ee21b154

Varsha S P, EE21B154 <ee21b154@smail.iitm.ac.in>

February 17, 2023

1 Assignment 3

1.1 Plotting and Visualization

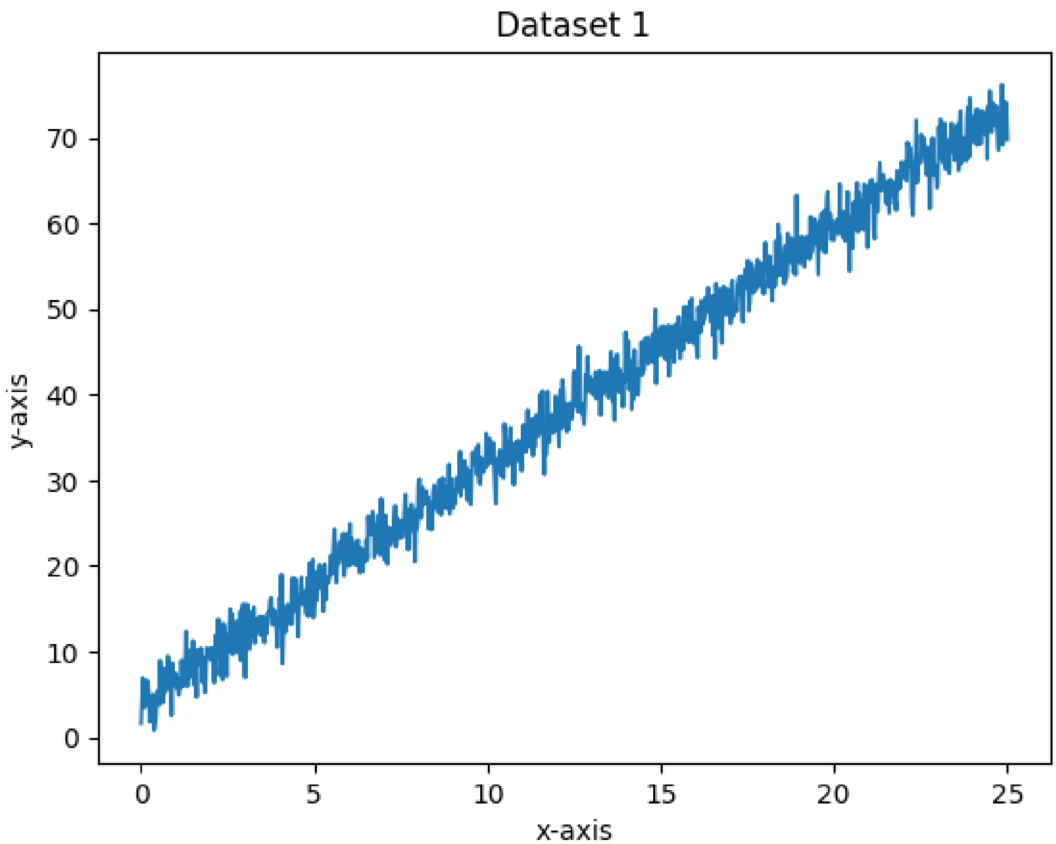
In this assignment we are trying to plot a reasonable curve for a bunch of known and unknown dataset. We are trying to predict the possible and use the `curve_fit` fuction to estimate if the predicted graph is accurate.

1.1.1 For Dataset 1

[92]:
```
import numpy as np
import matplotlib.pyplot as plt
import statistics
from scipy.optimize import curve_fit
%matplotlib inline
```

[93]:
```
np_numbers = np.loadtxt(r'dataset1.txt', skiprows=0, delimiter=' ')
#Seperating the data for x-axis and y-axis
x=[]
for i in range(len(np_numbers)):
    x.append(np_numbers[i][0])
y=[]
for i in range(len(np_numbers)):
    y.append(np_numbers[i][1])

x=np.array(x)
y=np.array(y)

plt.plot(x,y) #This is how the dataset looks
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('Dataset 1')
```

[93]:
```
Text(0.5, 1.0, 'Dataset 1')
```

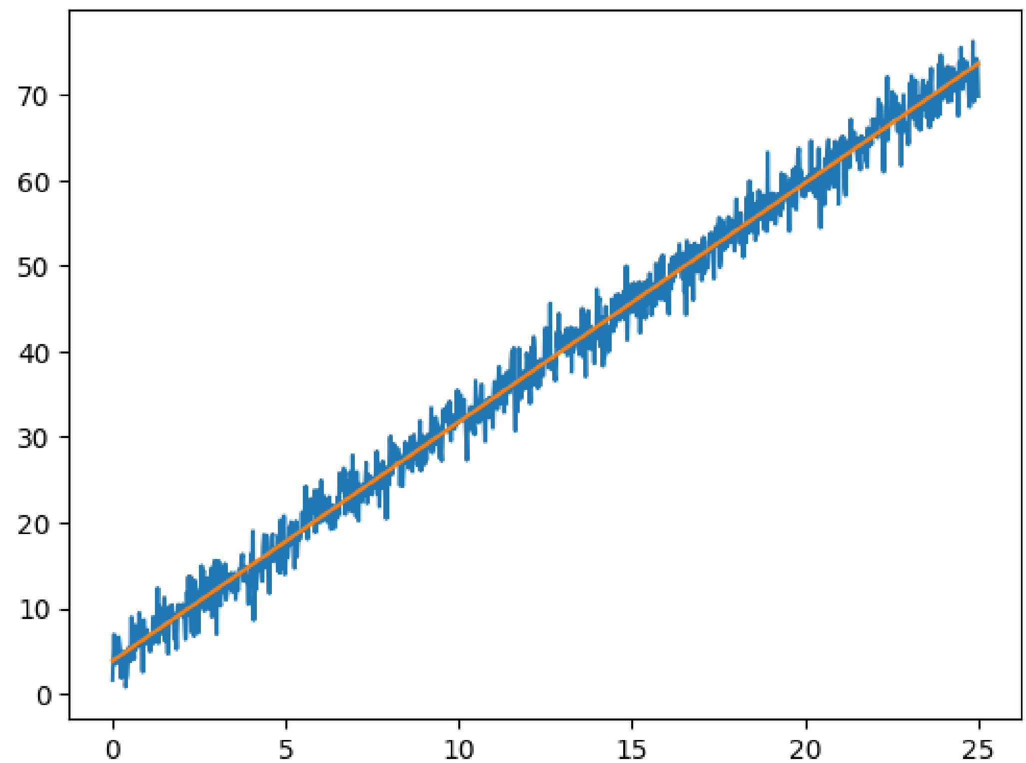


Using `linalg.lstsq` for this dataset as we know that it's a straight line, with 2 parameters varying linearly with x.

[94]:
```
M = np.column_stack([x, np.ones(len(x))])
# Using the lstsq function to solve for p_1 and p_2
(p1, p2), _, _, _ = np.linalg.lstsq(M, y, rcond=None)
print(f"The estimated equation is {p1} t + {p2}")
def stline(x, m, c):
    z=[]
    for i in range(len(x)):
        z.append(p1*x[i]+p2) #defining a straight line function
    return z

yest = stline(x, p1, p2) #Plotting the obtained linear curve through lstsq
plt.plot(x,y,x,yest )
```

```
The estimated equation is 2.791124245414918 t + 3.848800101430742
```

[94]:
```
[<matplotlib.lines.Line2D at 0x7f16a99d6f40>,
 <matplotlib.lines.Line2D at 0x7f16a99d67f0>]
```

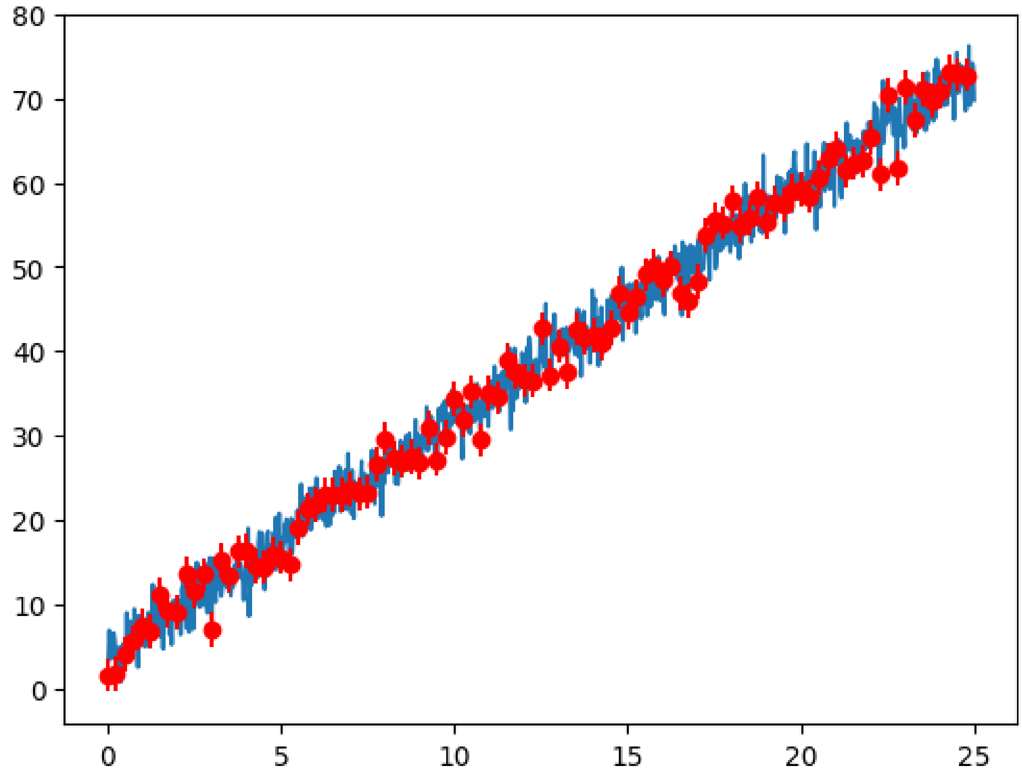


The error bar for this can be plotted as the standard deviation of the estimated value and actual value

[95]:
```
y_err=y-yest
plt.plot(x,y)
plt.errorbar(x[::10], y[::10], yerr=np.std(y_err), fmt='ro')
%timeit np.linalg.lstsq(M, y) #timeit function
```

```
<magic-timeit>:1: FutureWarning: `rcond` parameter will change to the default of
machine precision times ``max(M, N)`` where M and N are the input matrix
dimensions.
To use the future default and silence this warning we advise to pass
`rcond=None`, to keep using the old, explicitly pass `rcond=-1`.
```

```
35.4 µs ± 8 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

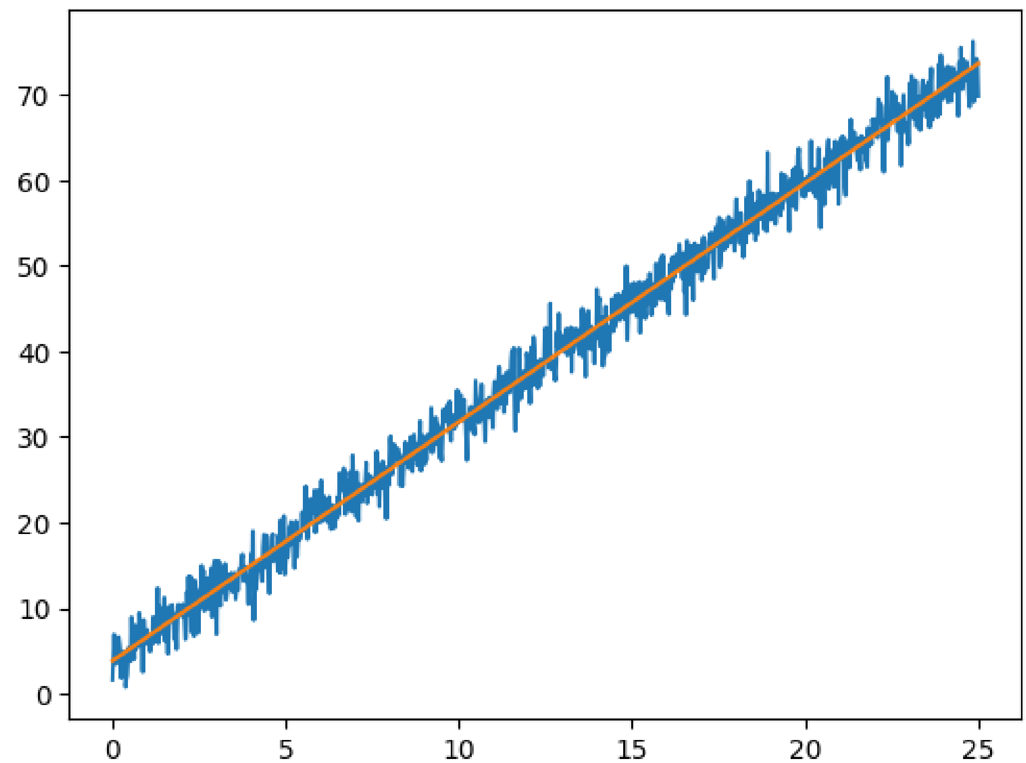


We are using `curve_fit` on the linear fuction to check which one of these methods of drawing a graph might be feasible

[96]:
```
from scipy.optimize import curve_fit
(zp1, zp2), pcov = curve_fit( stline,x, y)
print(f"Estimated function: exp(-{zp1}t) + {zp2}")

