`Linear, Matrix Alzebra & Probability` using Python

This notebook is created with an objective to uderstand some daily usage DS/ML functions by `implementing them from scratch.`

Notebook Contents

In [1]:

```
import os
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import scipy

from functools import reduce
from random import uniform
from sklearn.metrics.pairwise import cosine_similarity

%matplotlib inline
```

Matrices-DOT_product

CASE-1.1

1-d arrays

In [2]:

```
A = np.array([1,4,7])
B = np.array([3,5,2])
```

In [3]:
```
A.shape, B.shape
```
Out[3]:
```
((3,), (3,))
```

In [4]:
```
A.ndim, B.ndim
```
Out[4]:
```
(1, 1)
```

In [5]:
```
def dot_product_1d(vec1, vec2):
    """
    Description: This function is created for generating dot product result of 1-d a
    Inputs: It accepts two inputs:
        1. vec1 : 1-d numpy array
        2. vec2 : 1-d numpy array

    Return:
        dot_result : DOT product of two 1-d arrays.
    """
    prd_vals = []
    for idx in enumerate(vec1):
        elements_prd = vec1[idx[0]] * vec2[idx[0]]
        prd_vals.append(elements_prd)

    ## Way -1 : Using global sum method
    dot_result = sum(prd_vals)

    ## Way -2 : Using numpy sum
    # dot_result = np.sum(prd_vals)

    ## Way -3 : Using reduce function with lambda
    # dot_result = reduce(lambda x,y:x+y,prd_vals)

    return np.sum(prd_vals)
```

In [6]:
```
dot_product_1d(A,B)
```
Out[6]:
```
37
```

Match the results

In [7]:
```
np.dot(A,B)
```
Out[7]:
```
37
```

In [8]:
```
A @ B
```
Out[8]:
```
37
```

Bingo!! All matched above

CASE-1.2

2x2 matrices

In [9]:
```
A2 = np.array([[1,4],[3,6]])
B2 = np.array([[2,1],[4,5]])
```

In [10]:
```
A2.shape, B2.shape
```
Out[10]:
```
((2, 2), (2, 2))
```

In [11]:
```
A2.ndim, B2.ndim
```

Out[11]:
```
(2, 2)
```

In [12]:
```
def matrix_transpose(inp_matrix):
    """
    Description: This function is created for performing the transpose of a matrix.
    Input: It accepts any dimension matrix or array.
        1. inp_matrix : list
    Return: Transposed matrix
        2. transposed_matrix: numpy array
    """
    ## Generating the transposed matrix rows and cols
    trans_cols_num = len(inp_matrix)
    trans_rows_num = len(inp_matrix[0])

    ## Flattening the data row-wise
    flatten_matrix = [element for row in inp_matrix for element in row]

    ## Generating the transposed indices then finding the values from flatten matrix
    matrix = []
    for i in range(trans_rows_num):
        vals = []
        for j in range(i,trans_rows_num * trans_cols_num,trans_rows_num):
            vals.append(flatten_matrix[j])
        matrix.append(vals)

    ## Converting from list to numpy array
    transposed_matrix = np.array(matrix)
    return transposed_matrix

def dot_product_2d(m1,m2):
    """
    Description: This function is created for generating dot product result of 2-d m
    Inputs: It accepts two inputs:
        1. m1 : 2-d numpy array
        2. m2 : 2-d numpy array

    Return:
        dot_prds : DOT product of two 2-d matrices.
    """
    ## Generating transpose of 2nd matrix
    m2_transpose = matrix_transpose(m2)

    ## Generating dot product
    dot_prds = []
    for ix_a in enumerate(m1):
        vals = []
        for ix_b in enumerate(m2_transpose):
            vals.append(dot_product_1d(m1[ix_a[0]], m2_transpose[ix_b[0]]))
        dot_prds.append(vals)
    return np.array(dot_prds)
```

In [13]:
```
dot_product_2d(A2,B2)
```

Out[13]:
```
array([[18, 21],
       [30, 33]])
```

Match the results

In [14]:
```
np.dot(A2,B2)
```

Out[14]:
```
array([[18, 21],
       [30, 33]])
```

In [15]:
```
A2 @ B2
```

Out[15]:
```
array([[18, 21],
       [30, 33]])
```

Bingo!! All matched above

CASE-1.3

Non-square shape matrices

In [16]:
```
A3 = np.array([[1,4,5],[3,6,7]])
B3 = np.array([[2,1,3],[4,5,1]])
```

