Scipy

Prof.Dr. Bahadır AKTUĞ
803400815021 Machine Learning with Python

**Compiled from sources given in the references.*

Scipy

- Scipy (Scientific Python), provides numerous scientific and engineering utilities needed for mathematical operations (e.g. Bessel functions, optimization routines etc.)
- In this respect, Scipy is similar to the toolboxes in Matlab and consists of functions that form the scipy module.
- Scipy module is organized in sub modules and those submodules need to be imported before use.
- Similar to Numpy, Scipy is not a built-in modeule and need to be installed separately and imported before use.
- Many of Scipy functions (if not all) depend on the Numpy arrays both for input/output.

Scipy

- Some of the Scipy sub modules:

scipy.cluster	Cluster Analysis
scipy.constants	Matematical and physical constants
scipy.fftpack	Fast Fourier Transformation
scipy.integrate	Integration routines
scipy.interpolate	Interpolation
scipy.io	IO
scipy.linalg	Linear Algebra Utilities
scipy.ndimage	n-dimensional image module
scipy.odr	Orthogonal Distance Regression
scipy.optimize	Optimization
scipy.signal	Signal Processing
scipy.sparse	Sparse Matrices
scipy.spatial	Spatial Data Structures and Algorithms
scipy.special	Special Matrix Functions
scipy.stats	Statistical Functions

Scipy.io

- Scipy heavily depends on Numpy data structures.
- Scipy.io provides several useful practical I/O functions
- For instance Matlab *.mat files can be read and written directly in Python:

```
from scipy import io as spio
a = np.ones((3, 3))
spio.savemat('file.mat', {'a': a})
data = spio.loadmat('file.mat', struct_as_record=True)
data['a']
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Scipy.io

- Similarly, various image formats can be read directly:

```
from scipy import misc
misc.imread('fname.png')
array(...)
```

- Remember Numpy also has file reading utilities:

```
numpy.loadtxt()/numpy.savetxt()
numpy.genfromtxt()/numpy.recfromcsv()
```

- Numpy has its own binary format (similar to Matlab mat files) and its compressed form for fast reading/saving and disk size efficiency

```
numpy.save()/numpy.load()
numpy.savez() → for saving in binary ‘npz’ files
```

Scipy.linalg

- Scipy.linalg is based on highly optimized linear algebra libraries ATLAS LAPACK and BLAS.
- Numpy also has a linear algebra library.
- However, Scipy is not only more comprehensive but also more optimized and efficient for large scale problems.
- Since Scipy Linear Algebra modüle is based on LAPACK and BLAS, it is also much faster than Numpy.linalg modüle.
- Use Scipy Lineary Algebra whenever possible.

Scipy.linalg

- Scipy.linalg provides a comprehensive set of linear algebra functions.
- Example usage:

```
from scipy import linalg
arr = np.array([[1, 2],[3, 4]])
linalg.det(arr)
-2.0

iarr = linalg.inv(arr)
iarr
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Scipy.linalg.det

$$|\mathbf{A}| = \sum_j (-1)^{i+j} a_{ij} M_{ij}.$$

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}$$

$$\begin{aligned} |\mathbf{A}| &= 1 \begin{vmatrix} 5 & 1 \\ 3 & 8 \end{vmatrix} - 3 \begin{vmatrix} 2 & 1 \\ 2 & 8 \end{vmatrix} + 5 \begin{vmatrix} 2 & \\ 2 & 3 \end{vmatrix} \\ &= 1(5 \cdot 8 - 3 \cdot 1) - 3(2 \cdot 8 - 2 \cdot 1) + 5(2 \cdot 3 - 2 \cdot 5) = -25. \end{aligned}$$

```
import numpy as np
from scipy import linalg
arr = np.array([[1, 3,5],[2,5,1],[2,3,8]])
linalg.det(arr)
-25.0
```

Scipy.linalg.inv

$$\mathbf{A}=\begin{bmatrix}1&3&5\\2&5&1\\2&3&8\end{bmatrix}\qquad \mathbf{A}^{-1}=\frac{1}{25}\begin{bmatrix}-37&9&22\\14&2&-9\\4&-3&1\end{bmatrix}=\begin{bmatrix}-1.48&0.36&0.88\\0.56&0.08&-0.36\\0.16&-0.12&0.04\end{bmatrix}$$

```
import numpy as np
from scipy import linalg
arr = np.array([[1, 3,5],[2,5,1],[2,3,8]])
iarr = linalg.inv(arr)
iarr
array([[-1.48 , 0.36, 0.88 ],
       [ 0.56, 0.08, -0.36])
       [ 0.16, -0.12, 0.04]])
```

Scipy.linalg

- Many complex linear algebra routines (e.g. Singular Value Decompositon etc.) are also available:

```
arr = np.arange(9).reshape((3, 3)) + np.diag([1, 0, 1])
uarr, spec, vharr = linalg.svd(arr)
```

- While only SVD, QR and Eigen Decomposition is presented here, there are many others available including:
  - LU
  - Cholesky
  - Schur matrix factorization

Scipy.linalg

$$\begin{aligned} x + 3y + 5z &= 10 \\ 2x + 5y + z &= 8 \\ 2x + 3y + 8z &= 3 \end{aligned}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 10 \\ 8 \\ 3 \end{bmatrix} = \frac{1}{25} \begin{bmatrix} -232 \\ 129 \\ 19 \end{bmatrix} = \begin{bmatrix} -9.28 \\ 5.16 \\ 0.76 \end{bmatrix}$$

- Let’s solve the linear equation above with scipy.linalg:

```
import numpy as np
from scipy import linalg
A = np.array([[1, 3,5],[2,5,1],[2,3,8]])
b = np.array([[10], [8],[3]])
linalg.inv(A).dot(b)  OR linalg.inv(A) @ b
array([[-9.28], [5.16],[ 0.76]])
```

Scipy.linalg

$$\begin{aligned} x + 3y + 5z &= 10 \\ 2x + 5y + z &= 8 \\ 2x + 3y + 8z &= 3 \end{aligned} \qquad \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 10 \\ 8 \\ 3 \end{bmatrix} = \frac{1}{25} \begin{bmatrix} -232 \\ 129 \\ 19 \end{bmatrix} = \begin{bmatrix} -9.28 \\ 5.16 \\ 0.76 \end{bmatrix}$$

- A faster solution can be achieved by using "solve":

```
import numpy as np
from scipy import linalg
A = np.array([[1, 3,5],[2,5,1],[2,3,8]])
b = np.array([[10], [8],[3]])
np.linalg.solve(A, b)
array([[-9.28], [5.16],[ 0.76]])
```

Scipy.linalg Matrix/Vector Norms

$$\|\mathbf{x}\| = \begin{cases} \max |x_i| & \text{ord} = \text{inf} \\ \min |x_i| & \text{ord} = -\text{inf} \\ \left(\sum_i |x_i|^{\text{ord}}\right)^{1/\text{ord}} & |\text{ord}| < \infty. \end{cases}$$

$$\|\mathbf{A}\| = \begin{cases} \max_i \sum_j |a_{ij}| & \text{ord} = \text{inf} \\ \min_i \sum_j |a_{ij}| & \text{ord} = -\text{inf} \\ \max_j \sum_i |a_{ij}| & \text{ord} = 1 \\ \min_j \sum_i |a_{ij}| & \text{ord} = -1 \\ \max \sigma_i & \text{ord} = 2 \\ \min \sigma_i & \text{ord} = -2 \\ \sqrt{\text{trace}\left(\mathbf{A}^H \mathbf{A}\right)} & \text{ord} = \text{'fro'} \end{cases}$$

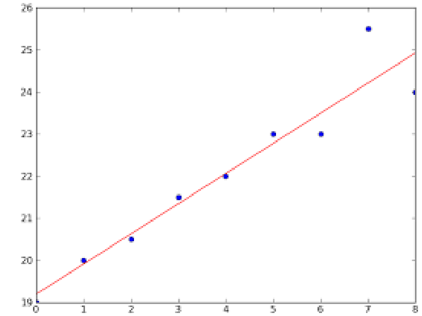
```
import numpy as np
from scipy import linalg
A=np.array([[1,2],[3,4]])
linalg.norm(A) →5.477225575051661
linalg.norm(A,'fro') →5.477225575051661
linalg.norm(A,1) → 6
linalg.norm(A,-1) → 4
linalg.norm(A,np.inf) → 7
```