zest = stline(x, zp1, zp2)
plt.plot(x, y, x, zest)
```

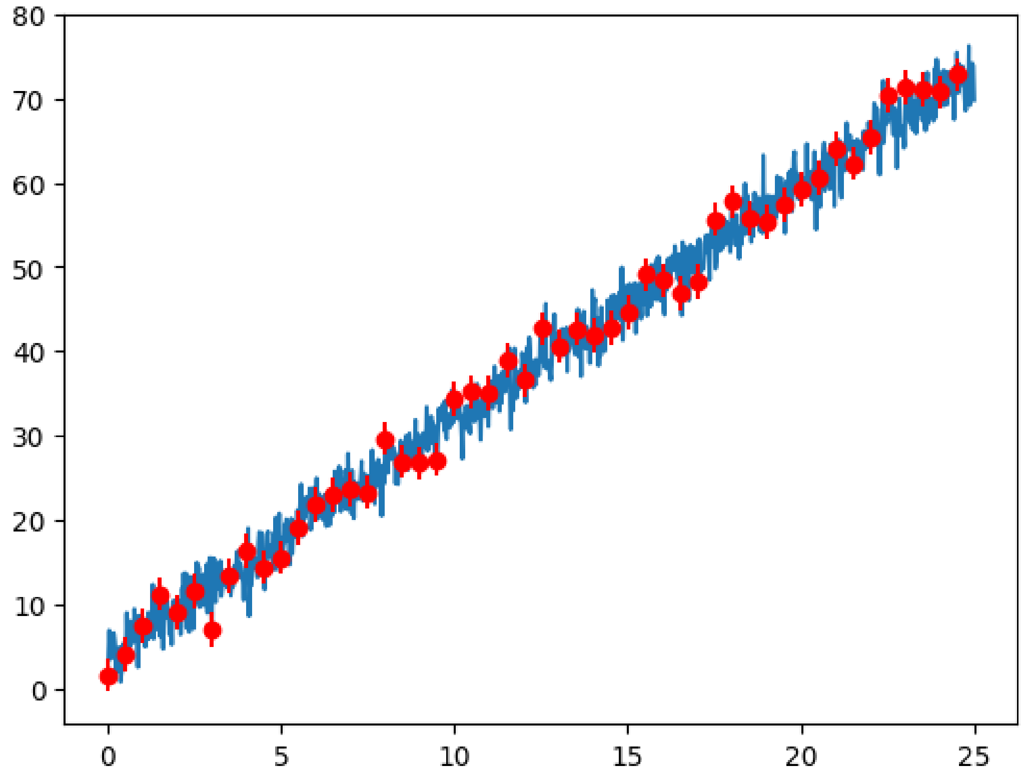
```
Estimated function: exp(-1.0t) + 1.0
```

[96]:
```
[<matplotlib.lines.Line2D at 0x7f16a90b5790>,
 <matplotlib.lines.Line2D at 0x7f16a90b5d30>]
```



[97]:
```
cuy_err=y-zest

plt.plot(x, y)
plt.errorbar(x[::20], y[::20], np.std(cuy_err), fmt='ro') #erroe bar for when␣
↪curvefit used

%timeit curve_fit(stline,x, y) #timeit function
```

```
1.42 ms ± 32.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

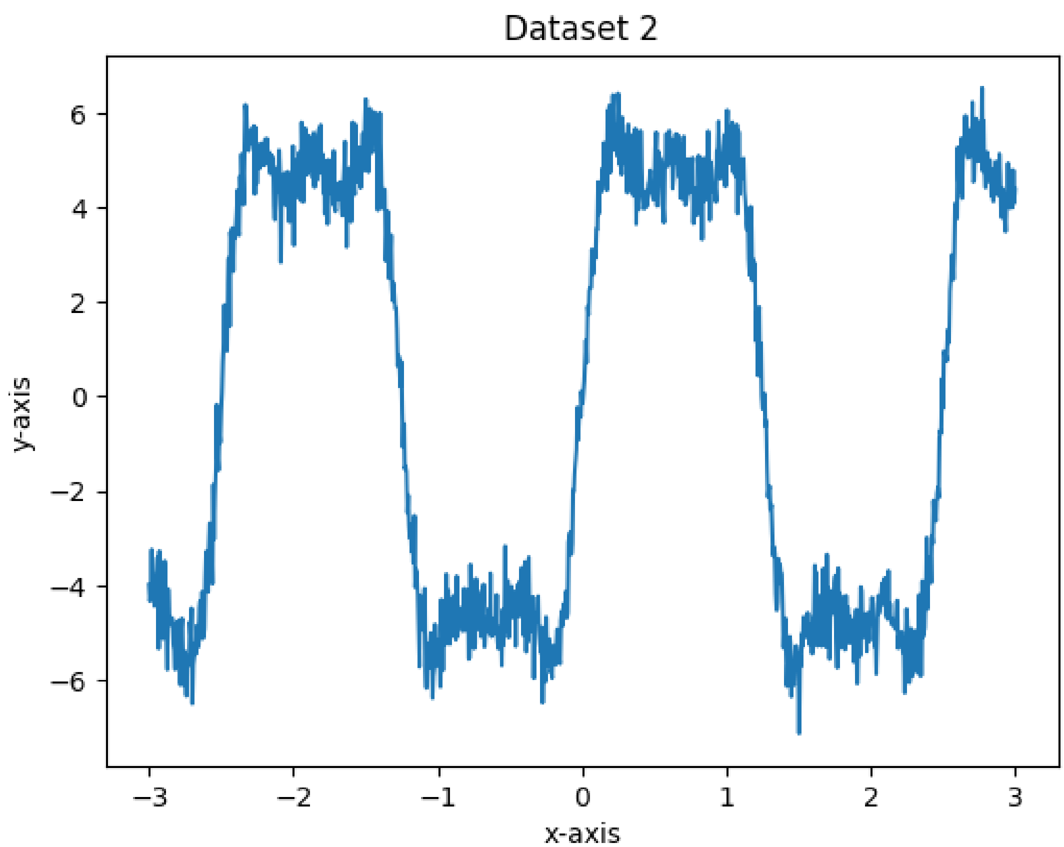


Clearly we can see that when lstsq was used for plotting cure for a known linear graph, it was much faster compared to using curvefit(which is mostly used for non-linear functions)

1.1.2 For Dataset 2

[98]:
```
np_numbers = np.loadtxt(r'dataset2.txt', skiprows=0, delimiter=' ')
#Seperating the data for x-axis and y-axis
x=[]
for i in range(len(np_numbers)):
    x.append(np_numbers[i][0])
y=[]
for i in range(len(np_numbers)):
    y.append(np_numbers[i][1])

x=np.array(x)
y=np.array(y)

plt.plot(x,y) #This is how the dataset looks
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('Dataset 2')
```

[98]: Text(0.5, 1.0, 'Dataset 2')



Fourier series equation

$$f(x) = \frac{4}{\pi} \sum_{n=1,3,5,\ldots}^{\infty} \frac{1}{n} \sin\left(\frac{n\pi x}{L}\right).$$

The given Dataset is said to be a Fourier series. Hence, we define functions for n=1,3,5… and see the best fit for the given data.

[99]:
```
#Fourier equations
def fourier_1(x,a,l):
    return a*(4/np.pi)*(np.sin(np.pi*x/l)) #For n=1
def fourier_3(x,a,l):
    return a*(4/np.pi)*((np.sin(np.pi*x/l))+((np.sin((np.pi*x*3)/l))/3)) #For n=3
def fourier_5(x,a,l):
```

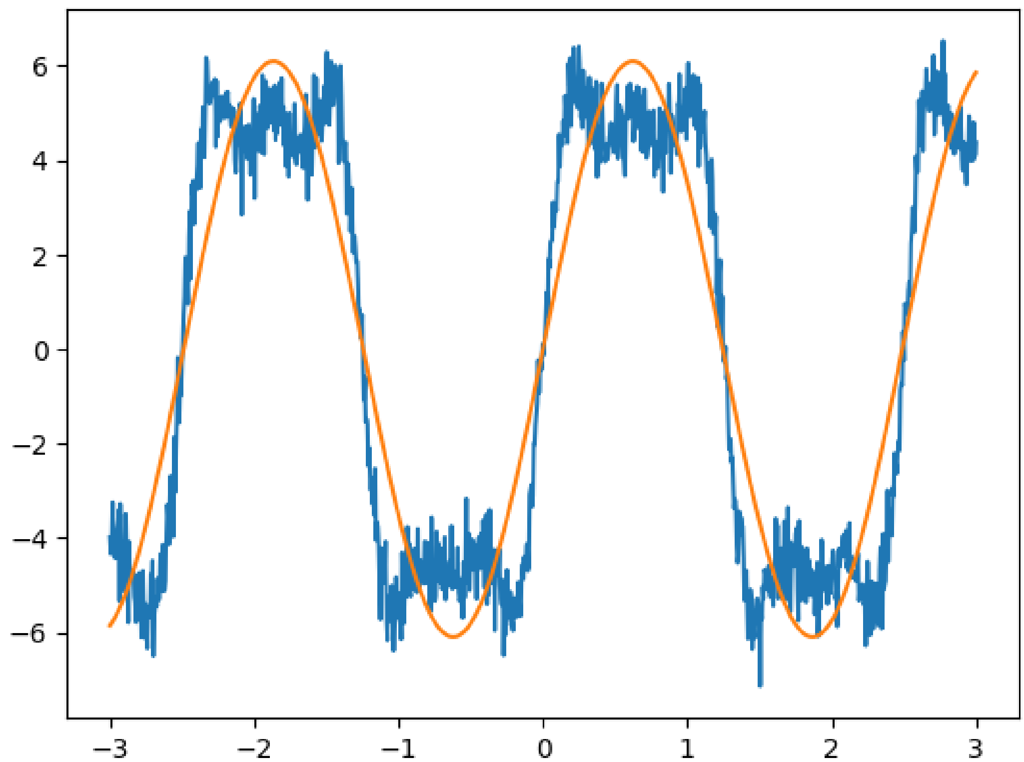
```
    return a*(4/np.pi)*((np.sin(np.pi*x/l))+((np.sin((np.pi*x*3)/l))/3)+((np.
↪sin((np.pi*x*5)/l))/5)) #For n=5
```

For n=1

[100]:
```
from scipy.optimize import curve_fit
(zp1, zp2), pcov = curve_fit( fourier_1,x, y)
print(f"Estimated function: exp(-{zp1}t) + {zp2}")
zest = fourier_1(x, zp1, zp2)
plt.plot(x, y, x, zest)
```

```
Estimated function: exp(-4.784329317633403t) + 1.2448266475990044
```

[100]:
```
[<matplotlib.lines.Line2D at 0x7f16a9846310>,
 <matplotlib.lines.Line2D at 0x7f16a9846370>]
```