In [17]:
```
A3.shape, B3.shape
```

Out[17]:
```
((2, 3), (2, 3))
```

In [18]:
```
A3.ndim, B3.ndim
```

Out[18]:
```
(2, 2)
```

In [19]:
```
dot_product_2d(A3,B3.T)
```

Out[19]:
```
array([[21, 29],
       [33, 49]])
```

Match the results

In [20]:
```
np.dot(A3,B3.T)
```

Out[20]:
```
array([[21, 29],
       [33, 49]])
```

In [21]:
```
A3 @ B3.T
```

Out[21]:
```
array([[21, 29],
       [33, 49]])
```

Bingo!! All matched above

CASE-1.4

3x3 matrices

In [22]:
```
A4 = np.array([[1,4,5],[3,6,7],[5,6,7]])
B4 = np.array([[2,1,3],[4,5,1],[7,3,2]])
```

In [23]:
```
A4.shape, B4.shape
```

Out[23]:
```
((3, 3), (3, 3))
```

In [24]:
```
A4.ndim, B4.ndim
```

Out[24]:
```
(2, 2)
```

In [25]:
```
dot_product_2d(A4,B4)
```

Out[25]:
```
array([[53, 36, 17],
       [79, 54, 29],
       [83, 56, 35]])
```

Match the results

In [26]:
```
np.dot(A4,B4)
```

Out[26]:
```
array([[53, 36, 17],
       [79, 54, 29],
       [83, 56, 35]])
```

In [27]:
```
A4 @ B4
```

Out[27]:
```
array([[53, 36, 17],
       [79, 54, 29],
       [83, 56, 35]])
```

Bingo!! All matched above

Cosine_similarity

Two 1-d vectors

In [28]:
```
A, B
```

Out[28]:
```
(array([1, 4, 7]), array([3, 5, 2]))
```

In [29]:
```
A2, B2
```

Out[29]:
```
(array([[1, 4],
        [3, 6]]),
 array([[2, 1],
        [4, 5]]))
```

In [30]:
```
A3, B3
```

Out[30]:
```
(array([[1, 4, 5],
        [3, 6, 7]]),
 array([[2, 1, 3],
        [4, 5, 1]]))
```

Vectors_Magnitude

In [31]:
```
def vector_magnitude(vect):
    """
    Description: This function is created for calculating the magnitude of the vecto
    Input: It accepts only one parameter:
        1. vect: np.array
            Vector whose magnitude to be calculated
    Return: Calculated length/magnitude of the vector
        vect_mag
    """
    if len(vect.shape) == 1:
        vect = [vect]

    ## Flattening the data row-wise
    flatten = lambda x : [element for row in x for element in row]
    flat_vect = flatten(vect)

    ## Squared sum of elements
    sqrd_elements_sum = reduce(lambda x,y:x+y,[element**2 for element in flat_vect])

    ## Square-root of squared elements sum
    vect_mag = np.sqrt(sqrd_elements_sum)
    return vect_mag
```

In [32]:
```
## Vectors magnitude
vector_magnitude(A),vector_magnitude(B), vector_magnitude(A2),vector_magnitude(B2),
```

Out[32]:
```
(8.12403840463596,
 6.164414002968976,
 7.874007874011811,
 6.782329983125268,
 11.661903789690601,
 7.483314773547883)
```

In [33]:
```
## Numpy generated vector's magnitude
np.linalg.norm(A), np.linalg.norm(B), np.linalg.norm(A2), np.linalg.norm(B2), np.lin
```

Out[33]:
```
(8.12403840463596,
 6.164414002968976,
 7.874007874011811,
 6.782329983125268,
 11.661903789690601,
 7.483314773547883)
```