Scipy.linalg Least Squares Solution

$$\|(w_1 x_i + w_2) - y_i\|^2$$

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ . & 1 \\ . & 1 \\ . & 1 \\ x_n & 1 \end{pmatrix} \mathbf{w} = \begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{pmatrix}$$

```
from numpy import arange,array,ones,linalg
xi = arange(0,9)
A = array([ xi, ones(9)])
y = [19, 20, 20.5, 21.5, 22, 23, 23, 25.5, 24]
w = linalg.lstsq(A.T,y)[0] → doğrunun parametreleri
## w, resid, rank, sigma = linalg.lstsq(....)
```



Scipy.linalg Generalized Matrix Inversion

- Generalized inverse of a matrix can be obtained through Moore-Penrose inverse function included in the Scipy.linalg module.
- Two different algorithms are available for Moore-Penrose inverse:
  - pinv (based on least-squares)
  - pinv2 (based on SVD)

```
A = floor(random.rand(4,4)*20-10)
b = floor(random.rand(4,1)*20-10) → Ax=b
pinv = linalg.pinv(A)
xPinv = dot(pinv,b)
```

Scipy.linalg Singular Value Decomposition (SVD)

- Singular Value Decomposition is one of the most powerful algorithms in Linear Algebra.
- SVD can be used to analyze vector spaces or ill-conditioned problems.
- One particular application is the Moore-Penrose inverse.
- The problem in the previous slide can solved explicitly by SVD as follows:

```
A = floor(random.rand(4,4)*20-10)
b = floor(random.rand(4,1)*20-10) → Ax=b
U,s,V = linalg.svd(A) # A'nın SVD Ayrıştırılması
pinv_svd = dot(dot(V.T,linalg.inv(diag(s))),U.T)
```

Scipy.linalg QR Factorization

- QR Matrix Decomposition (Factorization) is also another powerful method for linear equations.
- The equation in the previous slide can also be solved through QR as follows:

$$\|Ax - b\|_2 \qquad QRx = b \qquad Q^T QRx = Q^T b \qquad Rx = Q^T b$$

```
A = random.rand(5,3)
b = random.rand(5,1)
Q,R = linalg.qr(A)
Qb = dot(Q.T,b)
x_qr = linalg.solve(R,Qb)
```

Scipy.linalg Eigenvalue/Eigenvector

- Eigenvalue Decomposition is also another popular matrix factorization method.

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1, 2], [3, 4]])
>>> la, v = linalg.eig(A)
>>> l1, l2 = la
>>> print(l1, l2)   # eigen values
(-0.372281323269+0j) (5.37228132327+0j)
>>> print v[:, 0]   # first eigen vector
[-0.82456484  0.56576746]
```

Scipy.fftpack

- Scipy provides a very efficient Fast Fourier Transformation (FFT) modüle:

$$y[k] = \sum_{n=0}^{N-1} e^{-2\pi j \frac{kn}{N}} x[n] \qquad x[n] = \frac{1}{N} \sum_{n=0}^{N-1} e^{2\pi j \frac{kn}{N}} y[k]$$

```
>>> from scipy.fftpack import fft, ifft
>>> x = np.array([1.0, 2.0, 1.0, -1.0, 1.5])
>>> y = fft(x)
>>> y
array([ 4.50000000+0.j        ,  2.08155948-1.65109876j,
       -1.83155948+1.60822041j, -1.83155948-1.60822041j,
        2.08155948+1.65109876j])
>>> yinv = ifft(y)
>>> yinv
array([ 1.0+0.j,  2.0+0.j,  1.0+0.j, -1.0+0.j,  1.5+0.j])
```

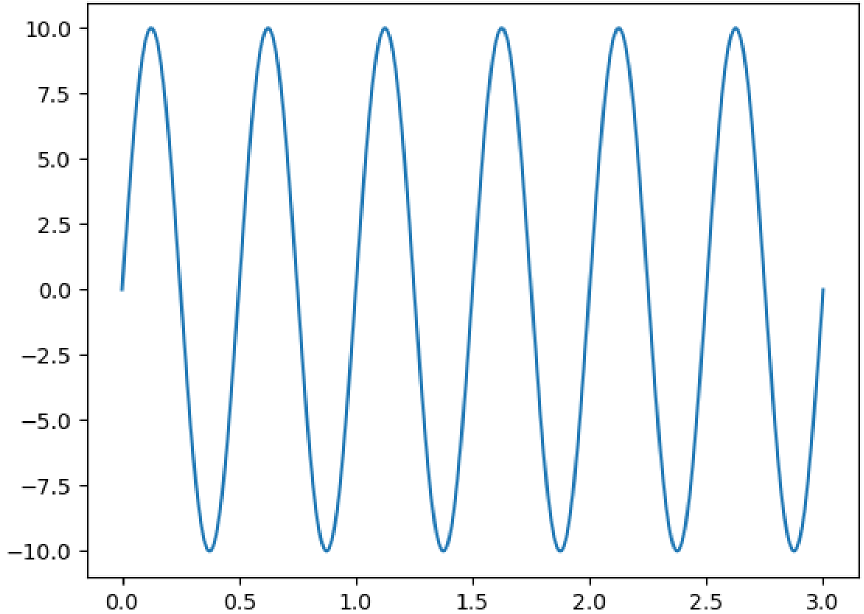
Scipy.fftpack

- Scipy.fftpack containes other Fast Fourier related functions as well:

```
from scipy import fftpack
time_step = 0.02
period = 5.
time_vec = np.arange(0, 20, time_step)
sig = np.sin(2 * np.pi / period * time_vec) + \
        0.5 * np.random.randn(time_vec.size)
sample_freq = fftpack.fftfreq(sig.size, d=time_step)
sig_fft = fftpack.fft(sig)
```

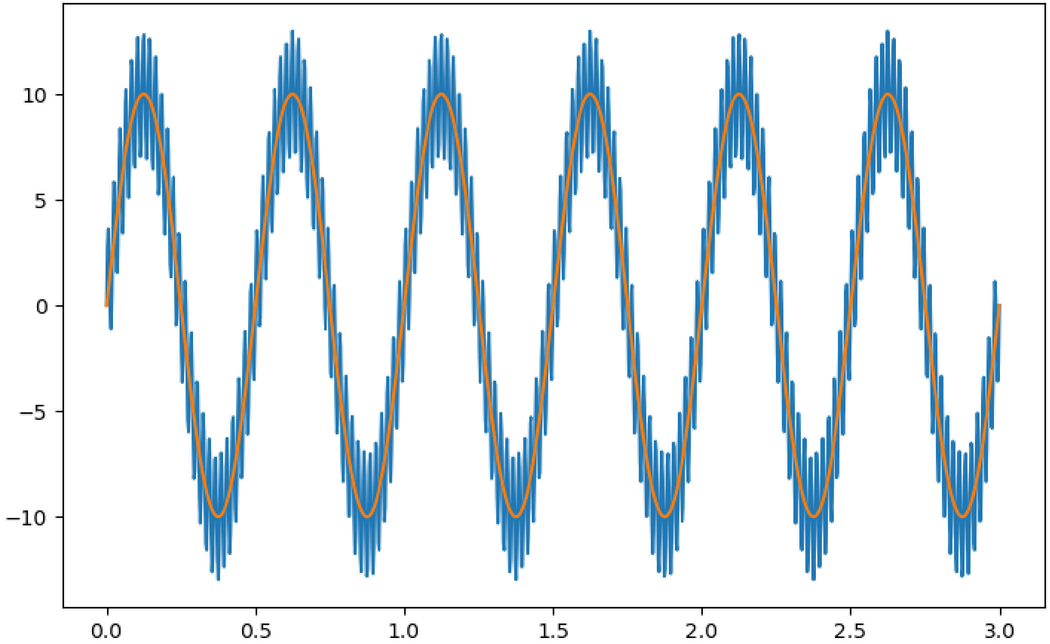
Scipy.signal