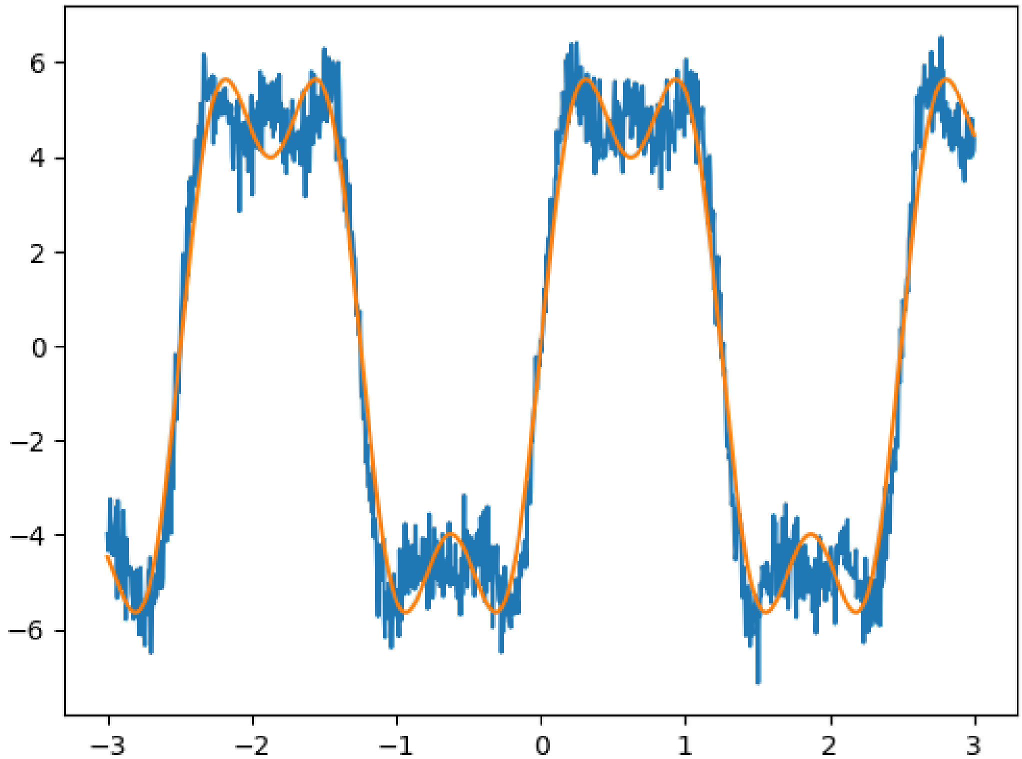


For n=3

[101]:
```
from scipy.optimize import curve_fit
(zp1, zp2), pcov = curve_fit( fourier_3,x, y)
print(f"Estimated function: exp(-{zp1}t) + {zp2}")
zest = fourier_3(x, zp1, zp2)
plt.plot(x, y, x, zest)
```

```
Estimated function: exp(-4.695927291303469t) + 1.2473976512331715
```

[101]: [<matplotlib.lines.Line2D at 0x7f16a8f885e0>,
 <matplotlib.lines.Line2D at 0x7f16a8f88640>]

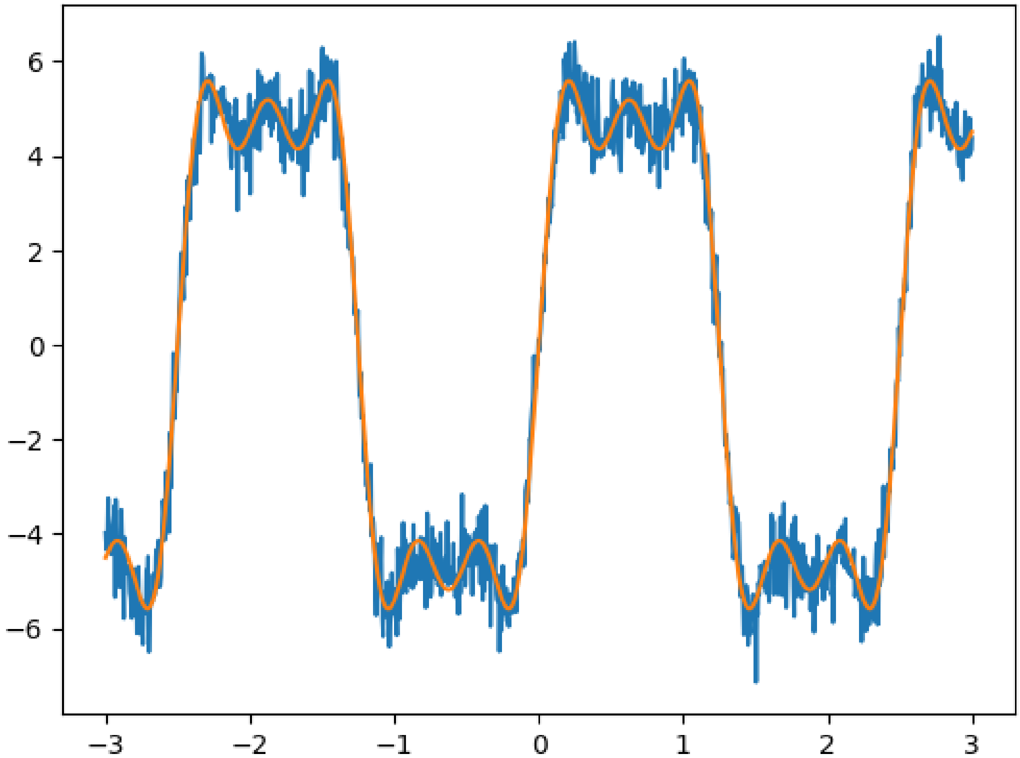


For n=5

[102]:
```
from scipy.optimize import curve_fit
(zp1, zp2), pcov = curve_fit( fourier_5,x, y)
print(f"Estimated function: exp(-{zp1}t) + {zp2}")
zest = fourier_5(x, zp1, zp2)
plt.plot(x, y, x, zest)
```

```
Estimated function: exp(-4.695620880486891t) + 1.2506520749990966
```

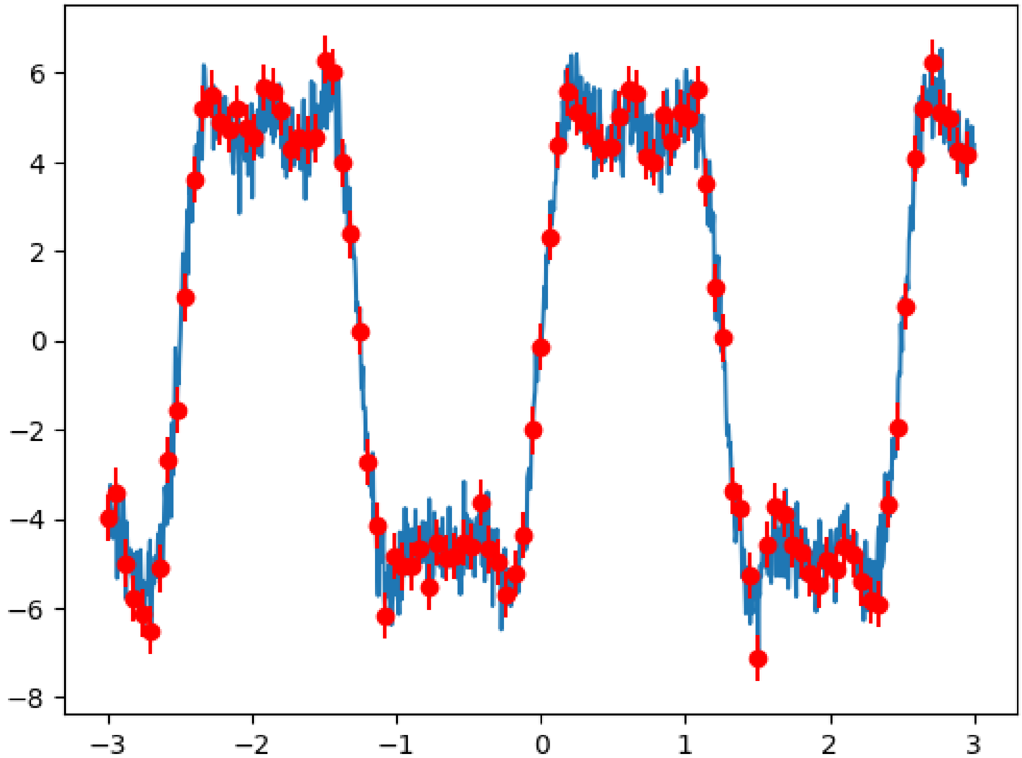
[102]: [<matplotlib.lines.Line2D at 0x7f16a8effc70>,
 <matplotlib.lines.Line2D at 0x7f16a8effcd0>]



We see that at n=5, the fourier graph fits well with given data set. It now makes sense to use `fourier_5` for `curve_fit`

[103]:
```
cuy_err=y-zest

plt.plot(x, y)
plt.errorbar(x[::10], y[::10], np.std(cuy_err), fmt='ro') #erroe bar for when␣
↪curvefit used
```

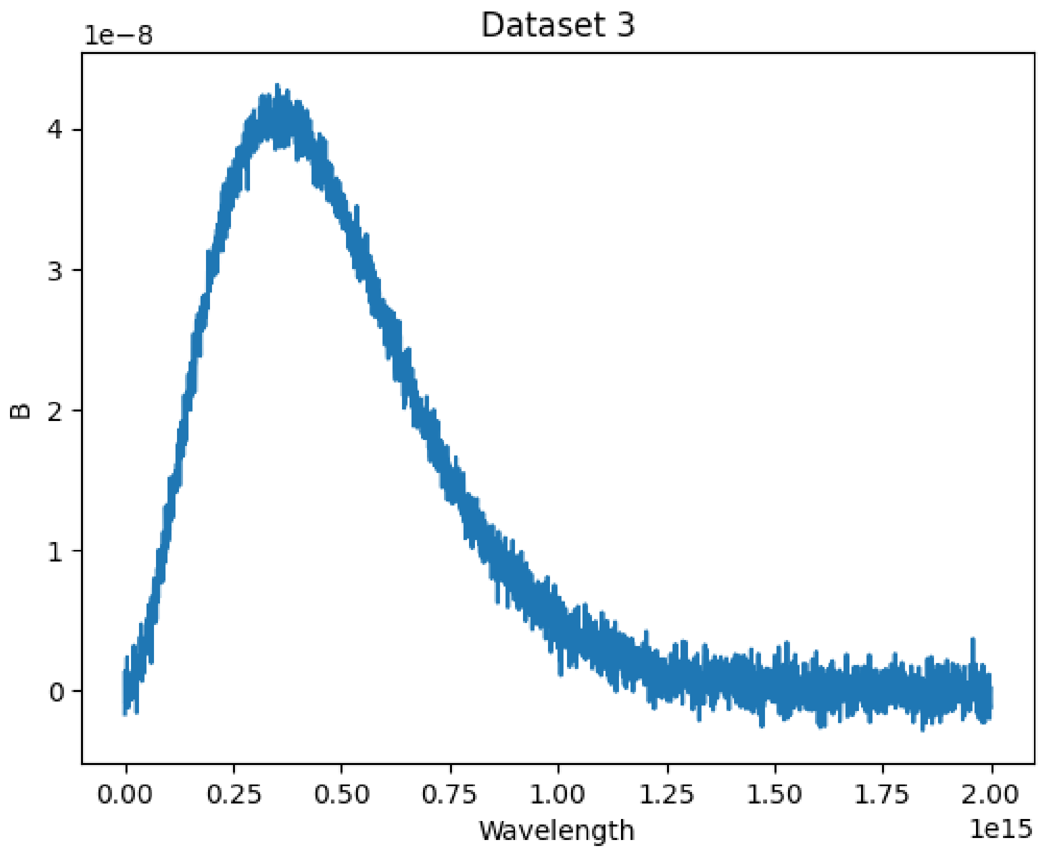
[103]:
```
<ErrorbarContainer object of 3 artists>
```



1.1.3 For Dataset 3

[104]:
```
np_numbers = np.loadtxt(r'dataset3.txt', skiprows=0, delimiter=' ')
#Seperating the data for x-axis and y-axis
x=[]
for i in range(len(np_numbers)):
    x.append(np_numbers[i][0])
y=[]
for i in range(len(np_numbers)):
    y.append(np_numbers[i][1])

x=np.array(x)
y=np.array(y)

plt.plot(x,y) #This is how the dataset looks
plt.xlabel('Wavelength')
plt.ylabel('B')
plt.title('Dataset 3')
```

[104]:
```
Text(0.5, 1.0, 'Dataset 3')
```



The given dataset is said to follow Planck's law.

$$B(\nu, T) = \frac{2h\nu^3}{c^2} \frac{1}{\frac{hv}{ek_B T} - 1}$$

Hence, we try to define a function having frequency, temperature and planck's constant.