Bingo!! All matched above

In [34]:
```
def cosine_sim(vect1,vect2):
    """
    Description: This function is created for finding the cosine similarity b/w 2 ve
    Input: It accepts 2 parameters:
        1. vect1: np.array
        2. vect2: np.array
    Return: Calculate the cosine similarity
        vects_cosine_similarity
    """
    ## Generating dot product
    dot_prd_vects = dot_product_1d(vect1,vect2)

    ## Calculating vectors magnitudes
    vect1_mag = vector_magnitude(vect1)
    vect2_mag = vector_magnitude(vect2)

    ## Finding the cosine similarity
    vects_cosine_similarity = dot_prd_vects / (vect1_mag * vect2_mag)

    return vects_cosine_similarity
```

In [35]:
```
## Self-implementation
cosine_sim(A,B)
```

Out[35]:
```
0.7388188340435563
```

In [36]:
```
## Sklearn result
cosine_similarity([A],[B])
```

Out[36]:
```
array([[0.73881883]])
```

Bingo!! results matched

Matrix_Transpose

1-d array

In [37]:
```
A
```

Out[37]:
```
array([1, 4, 7])
```

In [38]:
```
matrix_transpose([A])
```

Out[38]:
```
array([[1],
       [4],
```

```
        [7]])
```

2x2 matrix

In [39]:
```
A2
```

Out[39]:
```
array([[1, 4],
       [3, 6]])
```

In [40]:
```
matrix_transpose(A2)
```

Out[40]:
```
array([[1, 3],
       [4, 6]])
```

Non-square matrix

In [41]:
```
A3
```

Out[41]:
```
array([[1, 4, 5],
       [3, 6, 7]])
```

In [42]:
```
matrix_transpose(A3)
```

Out[42]:
```
array([[1, 3],
       [4, 6],
       [5, 7]])
```

3x3 matrix

In [43]:
```
A4
```

Out[43]:
```
array([[1, 4, 5],
       [3, 6, 7],
       [5, 6, 7]])
```

In [44]:
```
matrix_transpose(A4)
```

Out[44]:
```
array([[1, 3, 5],
       [4, 6, 6],
       [5, 7, 7]])
```

Generate_QQ_plot

In [45]:
```
from statsmodels.graphics.gofplots import qqplot
```

Normal_Dist

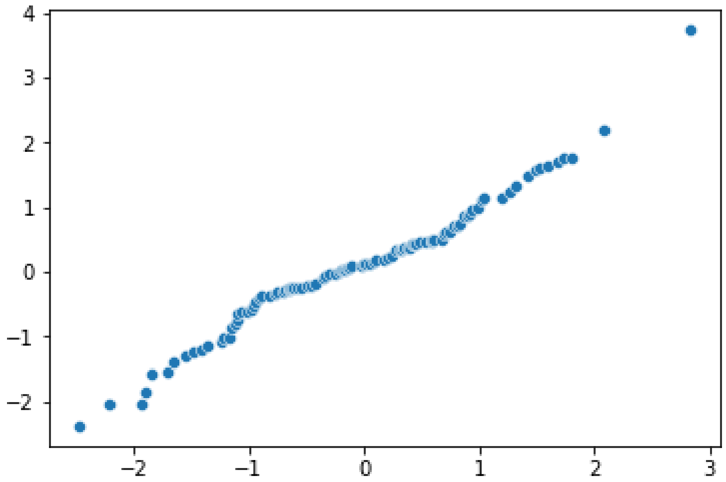
In [46]:
```
np.random.seed(33)
Aq = np.random.normal(size=200)
```

In [47]:
```
Aq_percentiles = []
percentiles_100 = np.arange(start=1,stop=101,step=1)
Aq_100_percentiles = np.percentile(Aq,percentiles_100)
sorted_Aq_100_percentiles = np.sort(Aq_100_percentiles)
sorted_Aq_100_percentiles
```