```
import numpy as np
time = np.linspace(0,3,1000,endpoint=True)
signal_freq = 2 # Signal Frequency
signal_amplitude = 10 # Signal Amplitude
signal = signal_amplitude*np.sin(2*np.pi*signal_freq*time)
```



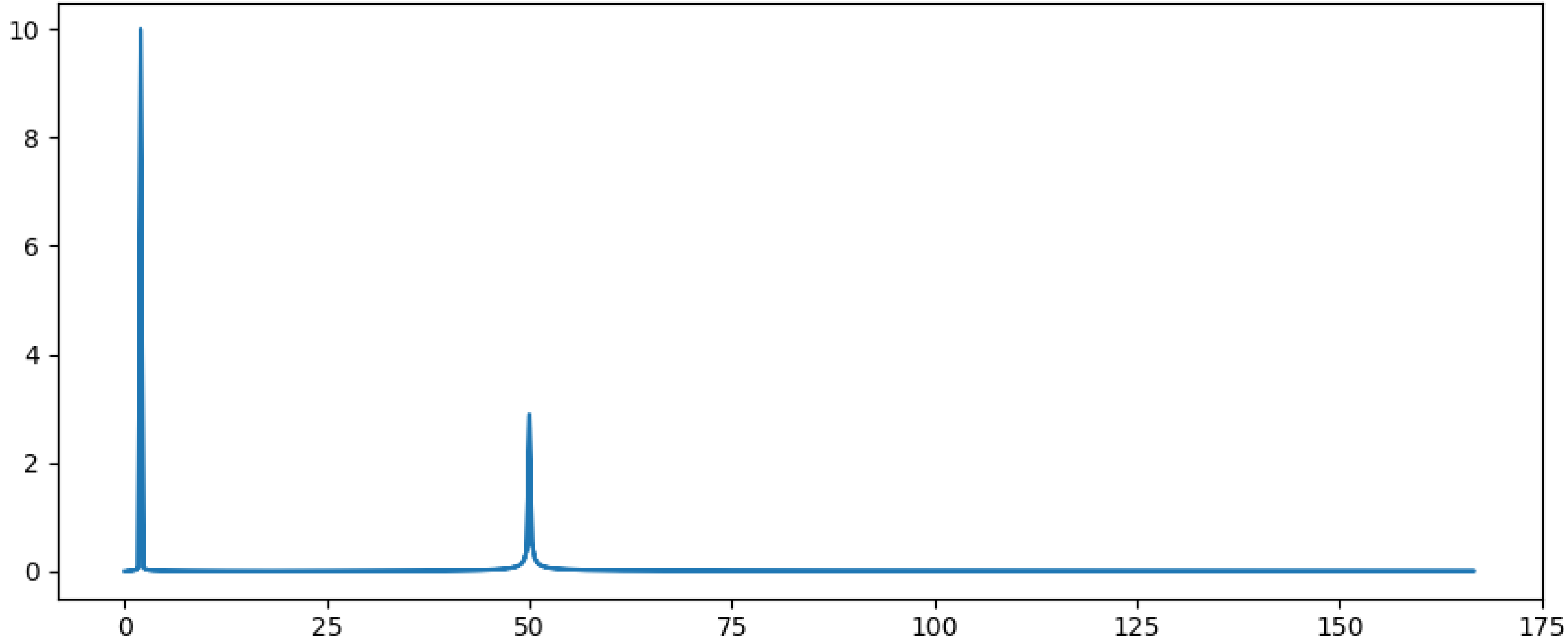
Scipy.signal

```
noise_freq = 50 # Noise Frequency
noise_amplitude = 3 # Noise Amplitude
#Sine wave Noise
noise = noise_amplitude*np.sin(2*np.pi*noise_freq*time)
# Generated Signal with Noise
signal_noise = signal + noise
```



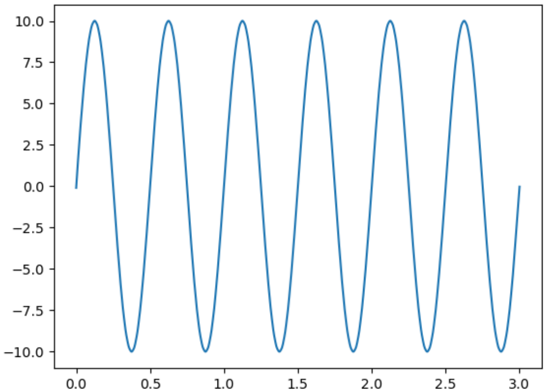
Scipy.signal

```
sig_noise_fft = fftpack.fft(signal_noise)
sig_noise_amp = 2 / time.size * np.abs(sig_noise_fft)
sig_noise_freq = np.abs(fftpack.fftfreq(time.size, 3/1000))
```



Scipy.signal

```python
from scipy.signal import butter,filtfilt
# Filter requirements.
fs = 50.0       # sample rate, Hz
cutoff = 2      # Hz
order = 2       # sin wave can be approx represented as quadratic

def butter_lowpass_filter(data, cutoff, fs, order):
    print("Cutoff freq " + str(cutoff))
    nyq = 0.5 * fs # Nyquist Frequency
    normal_cutoff = cutoff / nyq
    # Get the filter coefficients
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    y = filtfilt(b, a,data)
    return y
# Filter the data, and plot filtered signals.
y = butter_lowpass_filter(signal_noise, 2, fs, order)
```



Scipy.optimize

- Scipy.optimize provides local optimization algorithms (the most popular one is probably BFGS):

```
from scipy import optimize
def f(x):
    return x**2 + 10*np.sin(x)

optimize.fmin_bfgs(f, 0)
Optimization terminated successfully.
         Current function value: -7.945823
         Iterations: 5
         Function evaluations: 24
         Gradient evaluations: 8
array([-1.30644003])
```

Scipy.optimize

- Scipy.optimize also provides several global optimization functions:

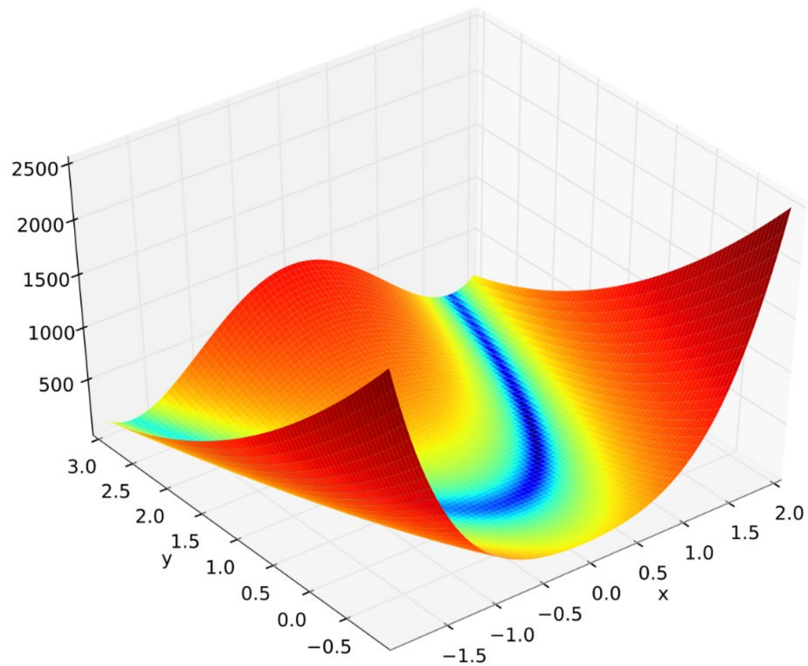
```
optimize.basinhopping(f, 0)
                  nfev: 1725
 minimization_failures: 0
                   fun: -7.9458233756152845
                     x: array([-1.30644001])
               message: ['requested number of basinhopping iterations completed successfully']
                  njev: 575
                   nit: 100
```

Unconstrained Optimization (minimize)

Rosenbrock function:

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} 100\left(x_{i+1} - x_i^2\right)^2 + \left(1 - x_i\right)^2.$$

The minimum of this function is obtained at x_i=1



Nelder-Mead Simplex algorithm (method='Nelder-Mead')

```python
import numpy as np
from scipy.optimize import minimize

def rosen(x):
    """The Rosenbrock function"""
    return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen, x0, method='nelder-mead',
               options={'xatol': 1e-8, 'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571
[1. 1. 1. 1. 1.]
```

Powell Method (method='powell')