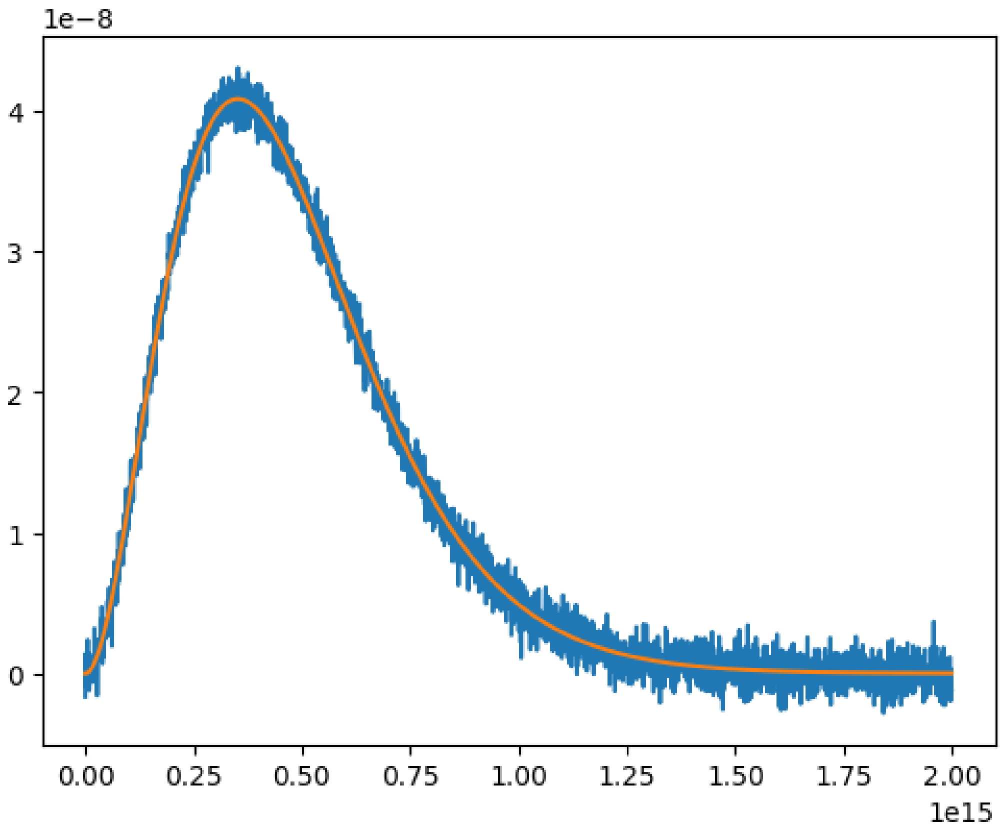
[105]:
```
import math as m
def planck(f,h,T):
    c=3.0*(m.pow(10,8))
    kb=1.38*(m.pow(10,-23))

    return ((2*h*(f*f*f))/(c*c)*(1/((np.exp((h*f)/(kb*T))-1))))
```

Now, you plot the curve using `curve_fit`.

[106]:
```
from scipy.optimize import curve_fit
(zp1, zp2), pcov = curve_fit( planck,x, y,p0=[6*m.pow(10,-34),300])
print(f"Estimated function: exp(-{zp1}t) + {zp2}")
zest = planck(x, zp1, zp2)
plt.plot(x, y, x, zest)
```

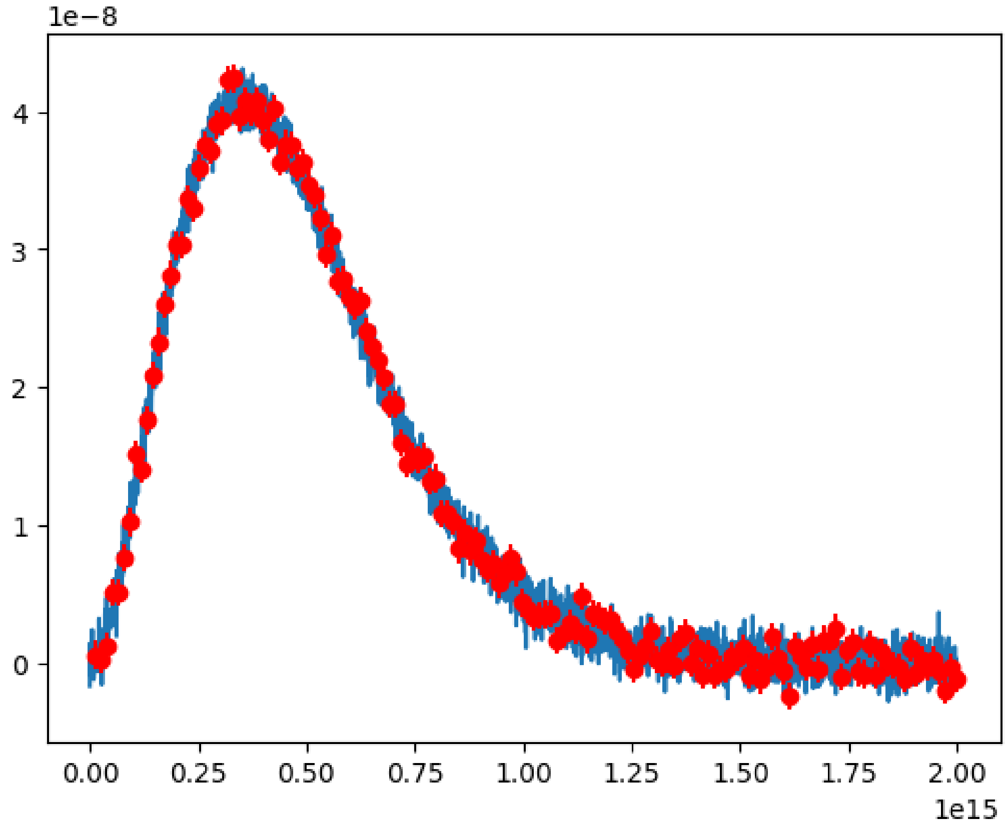
```
Estimated function: exp(-6.643229758011344e-34t) + 6011.36152125128
```

[106]:
```
[<matplotlib.lines.Line2D at 0x7f16a8d3f340>,
 <matplotlib.lines.Line2D at 0x7f16a8d3f3a0>]
```



[107]:
```
cuy_err=y-zest

plt.plot(x, y)
plt.errorbar(x[::20], y[::20], np.std(cuy_err), fmt='ro') #erroe bar for when␣
↪curvefit used
```

[107]:
```
<ErrorbarContainer object of 3 artists>
```



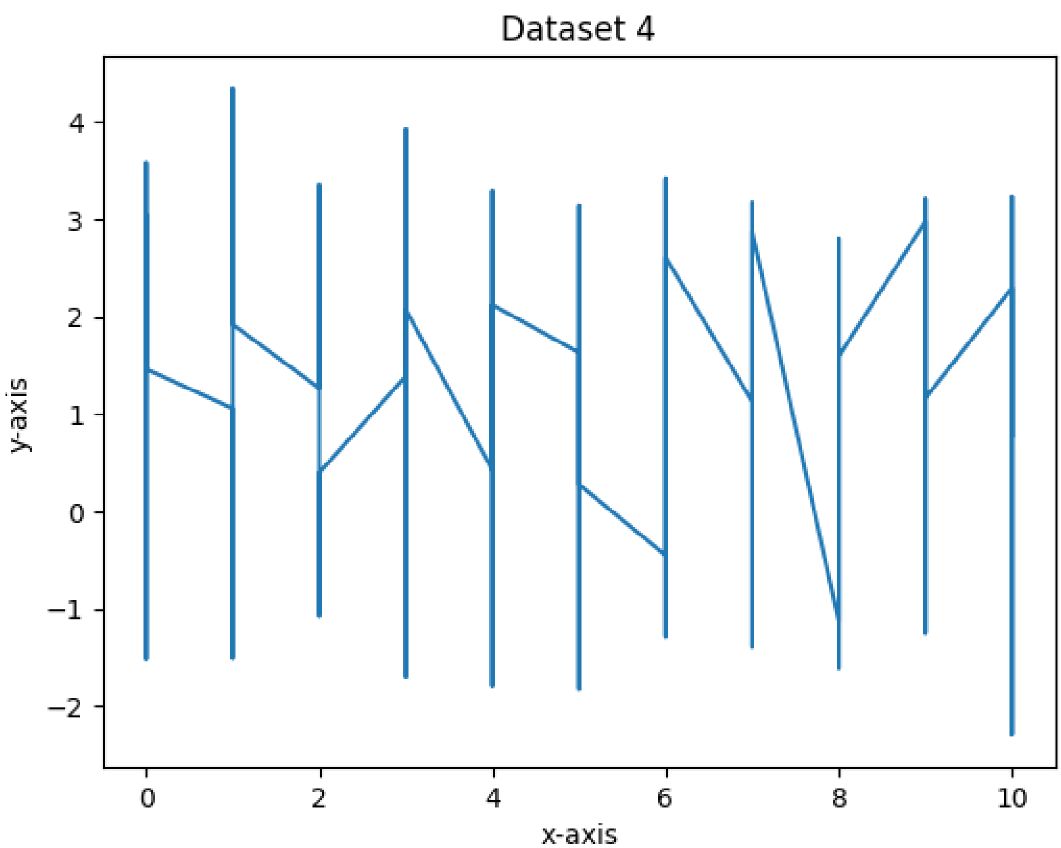
1.1.4 For Dataset 4

We do not know the nature of the the given dataset, hence we plot it using random methods to find a sequence.

[108]:
```
np_numbers = np.loadtxt('dataset4.txt', skiprows=0, delimiter=' ')
#Seperating the data for x-axis and y-axis
x=[]
for i in range(len(np_numbers)):
    x.append(np_numbers[i][0])
y=[]
for i in range(len(np_numbers)):
    y.append(np_numbers[i][1])

x=np.array(x)
y=np.array(y)

plt.plot(x,y) #This is how the dataset looks
plt.xlabel('x-axis')
plt.ylabel('y-axis')
```

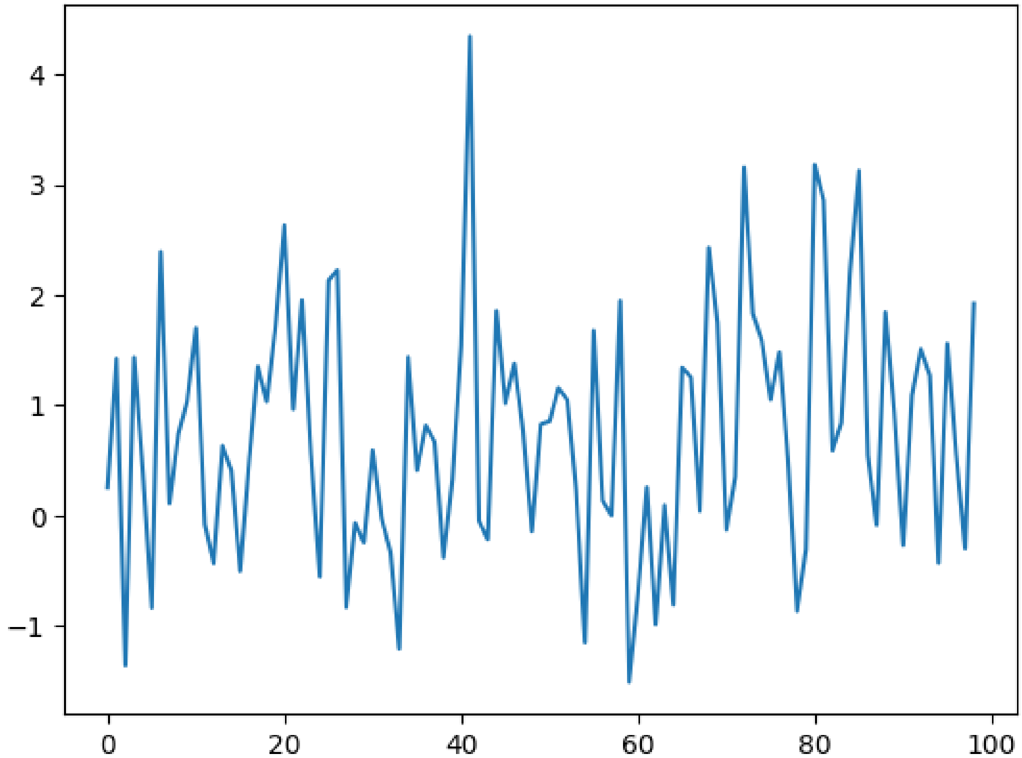
```
plt.title('Dataset 4')
```

[108]: Text(0.5, 1.0, 'Dataset 4')