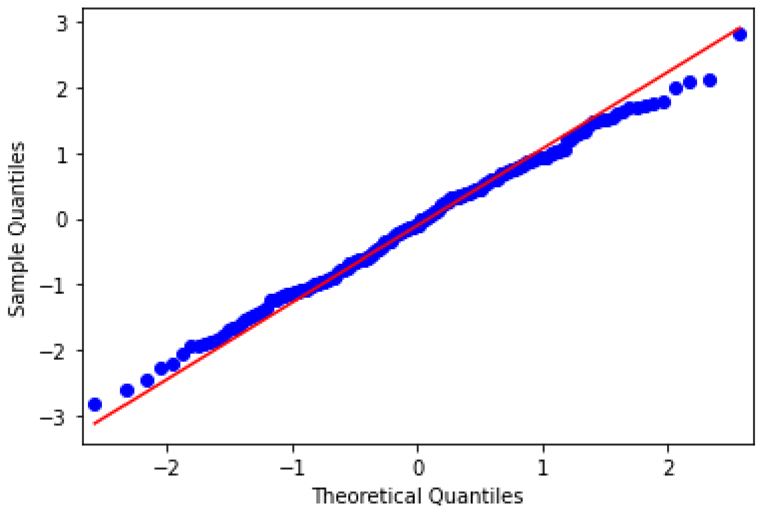
Out[47]:
```
array([-2.46599034e+00, -2.20718808e+00, -1.93756264e+00, -1.90145355e+00,
       -1.85187748e+00, -1.70389613e+00, -1.65477286e+00, -1.54063889e+00,
       -1.47095411e+00, -1.41252529e+00, -1.35914329e+00, -1.24005629e+00,
       -1.21815039e+00, -1.17260255e+00, -1.14471007e+00, -1.12181570e+00,
       -1.09882312e+00, -1.08945676e+00, -1.06780572e+00, -1.00372920e+00,
       -9.82088295e-01, -9.52161294e-01, -9.42296878e-01, -9.07972048e-01,
       -8.95722484e-01, -8.23487807e-01, -7.71375816e-01, -7.48173002e-01,
       -6.98078059e-01, -6.74285594e-01, -6.45210123e-01, -6.37765139e-01,
       -6.22855228e-01, -6.09429380e-01, -5.76342200e-01, -5.43078094e-01,
       -4.86952908e-01, -4.61321046e-01, -4.17134964e-01, -3.56951384e-01,
```

```
       -3.39976411e-01, -3.03754735e-01, -2.41162835e-01, -2.19266416e-01,
       -1.99036183e-01, -1.69834334e-01, -1.54899113e-01, -1.47856249e-01,
       -1.28421081e-01, -1.00257198e-01, -2.01397309e-02,  7.19443110e-04,
        1.19903444e-02,  5.33415291e-02,  7.72597688e-02,  1.08443618e-01,
        1.69862822e-01,  2.10027101e-01,  2.40350518e-01,  2.80324201e-01,
        3.10958282e-01,  3.26558265e-01,  3.37795318e-01,  3.45090672e-01,
        3.74260739e-01,  3.90046267e-01,  4.11446152e-01,  4.37987660e-01,
        4.51541635e-01,  4.83538964e-01,  5.30121724e-01,  5.82806466e-01,
        5.97042279e-01,  6.07814399e-01,  6.80318861e-01,  6.94200508e-01,
        7.17511460e-01,  7.42719732e-01,  7.75784906e-01,  8.16166492e-01,
        8.37458937e-01,  8.73758834e-01,  9.07505436e-01,  9.25354480e-01,
        9.34724870e-01,  9.92162479e-01,  1.01839827e+00,  1.04540433e+00,
        1.19501581e+00,  1.27442544e+00,  1.32538626e+00,  1.42386270e+00,
        1.48902368e+00,  1.52065137e+00,  1.59244787e+00,  1.68425669e+00,
        1.73481724e+00,  1.79728104e+00,  2.07630429e+00,  2.82127933e+00])
```

In [48]:
```
sns.scatterplot(x=sorted_Aq_100_percentiles,y=np.sort(np.random.normal(size=100)));
```



