)
c = M.sum(0)
r = M.sum(1)
print(stats.pearsonr(r,U[:,0]))
print(stats.pearsonr(c,V[0]))
plt.scatter(r,U[:,0])
plt.figure()
plt.scatter(c,V[0])
```

```
(-0.9328698906317444, 3.003528000231514e-45)
(-0.9014295173074168, 2.1652282587630272e-15)
```

[161]: `<matplotlib.collections.PathCollection at 0x147065be3c8>`

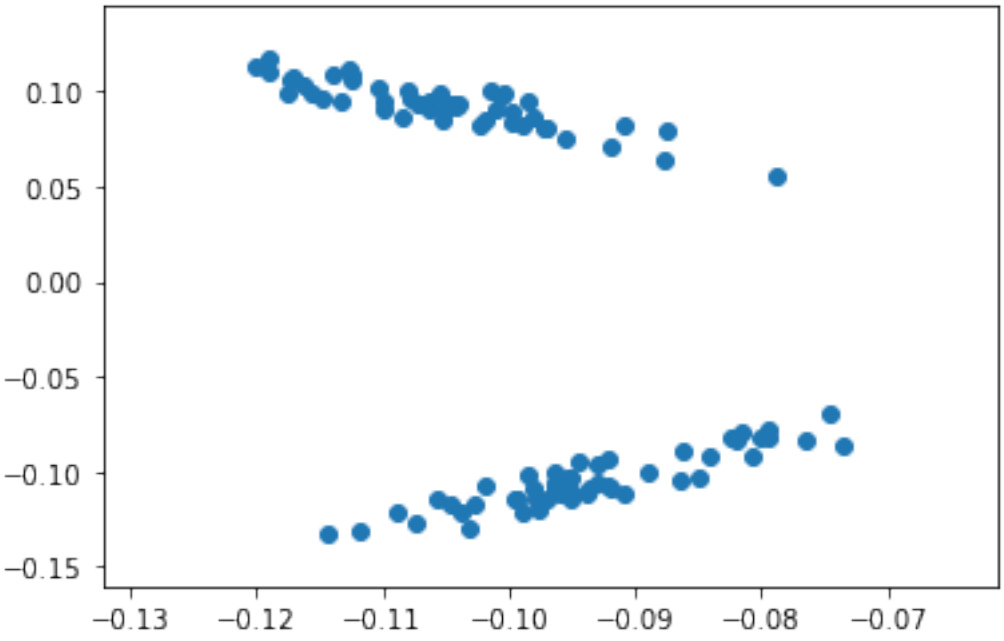




Using the first two signular vectors we can clearly differentiate the two blocks of rows

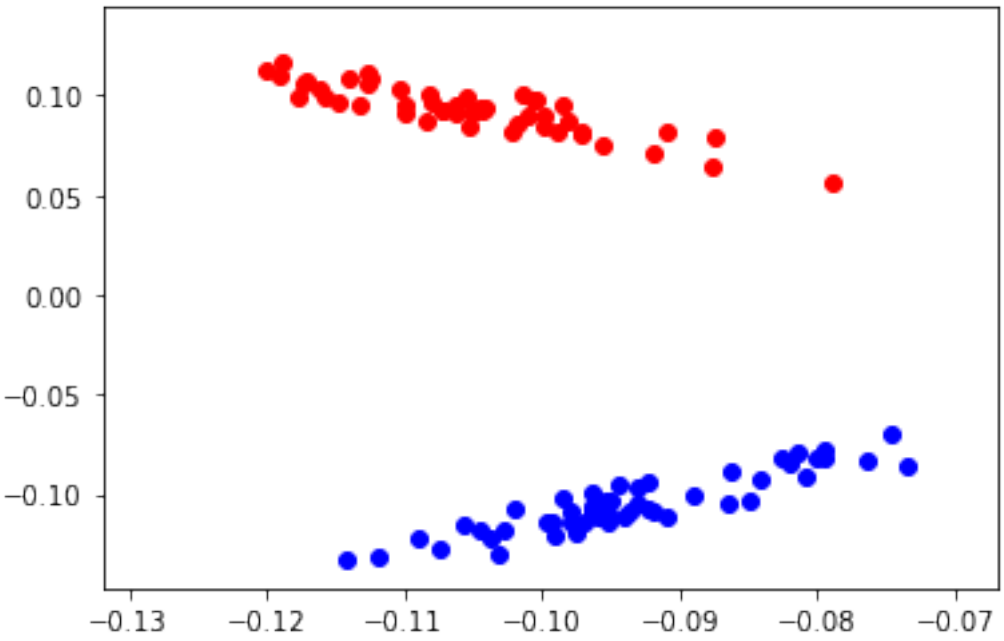
[162]:
```
plt.scatter(U[:,0],U[:,1])
```

[162]:
```
<matplotlib.collections.PathCollection at 0x147065fc668>
```



[163]:
```
plt.scatter(x = U[:50,0],y = U[:50,1],color='r')
plt.scatter(x = U[50:,0],y = U[50:,1], color = 'b')
```

[163]: <matplotlib.collections.PathCollection at 0x147065aeeb8>



2.3 PCA using SciKit Learn

We will now use the PCA package from the SciKit Learn (sklearn) library. PCA is the same as SVD but now the matrix is centered: the mean is removed from the columns of the matrix.

You can read more here: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

[164]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(M)
```