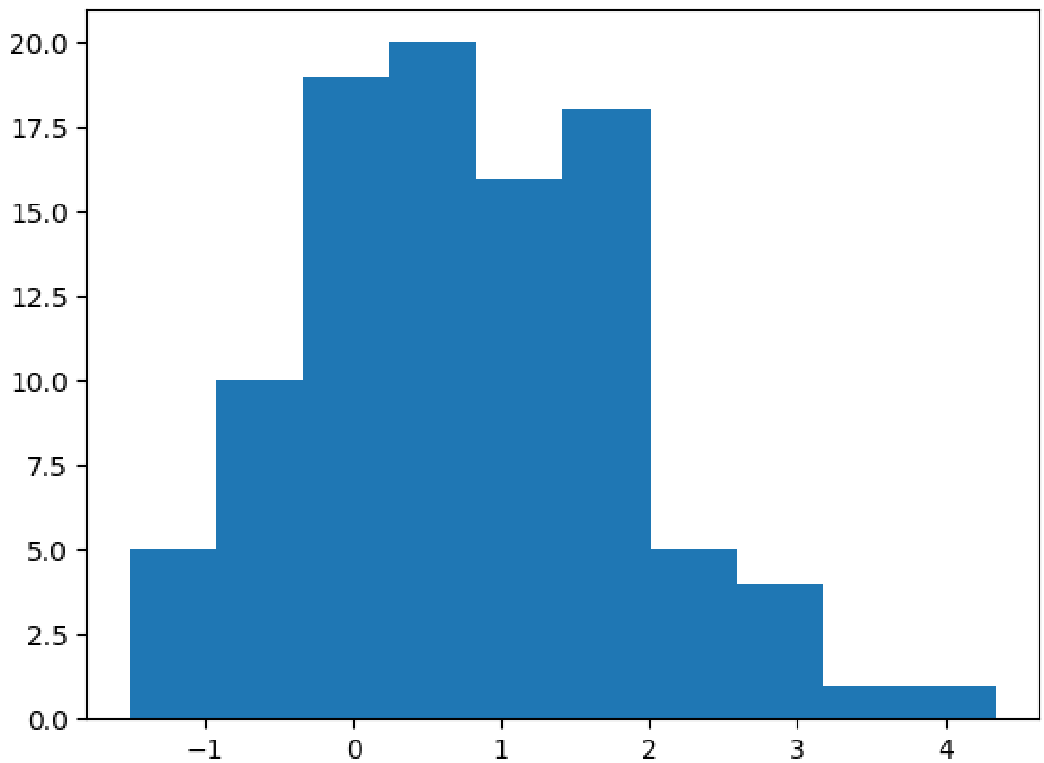
[109]:
```
plt.plot(y[51:150])
```

[109]: [<matplotlib.lines.Line2D at 0x7f16a8b72af0>]



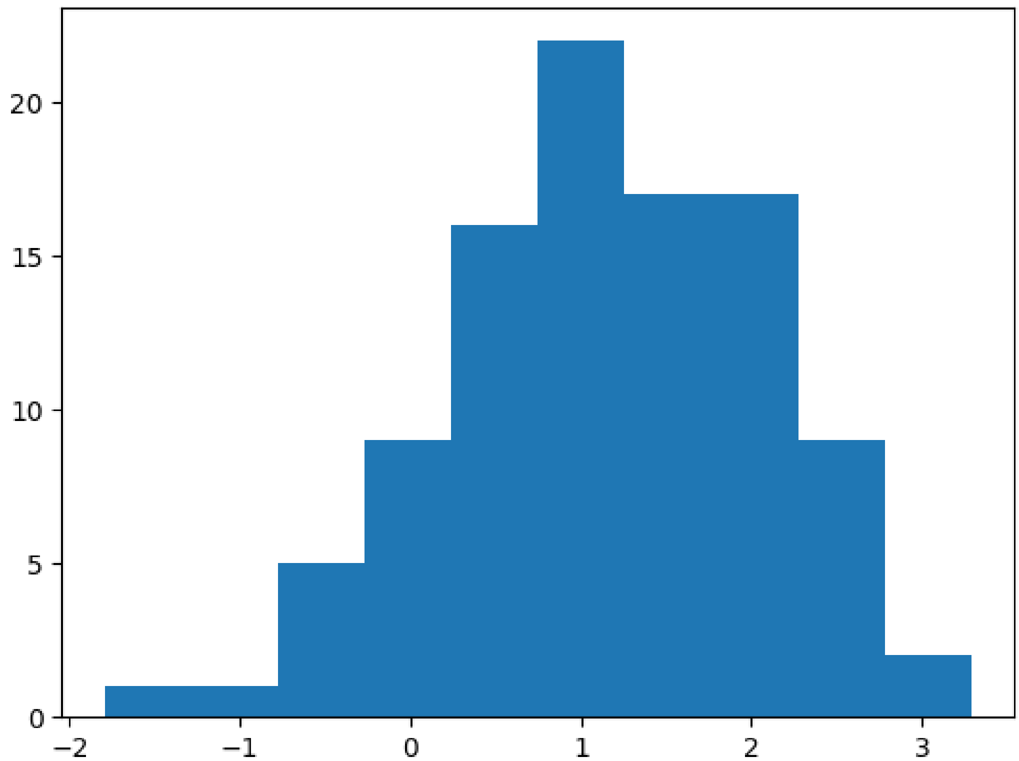
[110]:
```
plt.hist(y[51:150])
```

[110]:
```
(array([ 5., 10., 19., 20., 16., 18.,  5.,  4.,  1.,  1.]),
 array([-1.50648208, -0.921693  , -0.33690393,  0.24788515,  0.83267422,
         1.4174633 ,  2.00225237,  2.58704145,  3.17183053,  3.7566196 ,
         4.34140868]),
 <BarContainer object of 10 artists>)
```



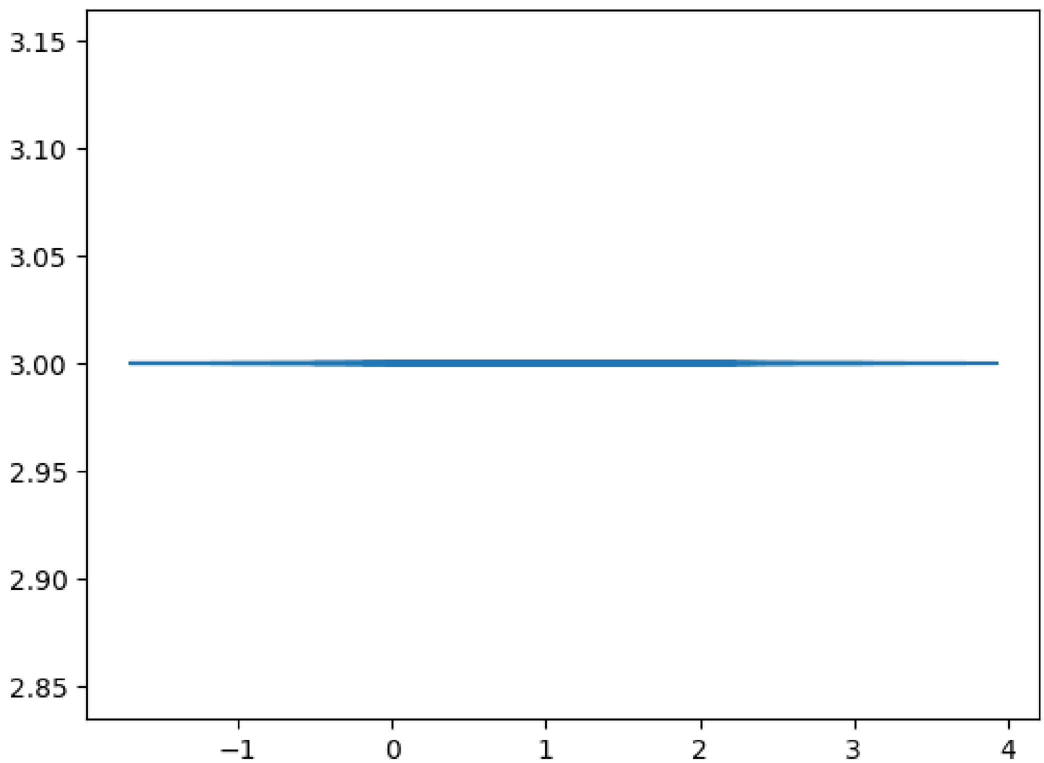
[111]:
```
plt.hist(y[351:450])
```

[111]:
```
(array([ 1.,  1.,  5.,  9., 16., 22., 17., 17.,  9.,  2.]),
 array([-1.79054622, -1.28246714, -0.77438805, -0.26630897,  0.24177011,
         0.74984919,  1.25792827,  1.76600735,  2.27408643,  2.78216552,
         3.2902446 ]),
 <BarContainer object of 10 artists>)
```



[112]:
```
plt.plot(y[251:350],x[251:350])
```

[112]:
```
[<matplotlib.lines.Line2D at 0x7f16a89fc5e0>]
```



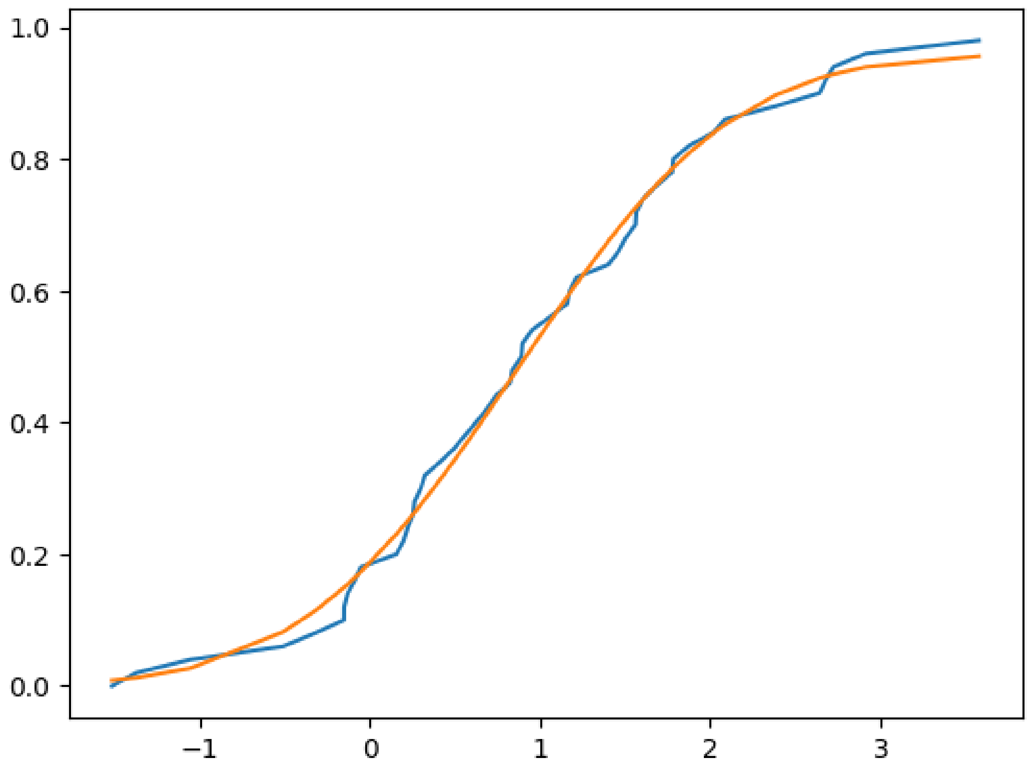
After plotting many graphs, we see that the values of y are peaking at a value at a certain point and is almost symmentrical(not exactly).

This reminds us of 'Bell curve'. Now, we define a function to return the cdf of the datapoints. And also we define an inverse cdf fuction for the fitted curve, to get data points

[113]:
```
import math
def inverse_cdf(x, mean, stddev, amplitude):
    return (norm.ppf(x/amplitude)+mean)*stddev

from scipy.stats import norm

def f_cdf(x, mean, stddev, amplitude):
    return amplitude * norm.cdf((x - mean)/stddev)

inp = np.arange(0, 1, 0.01)
x_pred = []
y_mean = []
y_pred = []
```

For x=0

[114]:
```
inpo=np.arange(0, 1, 0.02)
y_sort = sorted(y[1:51])

(mean, stddev, amplitude), cov = curve_fit(f_cdf, y_sort, inpo, p0 = [0.5, 1, 1])

plt.plot(y_sort, inpo)
plt.plot(y_sort, f_cdf(y_sort, mean, stddev, amplitude))


inverse_cdf_y = inverse_cdf(inpo, mean, stddev, amplitude)
inverse_cdf_y = [i for i in inverse_cdf_y if not math.isnan(i) and not math.
 ↪isinf(i)]
y_mean.append(np.mean(inverse_cdf_y))
x_pred = x_pred + [0 for j in range(len(inverse_cdf_y))]
y_pred = y_pred + inverse_cdf_y
```