```python
import numpy as np
from scipy.optimize import minimize

def rosen(x):
    """The Rosenbrock function"""
    return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen, x0, method='powell',
               options={'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 18
         Function evaluations: 1084
[1. 1. 1. 1. 1.]
```

Broyden-Fletcher-Goldfarb-Shanno algorithm (method='BFGS')

This method requires derivatives with respect to the parameters. The derivative is:

$$f\left(\mathbf{x}\right)=\sum_{i=1}^{N-1}100\left(x_{i+1}-x_{i}^{2}\right)^{2}+\left(1-x_{i}\right)^{2}.$$

$$\begin{aligned} \frac{\partial f}{\partial x_{j}} &= \sum_{i=1}^{N}200\left(x_{i}-x_{i-1}^{2}\right)\left(\delta_{i,j}-2x_{i-1}\delta_{i-1,j}\right)-2\left(1-x_{i-1}\right)\delta_{i-1,j}. \\ &= 200\left(x_{j}-x_{j-1}^{2}\right)-400x_{j}\left(x_{j+1}-x_{j}^{2}\right)-2\left(1-x_{j}\right). \end{aligned}$$

Broyden-Fletcher-Goldfarb-Shanno algorithm (method='BFGS')

```python
import numpy as np
from scipy.optimize import minimize

def rosen(x):
    """The Rosenbrock function"""
    return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)

def rosen_der(x):
    xm = x[1:-1]
    xm_m1 = x[:-2]
    xm_p1 = x[2:]
    der = np.zeros_like(x)
    der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm -
2*(1-xm)
    der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
    der[-1] = 200*(x[-1]-x[-2]**2)
    return der

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
               options={'disp': True})
print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 25
         Function evaluations: 30
         Gradient evaluations: 30
[1.00000004 1.0000001  1.00000021 1.00000044 1.00000092]
```

Newton-Conjugate-Gradient algorithm (method='Newton-CG')

- This method requires second derivatives (Hessian) along with first derivates (Jacobian)

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \mathbf{H}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} 100\left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2.$$

$$\frac{\partial f}{\partial x_j} = \sum_{i=1}^{N} 200\left(x_i - x_{i-1}^2\right)\left(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}\right) - 2\left(1 - x_{i-1}\right)\delta_{i-1,j}.$$

$$= 200\left(x_j - x_{j-1}^2\right) - 400x_j\left(x_{j+1} - x_j^2\right) - 2\left(1 - x_j\right).$$

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} = 200\left(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}\right) - 400x_i\left(\delta_{i+1,j} - 2x_i\delta_{i,j}\right) - 400\delta_{i,j}\left(x_{i+1} - x_i^2\right) + 2\delta_{i,j},$$

$$= \left(202 + 1200x_i^2 - 400x_{i+1}\right)\delta_{i,j} - 400x_i\delta_{i+1,j} - 400x_{i-1}\delta_{i-1,j},$$

Newton-Conjugate-Gradient algorithm (method='Newton-CG')

```
from scipy.optimize import minimize

def rosen(x):
    """The Rosenbrock function"""
    return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)

def rosen_der(x):
    xm = x[1:-1]
    xm_m1 = x[:-2]
    xm_p1 = x[2:]
    der = np.zeros_like(x)
    der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
    der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
    der[-1] = 200*(x[-1]-x[-2]**2)
    return der
def rosen_hess(x):
    x = np.asarray(x)
    H = np.diag(-400*x[:-1],1) - np.diag(400*x[:-1],-1)
    diagonal = np.zeros_like(x)
    diagonal[0] = 1200*x[0]**2-400*x[1]+2
    diagonal[-1] = 200
    diagonal[1:-1] = 202 + 1200*x[1:-1]**2 - 400*x[2:]
    H = H + np.diag(diagonal)
    return H

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen, x0, method='Newton-CG',
               jac=rosen_der, hess=rosen_hess,
               options={'xtol': 1e-8, 'disp': True})
print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 24
         Function evaluations: 33
         Gradient evaluations: 33
         Hessian evaluations: 24
[1.         1.         1.         0.99999999 0.99999999]
```

Constraints

- Interval Constraints (bounds)

$$0 \leq x_0 \leq 1 \text{ and } -0.5 \leq x_1 \leq 2.0$$

```
from scipy.optimize import Bounds
bounds = Bounds([0, -0.5], [1.0, 2.0])
```

- Linear Constraints

$$x_0 + 2x_1 \leq 1 \text{ and } 2x_0 + x_1 = 1$$

$$\begin{bmatrix} -\infty \\ 1 \end{bmatrix} \leq \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

```
from scipy.optimize import LinearConstraint
linear_constraint = LinearConstraint([[1, 2], [2, 1]], [-np.inf, 1], [1, 1])
```

Trust-Region Constrained Algorithm

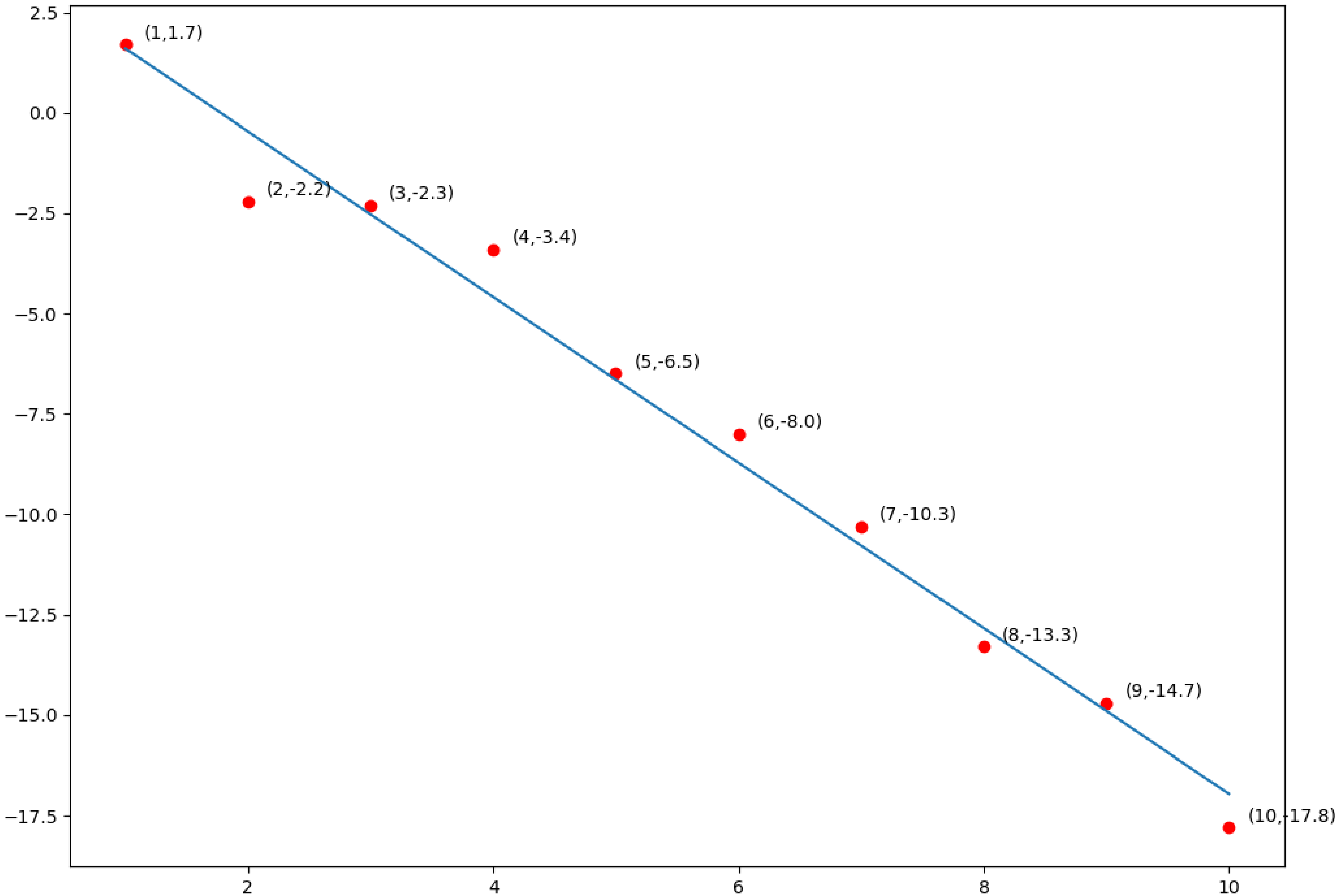
- Interval-constrained solution algorithm is generally used for the problems as follows:

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{subject to:} \quad & c^l \leq c(x) \leq c^u, \\ & x^l \leq x \leq x^u. \end{aligned}$$

- Linear and non-linear constraints can be given together.
- First and second derivates can also be used.

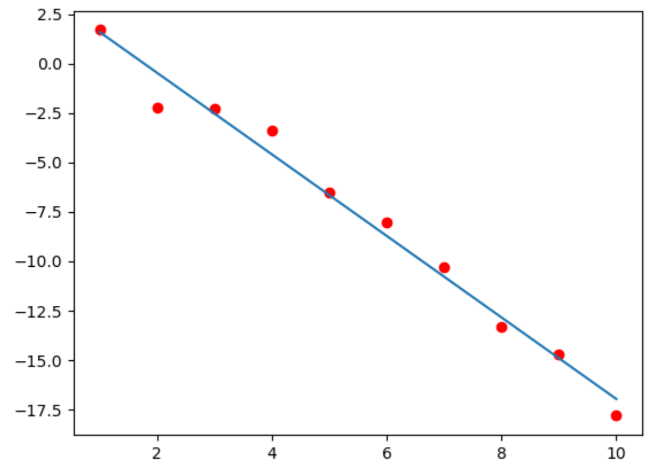
```
res = minimize(rosen, x0, method='trust-constr', jac=rosen_der, hessp=rosen_hess_p,
               constraints=[linear_constraint, nonlinear_constraint],
               options={'verbose': 1}, bounds=bounds)
```

Linear Regression (Line Fitting)



Linear Regression (Line Fitting)