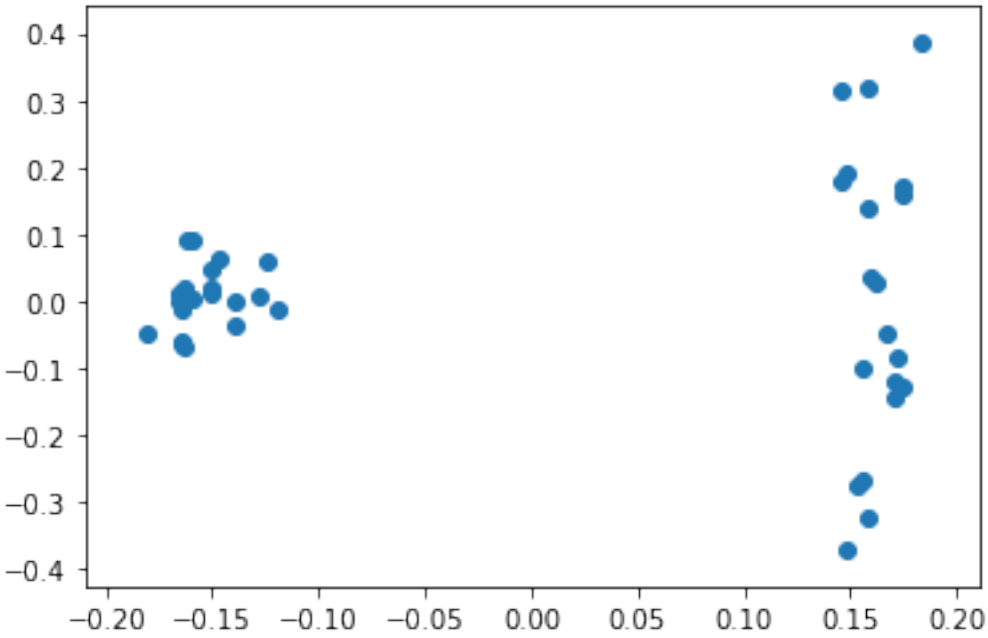
[164]:
```
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
```

[165]:
```
pca.components_
```

[165]:
```
array([[ 0.14607388,  0.14865017,  0.15972337,  0.18318436,  0.17027256,
         0.15578   ,  0.15571973,  0.15759695,  0.17440875,  0.17221999,
         0.16165382,  0.17062192,  0.15849397,  0.15307258,  0.14605926,
         0.1749874 ,  0.17397277,  0.15864738,  0.16719845,  0.14837048,
        -0.15079682, -0.16542681, -0.16337887, -0.15919873, -0.16462017,
        -0.16234409, -0.15110436, -0.11877008, -0.14702569, -0.12804771,
        -0.16402271, -0.15101036, -0.12468711, -0.18098556, -0.13940851,
        -0.15971498, -0.16253908, -0.16577765, -0.16451565, -0.13946264],
       [ 0.17821981,  0.19167097,  0.03486779,  0.38769785, -0.14501363,
        -0.10101094, -0.26519362, -0.32237956,  0.17090714, -0.08159651,
         0.02919841, -0.12086192,  0.14125049, -0.2751426 ,  0.31387488,
        -0.12910746,  0.15936744,  0.3205481 , -0.04865557, -0.37205754,
         0.02028452,  0.0108665 , -0.06655802,  0.09146386, -0.01112039,
         0.09024219,  0.01311397, -0.0128854 ,  0.06349532,  0.0065734 ,
        -0.05843307,  0.04685607,  0.06037158, -0.04720776, -0.0351829 ,
         0.00550299,  0.01875613,  0.00097104, -0.06527167,  0.0020941 ]])
```