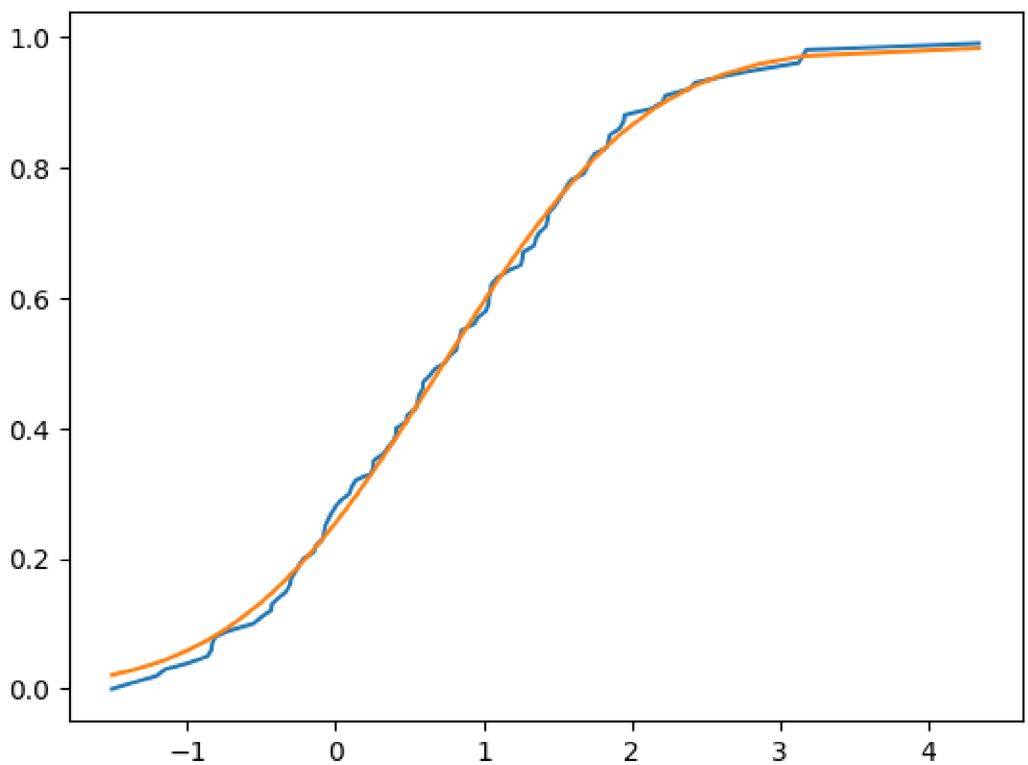


For x=1

[115]:
```
y_sort = sorted(y[51:151])

(mean, stddev, amplitude), cov = curve_fit(f_cdf, y_sort, inp, p0 = [0.5, 1, 1])

plt.plot(y_sort, inp)
```

```python
plt.plot(y_sort, f_cdf(y_sort, mean, stddev, amplitude))


inverse_cdf_y = inverse_cdf(inp, mean, stddev, amplitude)
inverse_cdf_y = [i for i in inverse_cdf_y if not math.isnan(i) and not math.
 ↪isinf(i)]
y_mean.append(np.mean(inverse_cdf_y))
print(y_mean)
x_pred = x_pred + [1 for j in range(len(inverse_cdf_y))]
y_pred = y_pred + inverse_cdf_y
```

```
[0.867968285719736, 0.8028339635925559]
```

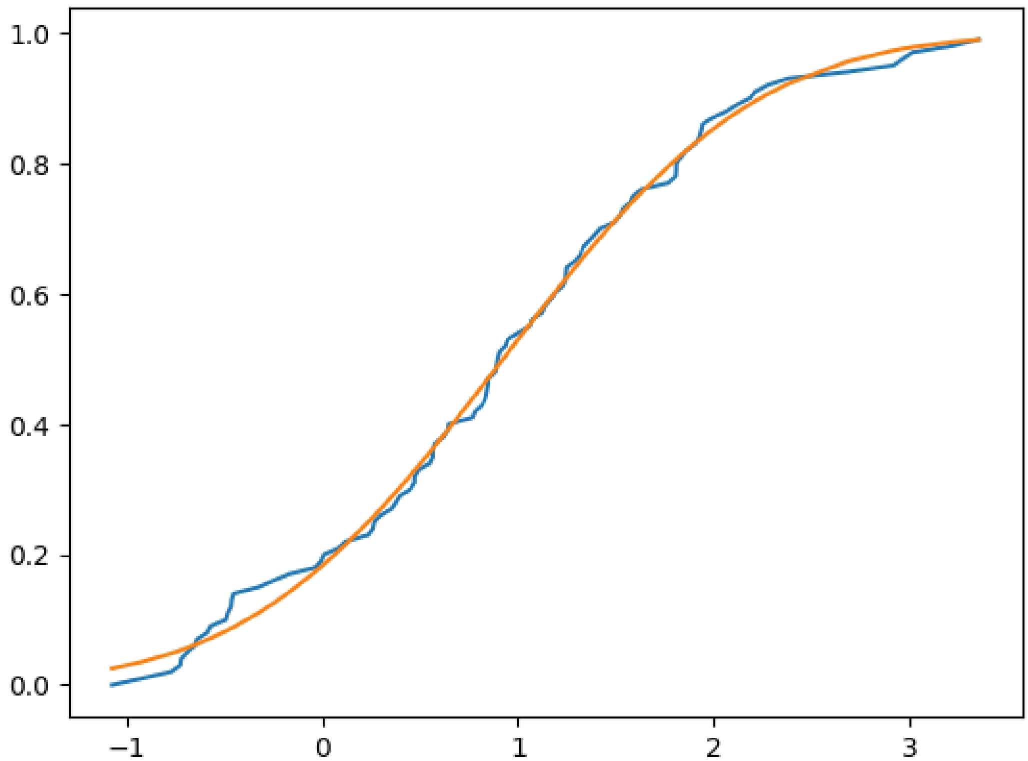


Similarly we plot it for eveyinteger value of x in the dataset

For x=2

[116]:
```python
y_sort = sorted(y[151:251])

(mean, stddev, amplitude), cov = curve_fit(f_cdf, y_sort, inp, p0 = [0.5, 1, 1])

plt.plot(y_sort, inp)
plt.plot(y_sort, f_cdf(y_sort, mean, stddev, amplitude))
```

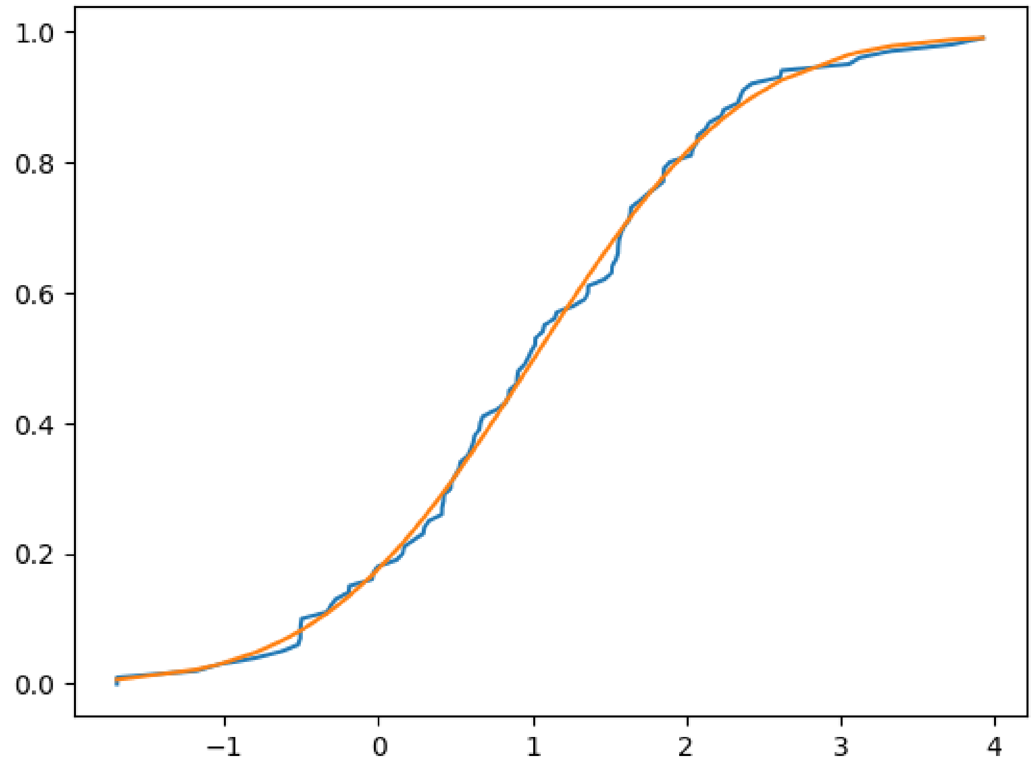
```
inverse_cdf_y = inverse_cdf(inp, mean, stddev, amplitude)
inverse_cdf_y = [i for i in inverse_cdf_y if not math.isnan(i) and not math.
↪isinf(i)]
y_mean.append(np.mean(inverse_cdf_y))

x_pred = x_pred + [2 for j in range(len(inverse_cdf_y))]
y_pred = y_pred + inverse_cdf_y
```



For x=3

[117]:
```
y_sort = sorted(y[251:351])

(mean, stddev, amplitude), cov = curve_fit(f_cdf, y_sort, inp, p0 = [0.5, 1, 1])

plt.plot(y_sort, inp)
plt.plot(y_sort, f_cdf(y_sort, mean, stddev, amplitude))


inverse_cdf_y = inverse_cdf(inp, mean, stddev, amplitude)
```

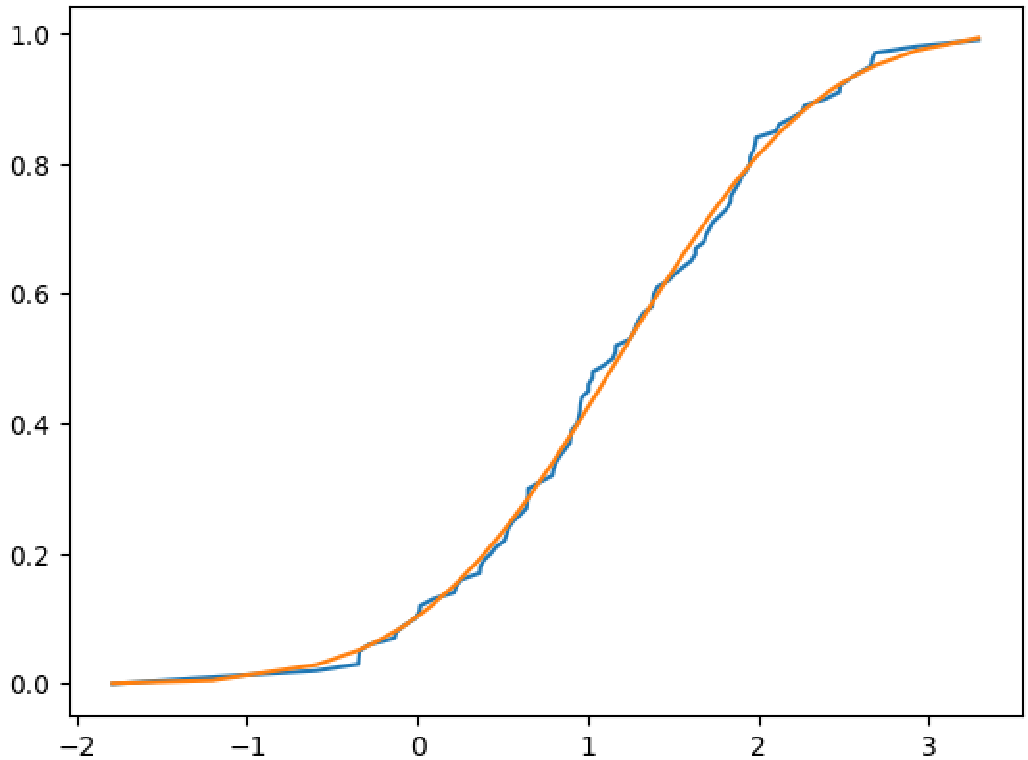
```
inverse_cdf_y = [i for i in inverse_cdf_y if not math.isnan(i) and not math.
 ↪isinf(i)]
y_mean.append(np.mean(inverse_cdf_y))

x_pred = x_pred + [3 for j in range(len(inverse_cdf_y))]
y_pred = y_pred + inverse_cdf_y
```