In [49]:
```
qqplot(Aq,line='q');
```



Bingo!! Looks quite similar

Log-Normal_Dist

In [50]:
```
Aq_log = np.random.lognormal(size=200)
```

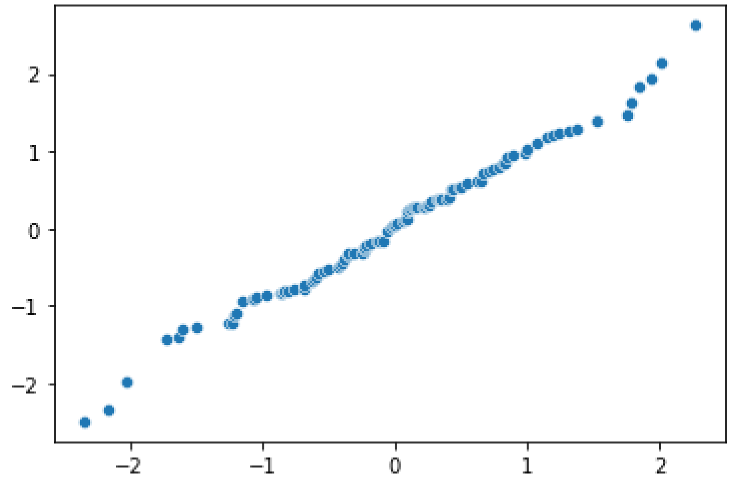
In [51]:
```
Aq_log_percentiles = []
Aq_log_100_percentiles = np.percentile(np.log(Aq_log),percentiles_100)
sorted_Aq_log_100_percentiles = np.sort(Aq_log_100_percentiles)
sorted_Aq_log_100_percentiles
```

Out[51]:
```
array([-2.33820973, -2.15270392, -2.02422615, -1.72585197, -1.62549138,
```

```
       -1.59856845, -1.49991339, -1.25393201, -1.22995031, -1.20401731,
       -1.19024699, -1.15117659, -1.06548097, -1.05532187, -1.04395245,
       -0.97042171, -0.86805412, -0.84764827, -0.82974912, -0.82575889,
       -0.79834578, -0.75447177, -0.67881047, -0.67475326, -0.62189926,
       -0.60935312, -0.58807236, -0.56751884, -0.53181589, -0.50453319,
       -0.42546144, -0.40505289, -0.39422791, -0.3769809 , -0.35080662,
       -0.3425501 , -0.30923394, -0.24694726, -0.23044649, -0.22405655,
       -0.21567083, -0.18650033, -0.13029206, -0.12363573, -0.0942395 ,
       -0.0662814 , -0.03149388, -0.02263501, -0.00454816,  0.01169328,
        0.05943071,  0.07636514,  0.08926398,  0.09381723,  0.09865338,
        0.11533098,  0.1361097 ,  0.15232059,  0.16282453,  0.2158422 ,
        0.21984901,  0.24099094,  0.25983132,  0.27837256,  0.31617368,
        0.32973924,  0.34098613,  0.34951088,  0.38557789,  0.40384873,
        0.42730743,  0.43572334,  0.4849015 ,  0.50207365,  0.53655899,
        0.61355499,  0.64902672,  0.66780204,  0.71349242,  0.74572818,
        0.77962727,  0.81625619,  0.83591029,  0.85069217,  0.8917861 ,
        0.98551805,  0.99378238,  1.0763369 ,  1.15371561,  1.19941537,
        1.24246709,  1.31750439,  1.36708015,  1.52771237,  1.74468211,
        1.78443762,  1.83952594,  1.93578151,  2.00866447,  2.25733657])
```

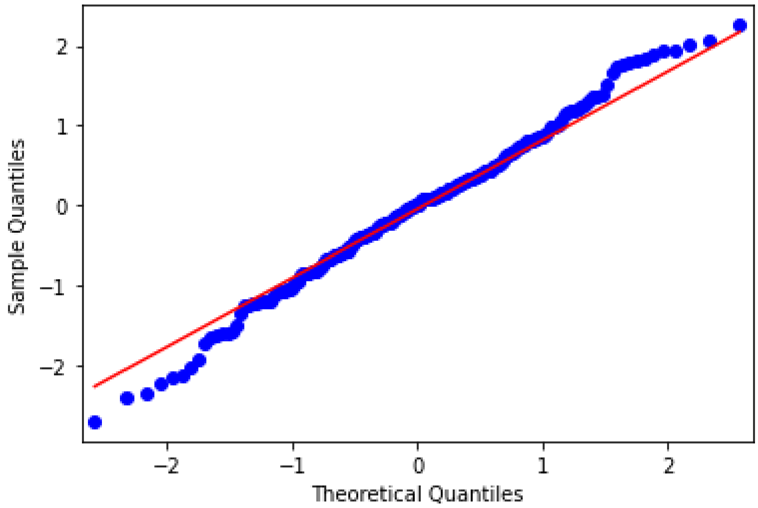
In [52]:

```
sns.scatterplot(x=sorted_Aq_log_100_percentiles,y=np.sort(np.random.normal(size=100)
```