```python
import matplotlib.pyplot as plt

x = np.array([1,2,3,4,5,6,7,8,9,10])
y = np.array([1.7,-2.2,-2.3,-3.4,-6.5,
              -8.0,-10.3,-13.3,
              -14.7,-17.8])
A = []
for i in range(len(x)):
   A.append([1,x[i]])

A = np.array(A)
y = y.reshape(len(y),1)

# a = np.linalg.solve(A.T@A,A.T@y)
# a = np.linalg.lstsq(A,y)
a = np.linalg.pinv(A) @ y
ny = a[0][0] +  a[1][0]*15
print(ny)

n1 = a[0][0] +  a[1][0]*1
n2 = a[0][0] +  a[1][0]*10

plt.plot([1,10],[n1,n2])
plt.scatter(x,y,color='red')
plt.show()
```



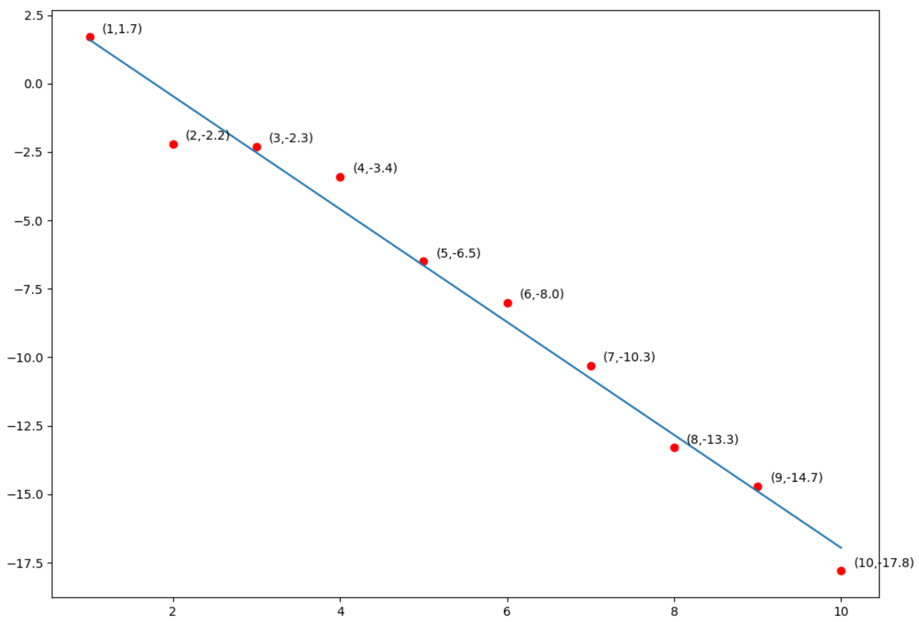
array([[3.66   ],
       [-2.06181818]])

-27.26727272727273

Linear Regression (Nelder-Mead)

- Let's solve the regression problem with Nelder-Mead method and Least-Squares.

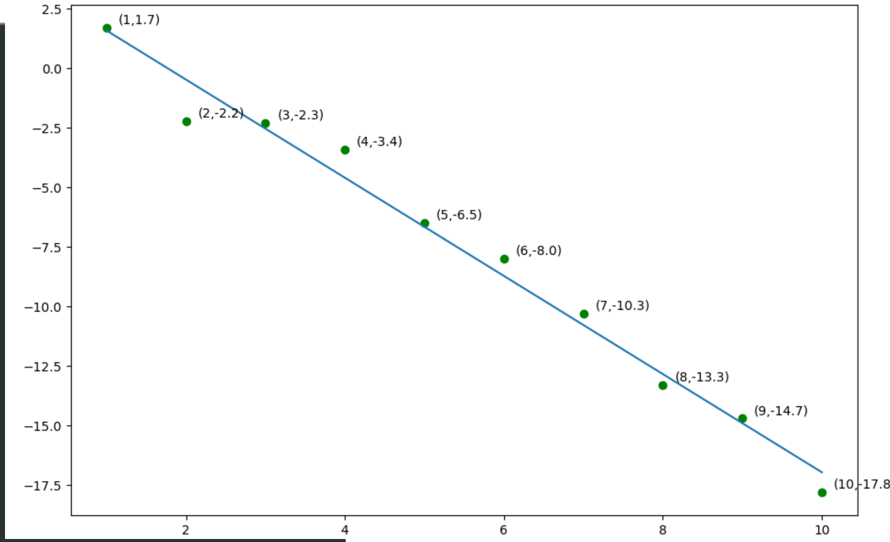
```
x = np.array([1,2,3,4,5,6,7,8,9,10])
y = np.array([1.7,-2.2,-2.3,-3.4,-6.5,
              -8.0,-10.3,-13.3,
              -14.7,-17.8])
```




```
array([[ 3.66      ],
       [-2.06181818]])
```

Linear Regression (Nelder-Mead)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

x = np.array([1,2,3,4,5,6,7,8,9,10])
y = np.array([1.7,-2.2,-2.3,-3.4,-6.5,
            -8.0,-10.3,-13.3,
            -14.7,-17.8])

def costFunc(coef):
    tot = 0
    for ix,iy in zip(x,y):
        tot += (coef[0]+coef[1]*ix - iy)**2
    return tot

coef = np.array([1.0, 1.0])
res = minimize(costFunc, coef, method='nelder-mead',
               options={'xatol': 1e-18, 'ftol':1E-18,
               'maxfev':10000, 'disp': True})

print(res.x)
```




```
[ 3.65999998 -2.06181818]
```

Linear Regression (Nelder-Mead)