For x=4

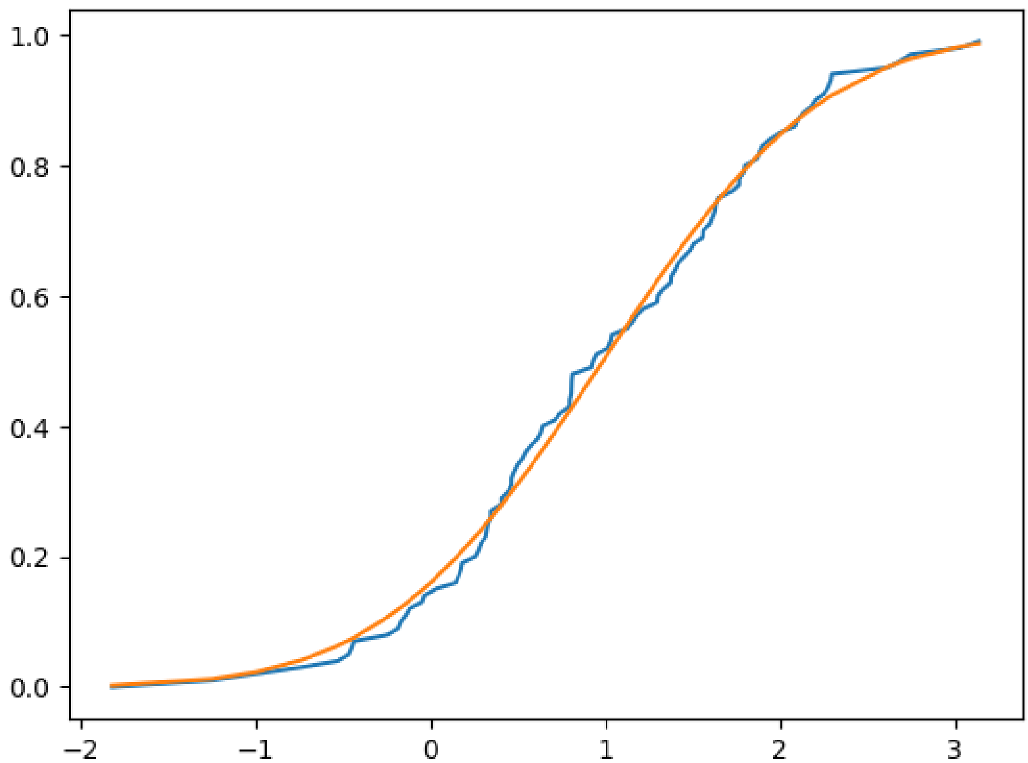
[118]:
```
y_sort = sorted(y[351:451])

(mean, stddev, amplitude), cov = curve_fit(f_cdf, y_sort, inp, p0 = [0.5, 1, 1])

plt.plot(y_sort, inp)
plt.plot(y_sort, f_cdf(y_sort, mean, stddev, amplitude))


inverse_cdf_y = inverse_cdf(inp, mean, stddev, amplitude)
inverse_cdf_y = [i for i in inverse_cdf_y if not math.isnan(i) and not math.
 ↪isinf(i)]
y_mean.append(np.mean(inverse_cdf_y))
```

```
x_pred = x_pred + [4 for j in range(len(inverse_cdf_y))]
y_pred = y_pred + inverse_cdf_y
```



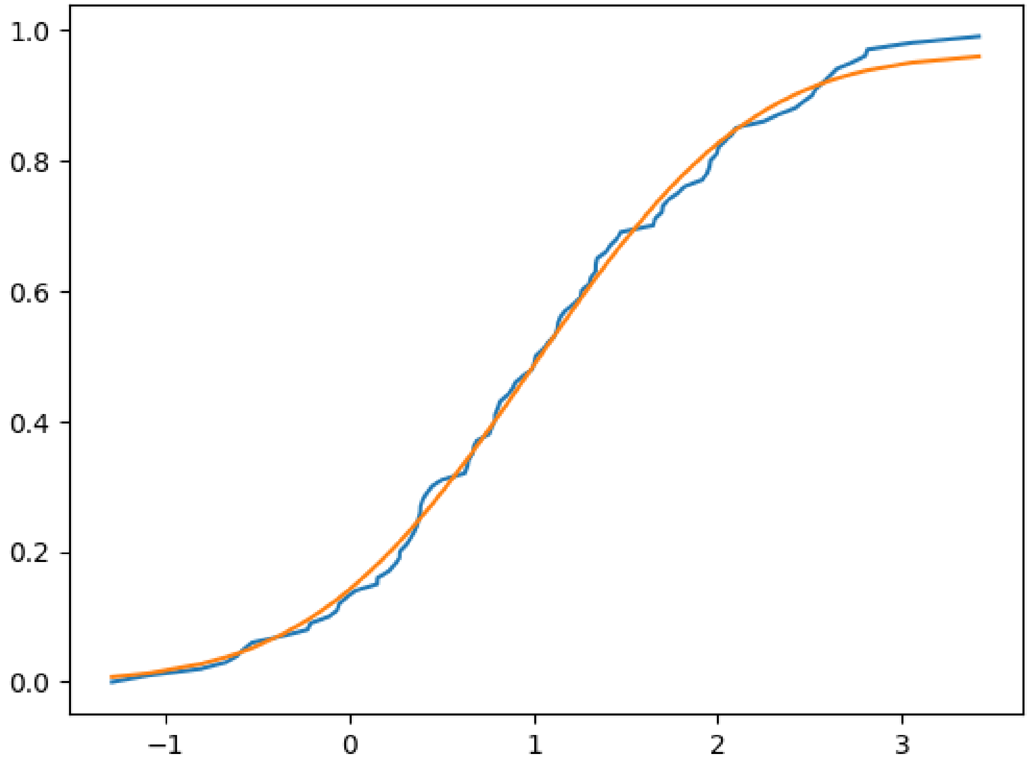
For x=5

[119]:
```
y_sort = sorted(y[451:551])

(mean, stddev, amplitude), cov = curve_fit(f_cdf, y_sort, inp, p0 = [0.5, 1, 1])

plt.plot(y_sort, inp)
plt.plot(y_sort, f_cdf(y_sort, mean, stddev, amplitude))


inverse_cdf_y = inverse_cdf(inp, mean, stddev, amplitude)
inverse_cdf_y = [i for i in inverse_cdf_y if not math.isnan(i) and not math.
 ↪isinf(i)]
y_mean.append(np.mean(inverse_cdf_y))

x_pred = x_pred + [5 for j in range(len(inverse_cdf_y))]
y_pred = y_pred + inverse_cdf_y
```



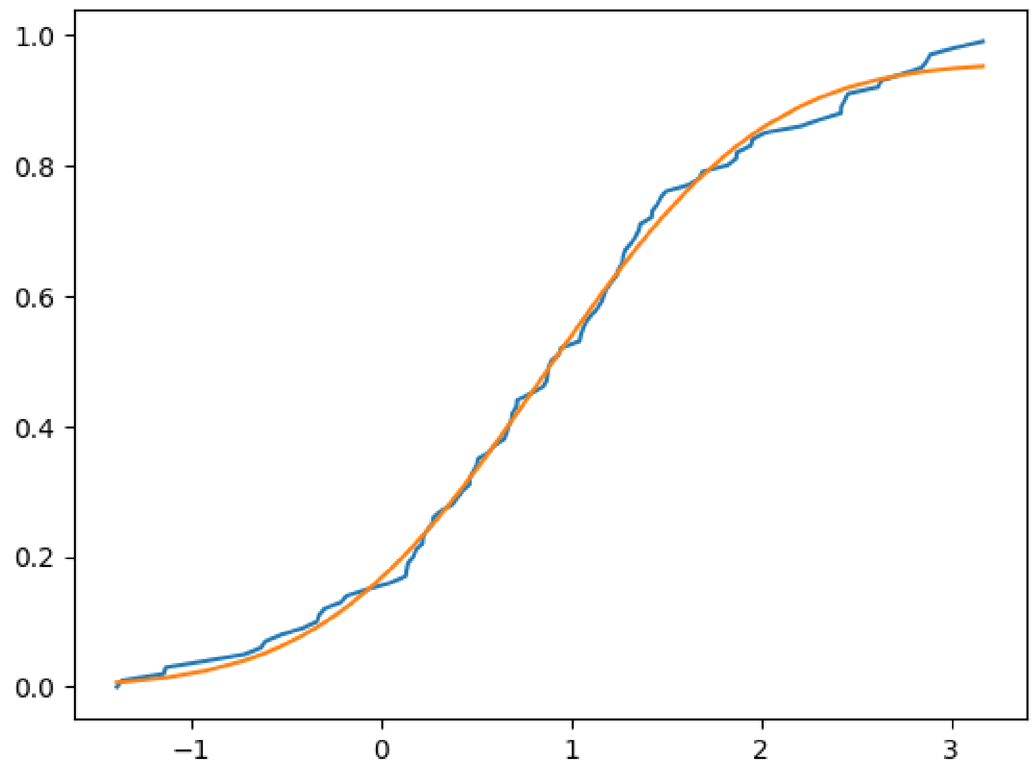
For x=6

[120]:
```
y_sort = sorted(y[551:651])

(mean, stddev, amplitude), cov = curve_fit(f_cdf, y_sort, inp, p0 = [0.5, 1, 1])

plt.plot(y_sort, inp)
plt.plot(y_sort, f_cdf(y_sort, mean, stddev, amplitude))


inverse_cdf_y = inverse_cdf(inp, mean, stddev, amplitude)
inverse_cdf_y = [i for i in inverse_cdf_y if not math.isnan(i) and not math.
 ↪isinf(i)]
y_mean.append(np.mean(inverse_cdf_y))

x_pred = x_pred + [6 for j in range(len(inverse_cdf_y))]
y_pred = y_pred + inverse_cdf_y
```



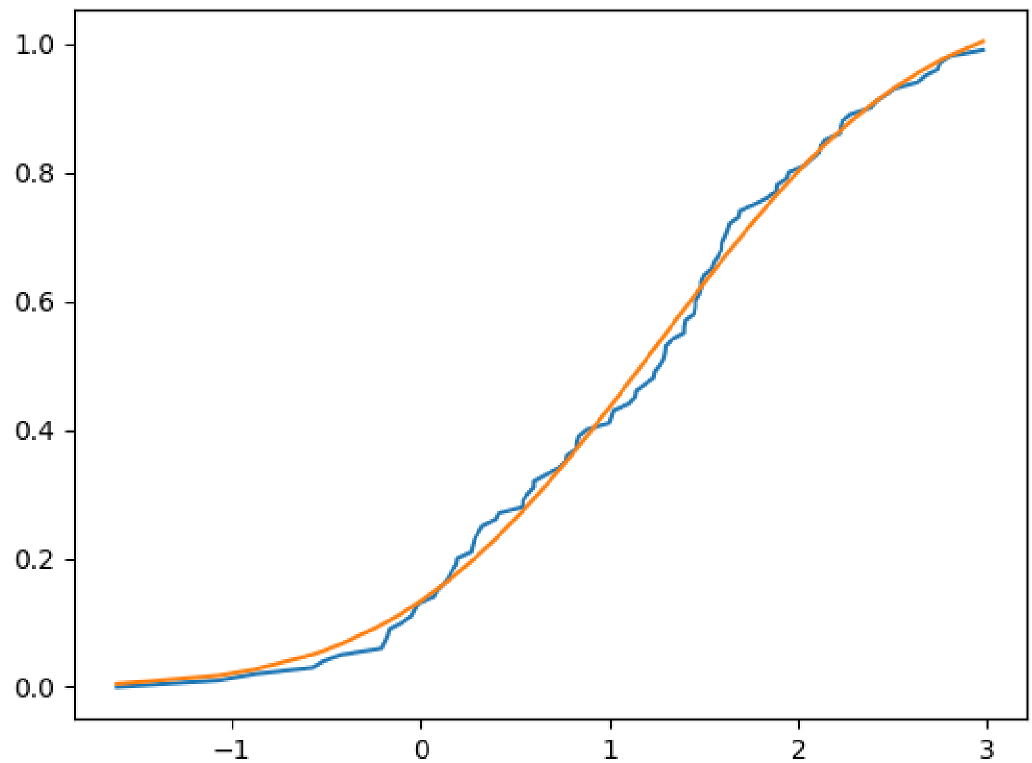
For x=7

[121]:
```
y_sort = sorted(y[651:751])

(mean, stddev, amplitude), cov = curve_fit(f_cdf, y_sort, inp, p0 = [0.5, 1, 1])

plt.plot(y_sort, inp)
plt.plot(y_sort, f_cdf(y_sort, mean, stddev, amplitude))


inverse_cdf_y = inverse_cdf(inp, mean, stddev, amplitude)
inverse_cdf_y = [i for i in inverse_cdf_y if not math.isnan(i) and not math.
 ↪isinf(i)]
y_mean.append(np.mean(inverse_cdf_y))

x_pred = x_pred + [7 for j in range(len(inverse_cdf_y))]
y_pred = y_pred + inverse_cdf_y
```