In [53]:

```
qqplot(np.log(Aq_log),line='q');
```



Bingo!! Looks quite similar

Some_Matrix_operations

- Cross-product
- Minors of matrix
    - Minors of Diagonals

- Co-factors
- Adjugate
    - Given a square matrix A, the transpose of the matrix of the cofactor of A is called adjoint of A and is denoted by adj A. An adjoint matrix is also called an adjugate matrix. In other words, we can say that matrix A is another matrix formed by replacing each element of the current matrix by its corresponding cofactor and then taking the transpose of the new matrix formed.
- Determinant
- Inverse
- Trace --> (Sum of Diagonal elements)

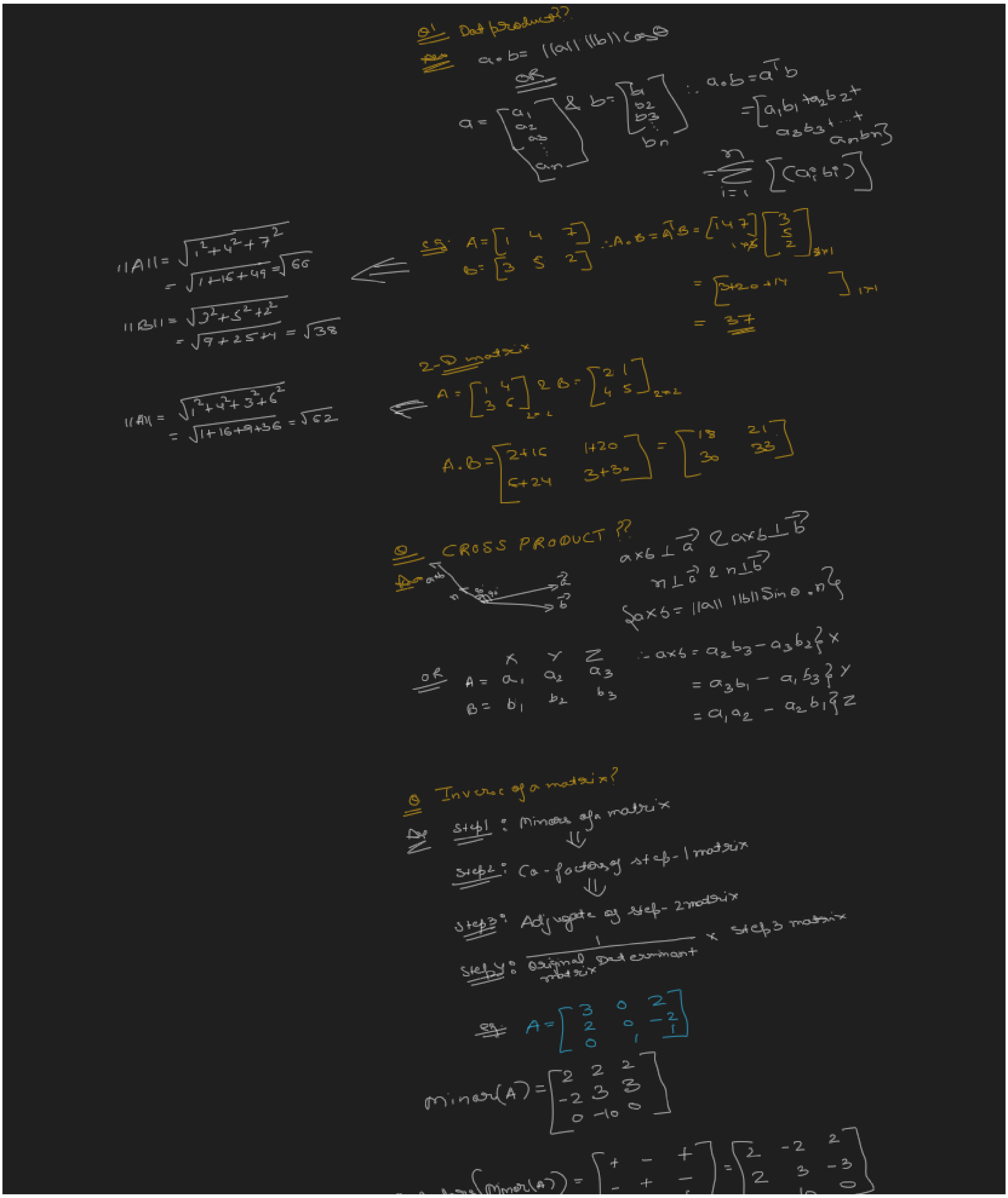
In [54]:

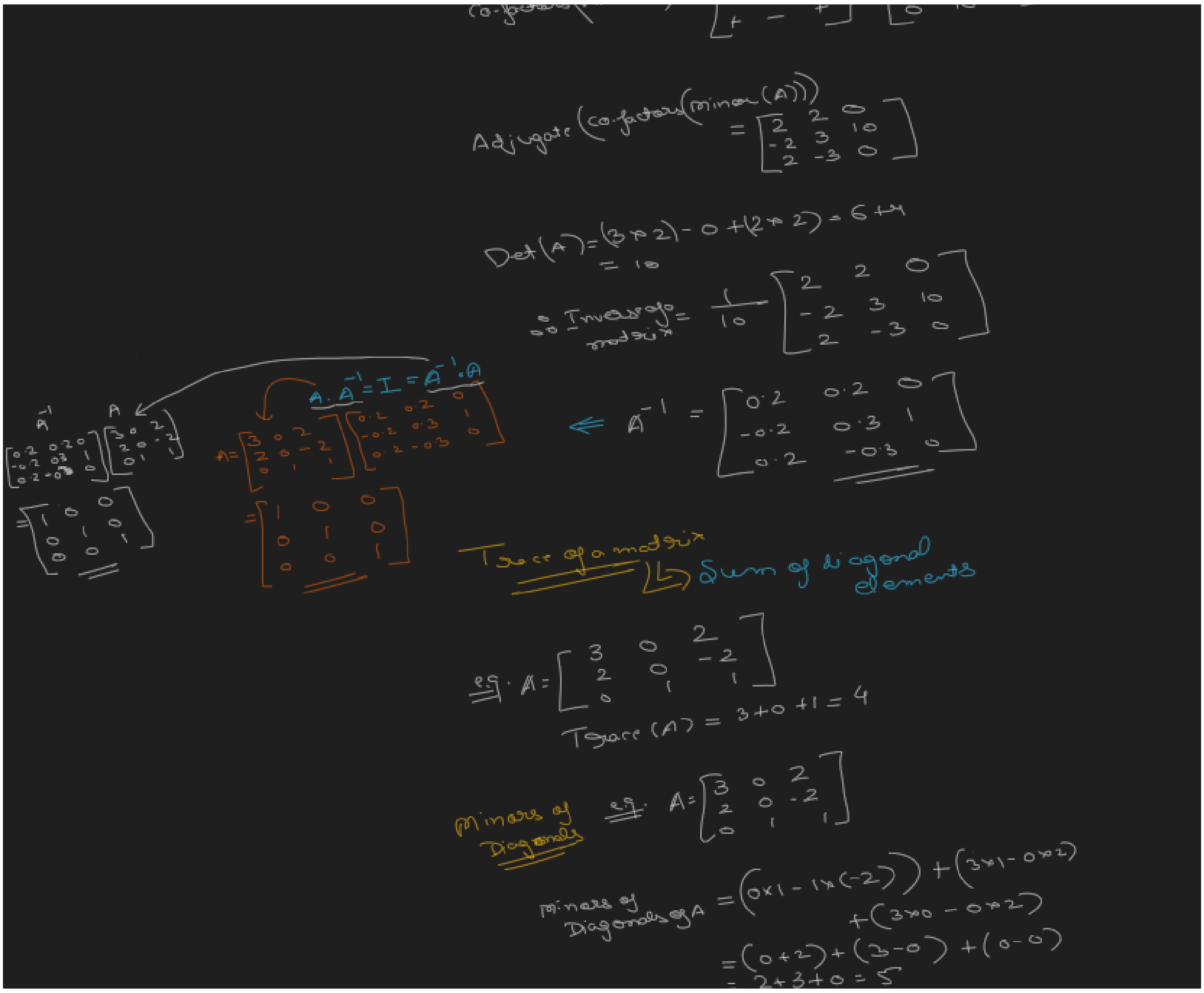
```
from IPython.display import Image
```