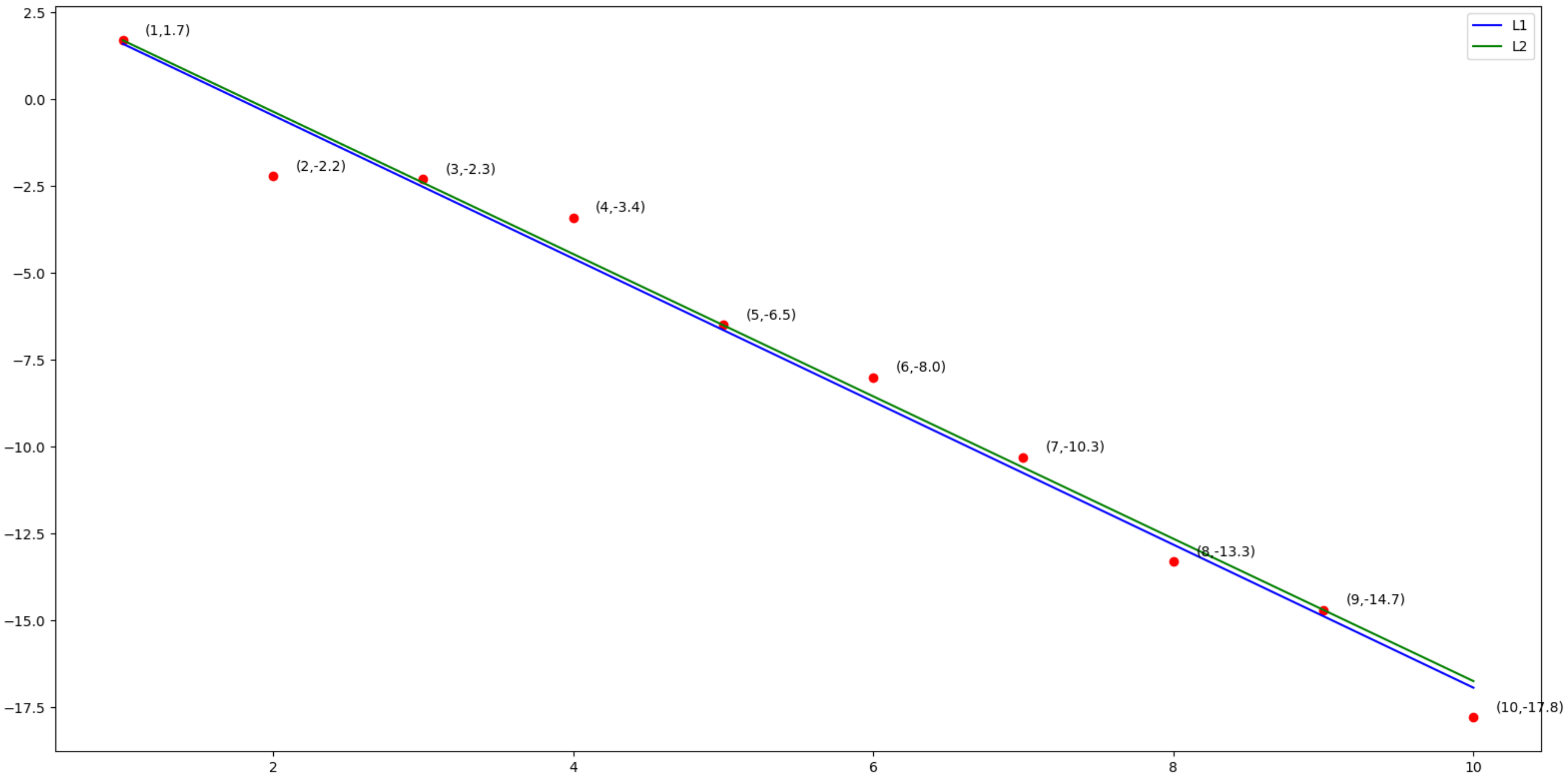
- Let's solve the regression problem with Nelder-Mead method but with L1 norm this time.

```python
def costFunc(coef):
    tot = 0
    for ix,iy in zip(x,y):
        # tot += (coef[0]+coef[1]*ix - iy)**2
        tot += abs(coef[0]+coef[1]*ix - iy)
    return tot
```

```
[ 3.75 -2.05]
```

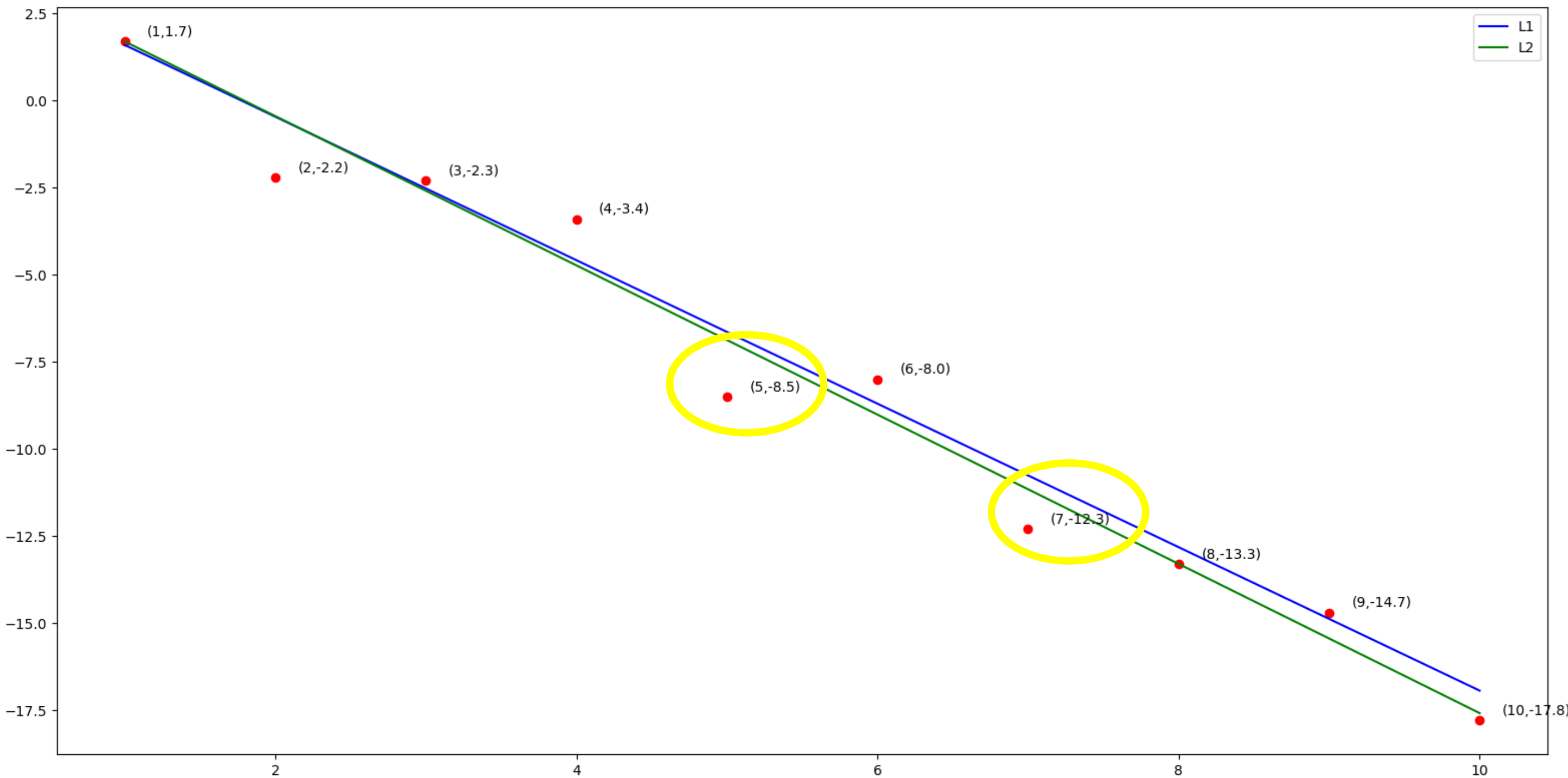
```
array([[ 3.66      ],
       [-2.06181818]])
```

Linear Regression (Nelder-Mead)