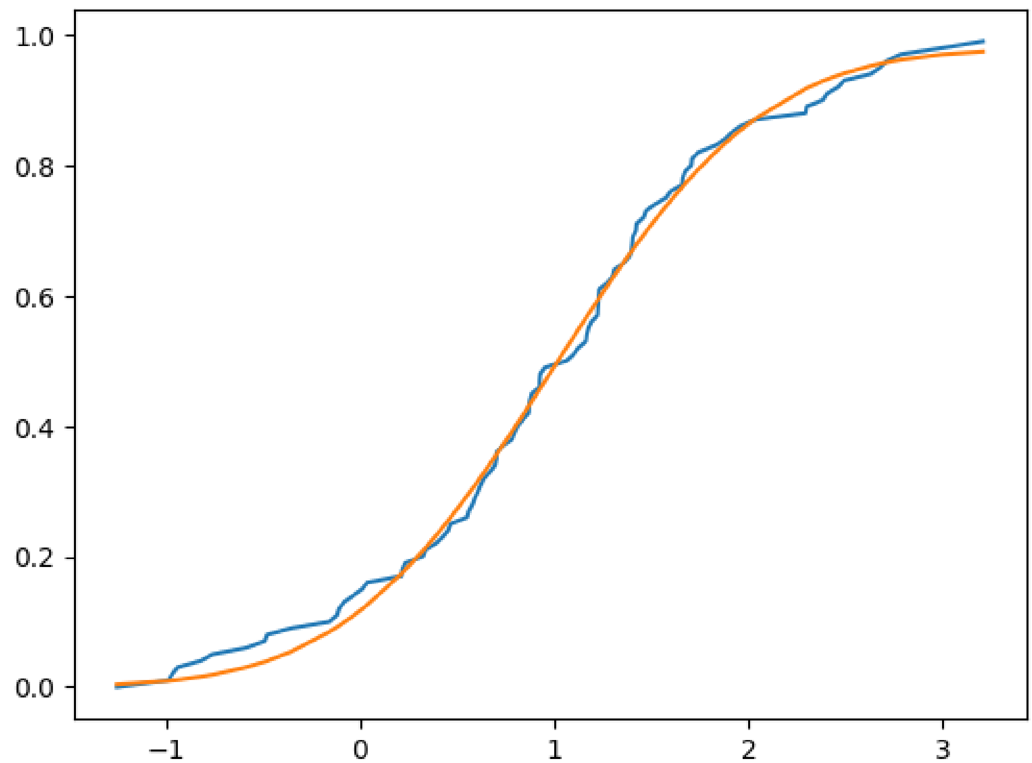
For x=8

[122]:
```
y_sort = sorted(y[751:851])

(mean, stddev, amplitude), cov = curve_fit(f_cdf, y_sort, inp, p0 = [0.5, 1, 1])

plt.plot(y_sort, inp)
plt.plot(y_sort, f_cdf(y_sort, mean, stddev, amplitude))


inverse_cdf_y = inverse_cdf(inp, mean, stddev, amplitude)
inverse_cdf_y = [i for i in inverse_cdf_y if not math.isnan(i) and not math.
 ↪isinf(i)]
y_mean.append(np.mean(inverse_cdf_y))

x_pred = x_pred + [8 for j in range(len(inverse_cdf_y))]
y_pred = y_pred + inverse_cdf_y
```



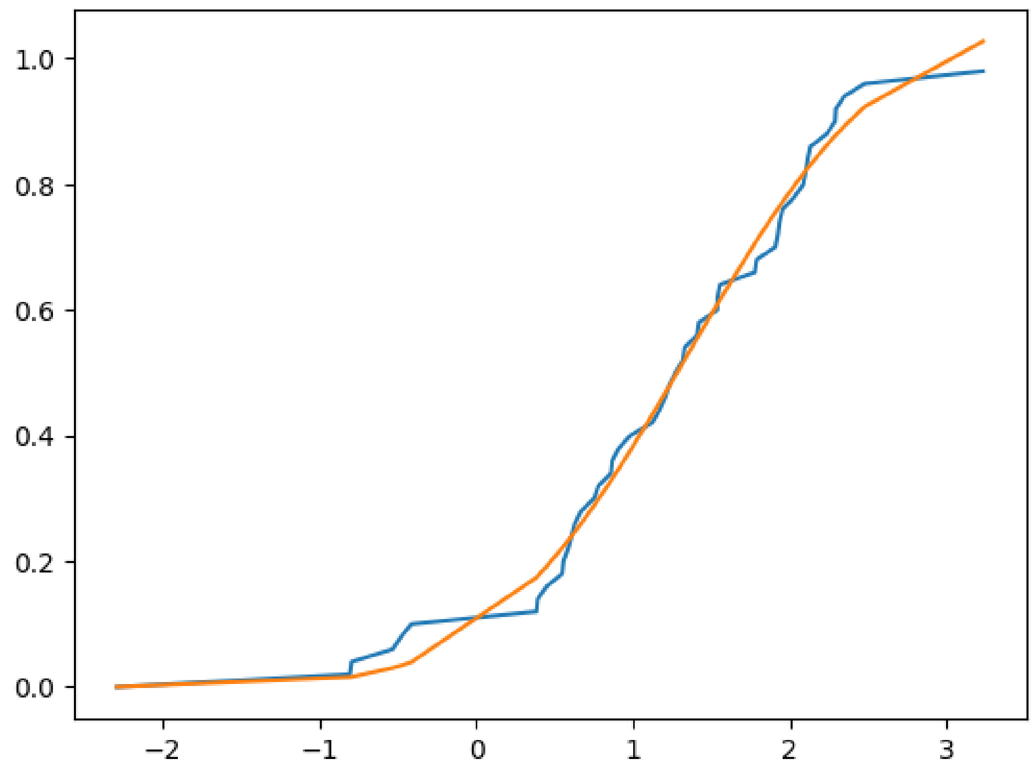
For x=9

[123]:
```
y_sort = sorted(y[851:951])

(mean, stddev, amplitude), cov = curve_fit(f_cdf, y_sort, inp, p0 = [0.5, 1, 1])

plt.plot(y_sort, inp)
plt.plot(y_sort, f_cdf(y_sort, mean, stddev, amplitude))


inverse_cdf_y = inverse_cdf(inp, mean, stddev, amplitude)
inverse_cdf_y = [i for i in inverse_cdf_y if not math.isnan(i) and not math.
 ↪isinf(i)]
y_mean.append(np.mean(inverse_cdf_y))

x_pred = x_pred + [9 for j in range(len(inverse_cdf_y))]
y_pred = y_pred + inverse_cdf_y
```



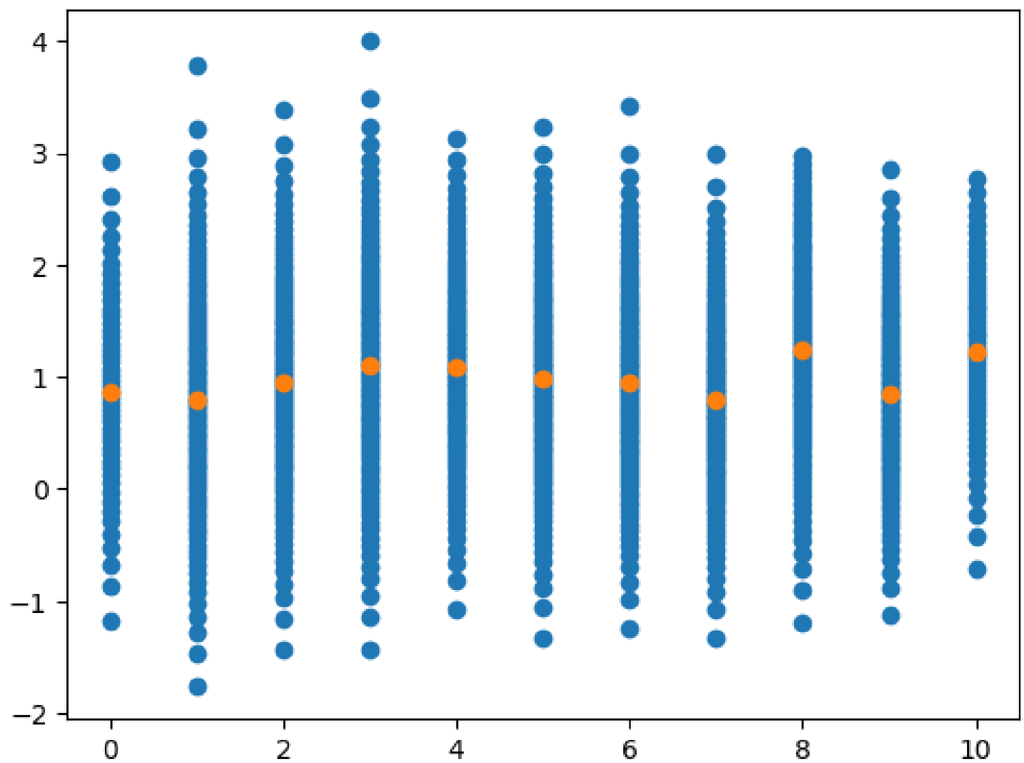
For x=10

```
[124]: y_sort = sorted(y[950:1000])

(mean, stddev, amplitude), cov = curve_fit(f_cdf, y_sort, inpo, p0 = [0.5, 1, 1])

plt.plot(y_sort, inpo)
plt.plot(y_sort, f_cdf(y_sort, mean, stddev, amplitude))


inverse_cdf_y = inverse_cdf(inpo, mean, stddev, amplitude)
inverse_cdf_y = [i for i in inverse_cdf_y if not math.isnan(i) and not math.
 ↪isinf(i)]
y_mean.append(np.mean(inverse_cdf_y))
x_pred = x_pred + [10 for j in range(len(inverse_cdf_y))]
y_pred = y_pred + inverse_cdf_y
```



[125]:
```
x_un=[i for i in range(11)]

plt.scatter(x_pred,y_pred)
plt.scatter(x_un,y_mean)
```

[125]: <matplotlib.collections.PathCollection at 0x7f16a898a640>

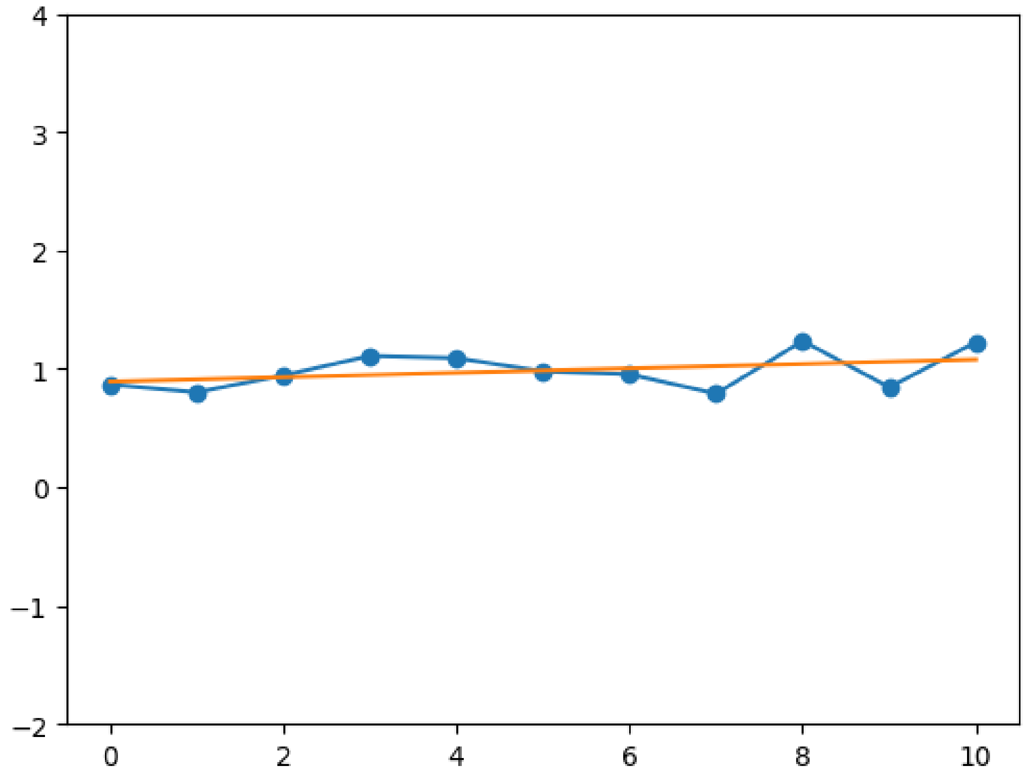


The orange highlighted datapoints are the mean points of the predicted y values for a given x from the Gaussian curve predicted from through `curve_fit`. Now I tried to fit them using a straight line and this is the acquired curve.

[126]:
```
from scipy.optimize import curve_fit
def stline(x, m, c): #defining a straight line function
    return m*x + c
(zp1, zp2), pcov = curve_fit( stline,x_un, y_mean)
print(f"Estimated function: exp(-{zp1}t) + {zp2}")
z_est = stline(np.array(x_un), zp1, zp2)
plt.plot(x_un, y_mean,x_un, z_est)
plt.scatter(x_un,y_mean)
plt.ylim(-2,4)
```

```
Estimated function: exp(-0.018704187577556874t) + 0.8929939862598517
```

[126]:
```
(-2.0, 4.0)
```


4
3
2
1
0
−1
−2
0
2
4
6
8
10