[166]:
```
plt.scatter(pca.components_[0],pca.components_[1])
```

[166]:
```
<matplotlib.collections.PathCollection at 0x14707b9f8d0>
```



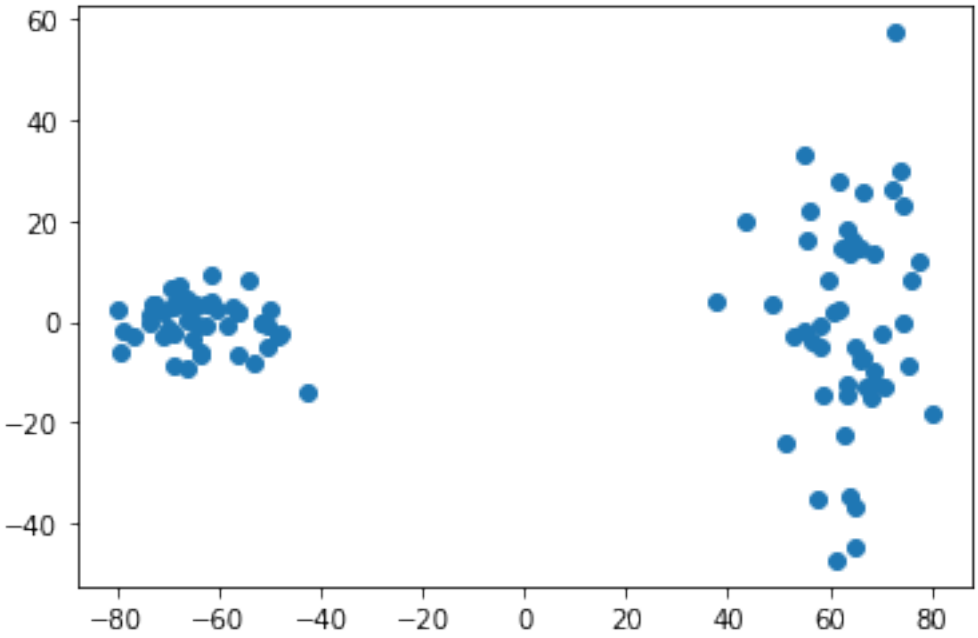
Using the operation transform we can transform the data directly to the lower-dimensional space