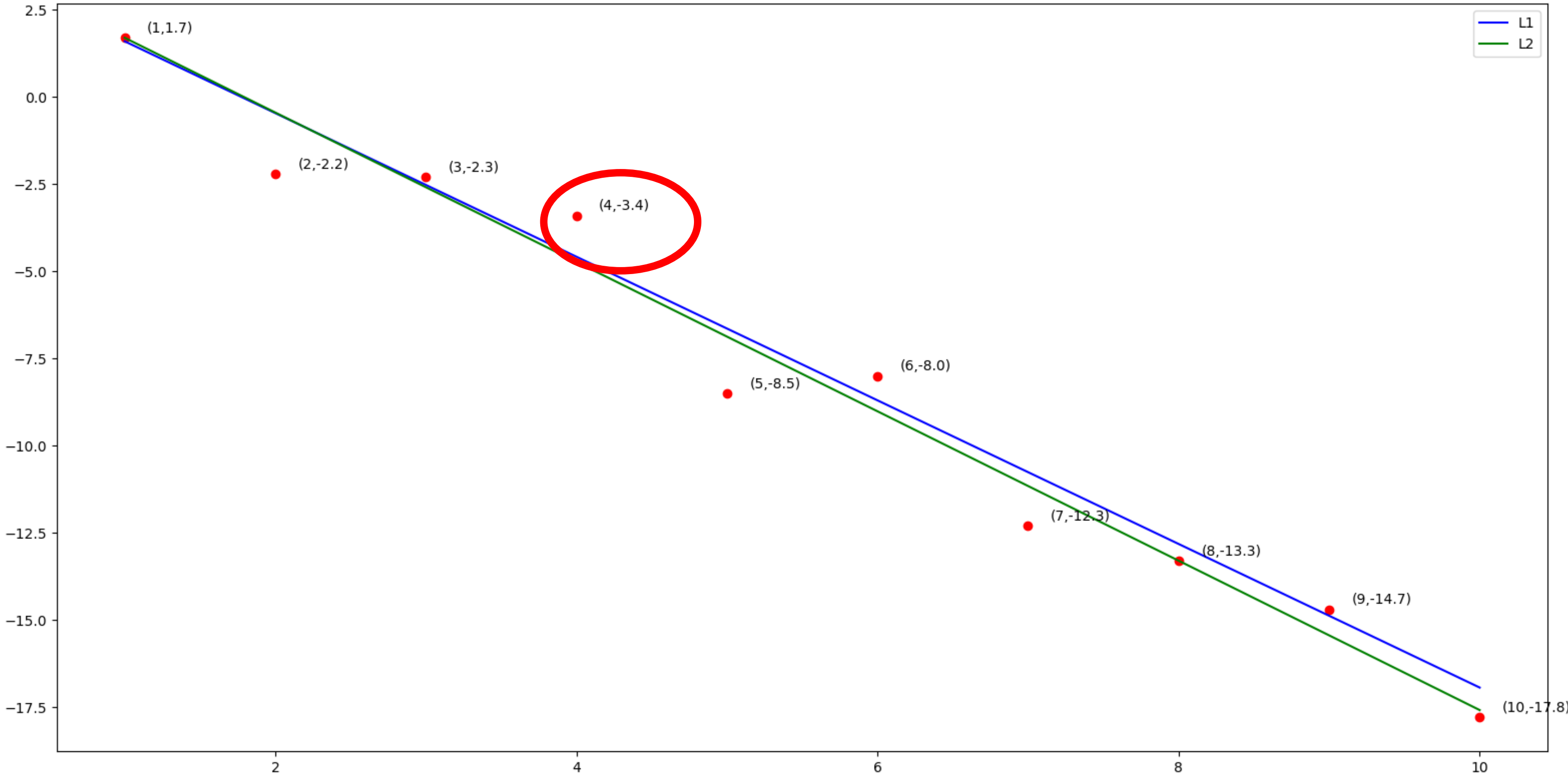
Doğrusal Regresyon (Nelder-Mead)

- Değerleri değiştirerek çözelim.

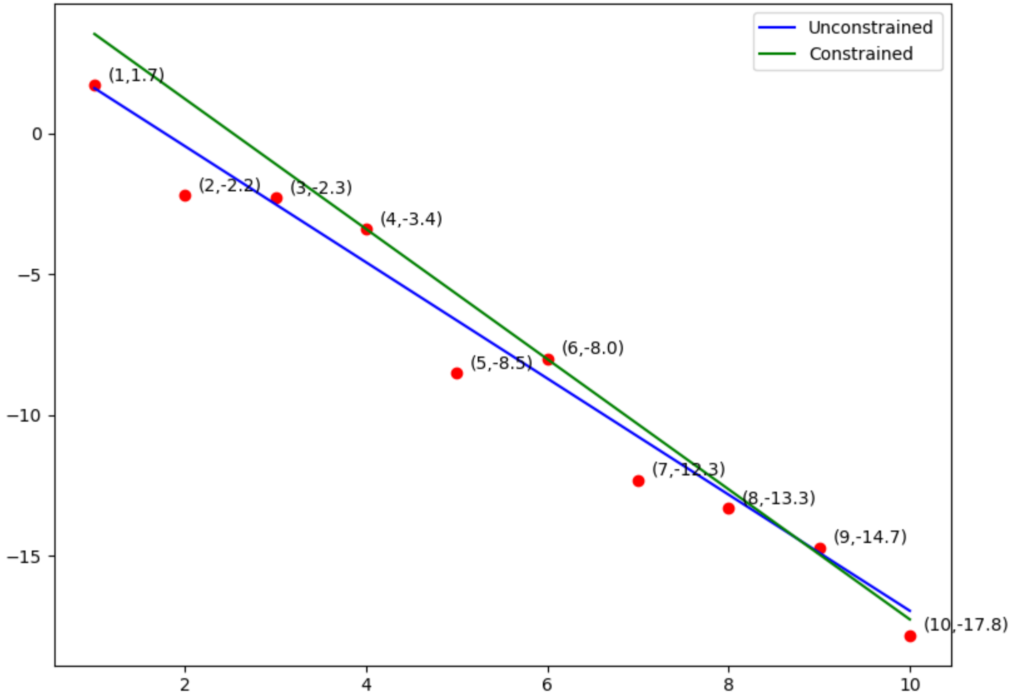


Linear Regression (Trust-Region Constrained Algorithm)

- Let's define a specific constraint.
- We want our line to pass through point at (4,-3.4).



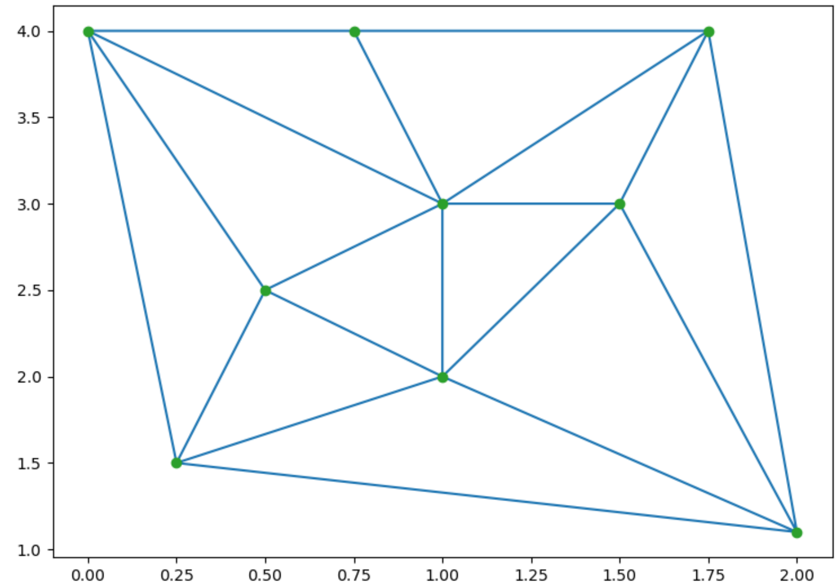
Linear Regression (Trust-Region Constrained Algorithm)




```
coef = np.array([1.0, 1.0])
linear_constraint = LinearConstraint([[1, 4]], [-3.4], [-3.4])
res = minimize(costFunc, coef, method='trust-constr',
               constraints=[linear_constraint],
               options={'disp': True})
```

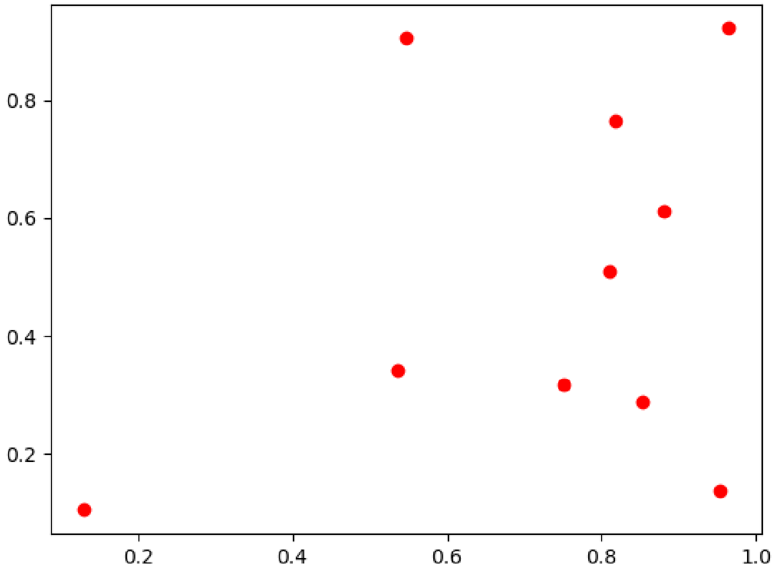
```
Number of iterations: 16, function evaluations: 36, CG iterations: 15, optimality: 1.56e-07, constraint violation: 4.44e
-16, execution time: 0.032 s.
[ 5.83047588 -2.30761897]
```