[167]:
```
MPCA = pca.transform(M)
print(MPCA.shape)
```

```
(100, 2)
```

[168]:
```
plt.scatter(MPCA[:,0],MPCA[:,1])
```

[168]:
```
<matplotlib.collections.PathCollection at 0x14707c0b860>
```



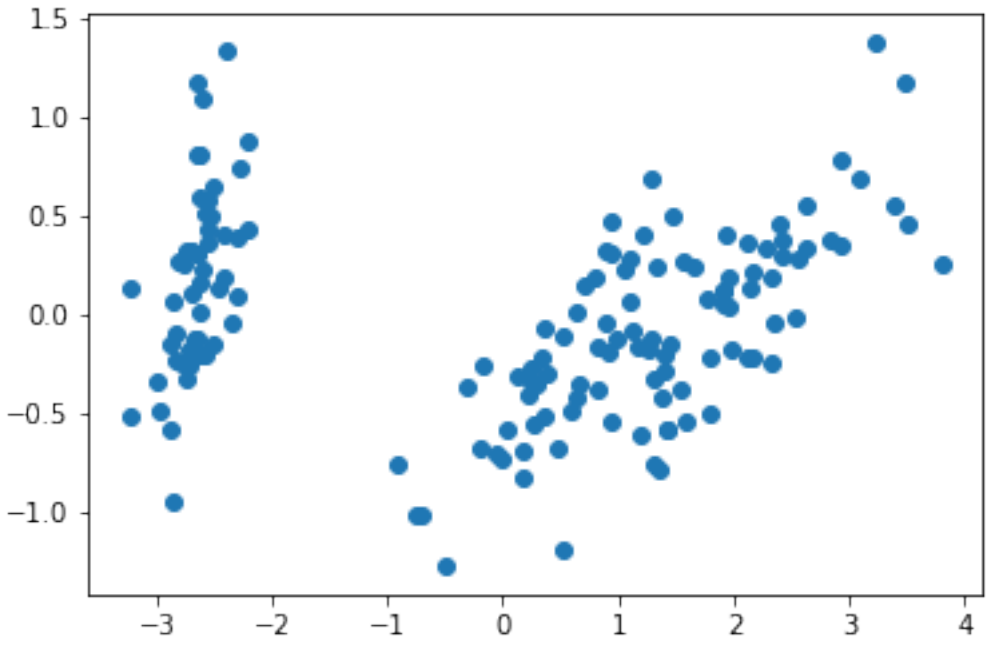
[169]:
```
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target

pca = PCA(n_components=3)
pca.fit(X)
X = pca.transform(X)
```

[170]:
```
pca.explained_variance_
```

[170]:
```
array([4.22824171, 0.24267075, 0.0782095 ])
```

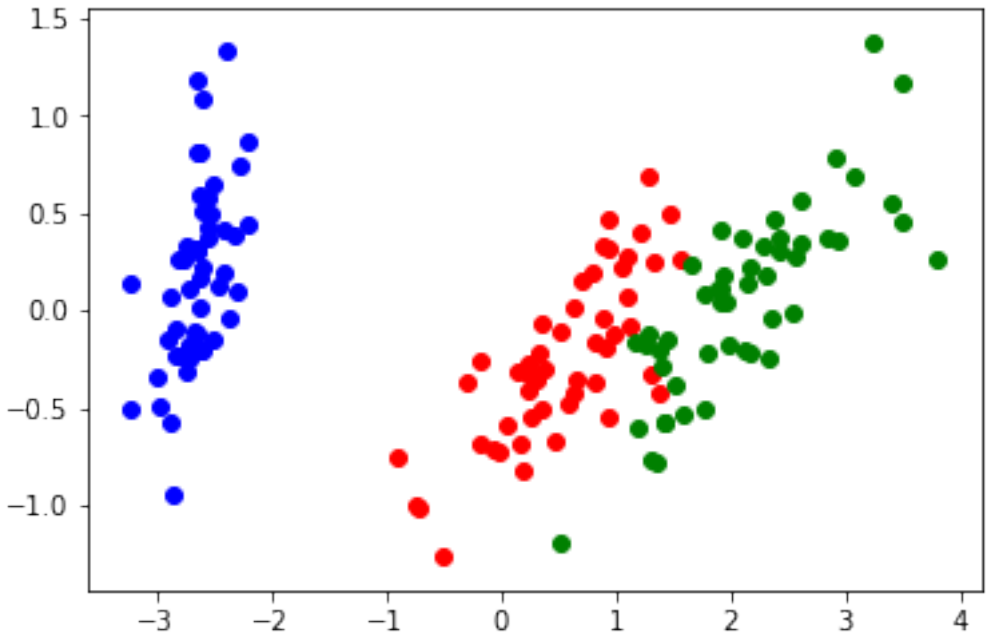
[171]:
```
plt.scatter(X[:,0],X[:,1])
```

[171]:
```
<matplotlib.collections.PathCollection at 0x14707c70a90>
```



[46]:
```
plt.scatter(X[y==0,0],X[y==0,1], color='b')
plt.scatter(X[y==1,0],X[y==1,1], color='r')
plt.scatter(X[y==2,0],X[y==2,1], color='g')
```

[46]: <matplotlib.collections.PathCollection at 0x14706156da0>



[172]:
```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X[y==0,0],X[y==0,1], X[y==0,2], color='b')
ax.scatter(X[y==1,0],X[y==1,1], X[y==1,2], color='r')
ax.scatter(X[y==2,0],X[y==2,1], X[y==2,2], color='g')
```

[172]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x14707cc6240>
```