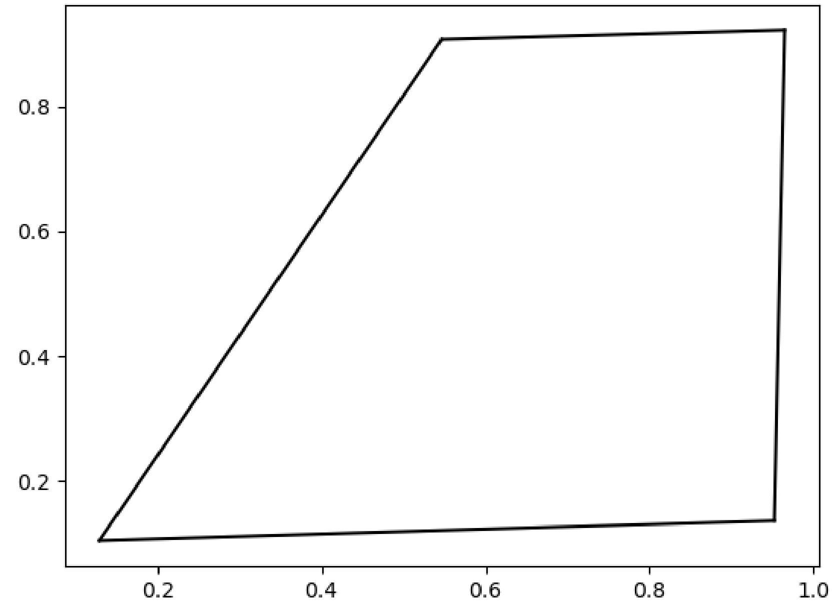
Scipy.spatial (Delaunay Triangulation)




```
from scipy.spatial import Delaunay
import numpy as np
import matplotlib.pyplot as plt

points = np.array([[0,4],[2,1.1],[1,3],[1,2],[0.25,1.5],
[0.75,4.0],[1.5,3.0],[1.75,4],[0.5,2.5]])
tri = Delaunay(points)
plt.triplot(points[:,0], points[:,1], tri.simplices.copy())
plt.plot(points[:,0], points[:,1], 'o')
plt.show()
```

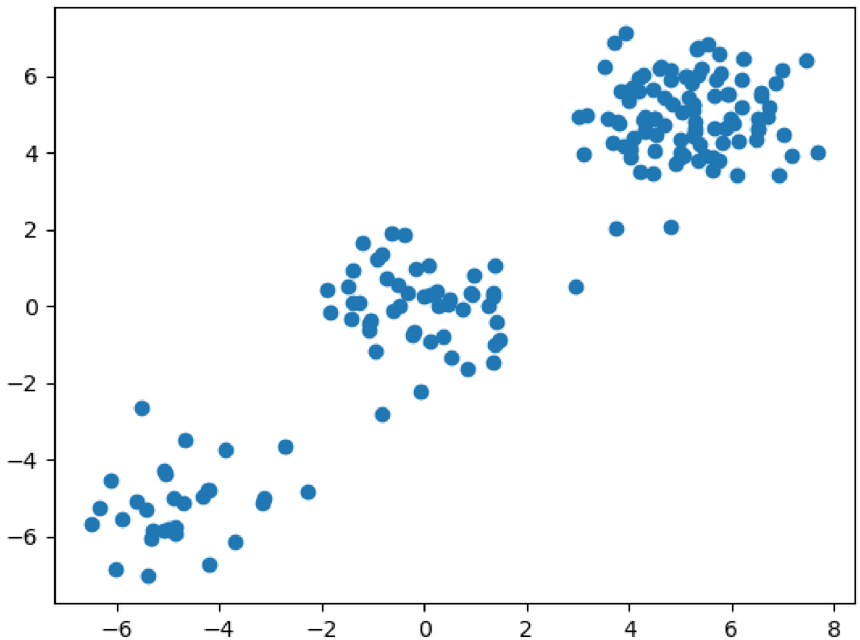
Scipy.spatial (Convex Hull)

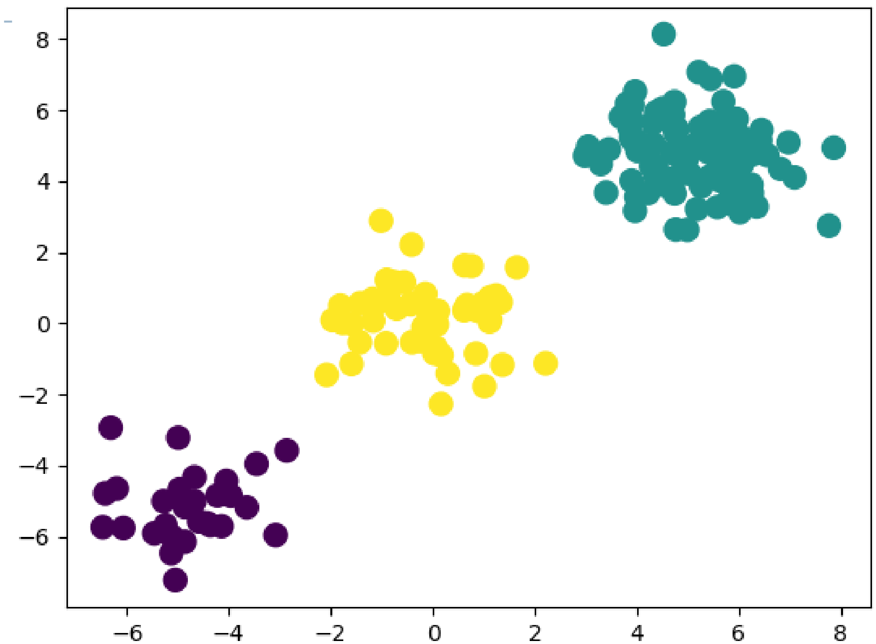





```python
from scipy.spatial import ConvexHull
import numpy as np
points = np.random.rand(10, 2) # 30 random points in 2-D
hull = ConvexHull(points)
import matplotlib.pyplot as plt
plt.plot(points[:,0], points[:,1], 'o')
for simplex in hull.simplices:
    plt.plot(points[simplex,0], points[simplex,1], 'k-')
plt.show()
```

Scipy.cluster (Spatial Grouping)






```
from scipy.cluster import vq
centroids, variance  = vq.kmeans(battles, 3)
identified, distance = vq.vq(battles, centroids)
cluster_1 = battles[identified == 0]
cluster_2 = battles[identified == 1]
cluster_3 = battles[identified == 2]
plt.scatter(battles[:,0], battles[:,1], s=100, c=identified)
plt.show()
```

Scipy.interpolate

- For interpolation tasks, Scipy.interpolate can be used:

```
from scipy.interpolate import interp1d
measured_time = np.linspace(0, 1, 10)
noise = (np.random.random(10)*2 - 1) * 1e-1
measures = np.sin(2 * np.pi * measured_time) + noise
linear_interp = interp1d(measured_time, measures)
```

Scipy.interpolate

- The interpolation can be implemented linearly, quadratically or cubicly:

```
from scipy.interpolate import interp1d
measured_time = np.linspace(0, 1, 10)
noise = (np.random.random(10)*2 - 1) * 1e-1
measures = np.sin(2 * np.pi * measured_time) + noise
cubic_interp = interp1d(measured_time, measures,
kind='cubic')
cubic_results = cubic_interp(computed_time)
```

References

1 Wentworth, P., Elkner, J., Downey, A.B., Meyers, C. (2014). How to Think Like a Computer Scientist: Learning with Python (3nd edition).
2 Pilgrim, M. (2014). Dive into Python 3 by. Free online version: DiveIntoPython3.org ISBN: 978-1430224150.
3 Summerfield, M. (2014) Programming in Python 3 2nd ed (PIP3) : - Addison Wesley ISBN: 0-321-68056-1.
4 Jones E, Oliphant E, Peterson P, et al. SciPy: Open Source Scientific Tools for Python, 2001-, http://www.scipy.org/.
5 Millman, K.J., Aivazis, M. (2011). Python for Scientists and Engineers, Computing in Science & Engineering, 13, 9-12.
6 John D. Hunter (2007). Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering, 9, 90-95.
7 Travis E. Oliphant (2007). Python for Scientific Computing, Computing in Science & Engineering, 9, 10-20.
8 Goodrich, M.T., Tamassia, R., Goldwasser, M.H. (2013). Data Structures and Algorithms in Python, Wiley.
9 http://www.diveintopython.net/
10 https://docs.python.org/3/tutorial/
11 http://www.python-course.eu
12 https://developers.google.com/edu/python/
13 http://learnpythonthehardway.org/book/