In [55]:

```
## Some Important Matrix operations
Image("Some_Matrix_Operations.png",width=1000,height=1000)
```

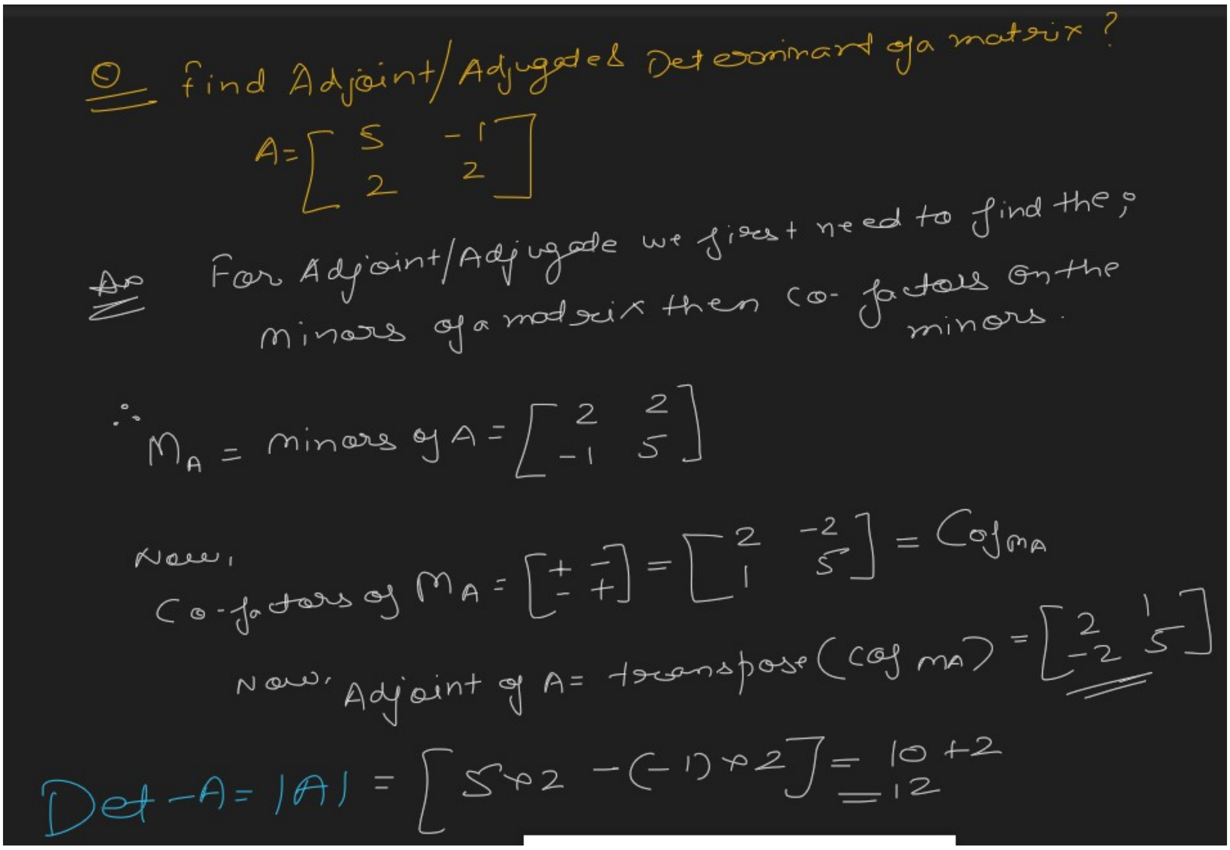
Out[55]:





In [56]:
```
## Solve MAtrix Adjugate/Adjoint question
Image("Matrix_Adjugate_Det.jpg",width=1000,height=1000)
```

Out[56]:



In [57]:
```
## Self revision notes
Image("Linear_Matrix_Alzebra.png",width=2000,height=2000)
```

Out[57]:

In [58]:

```python
## Solve this question on Cross product
Image("Matrix_Alzebra_Q1.jpg")
```

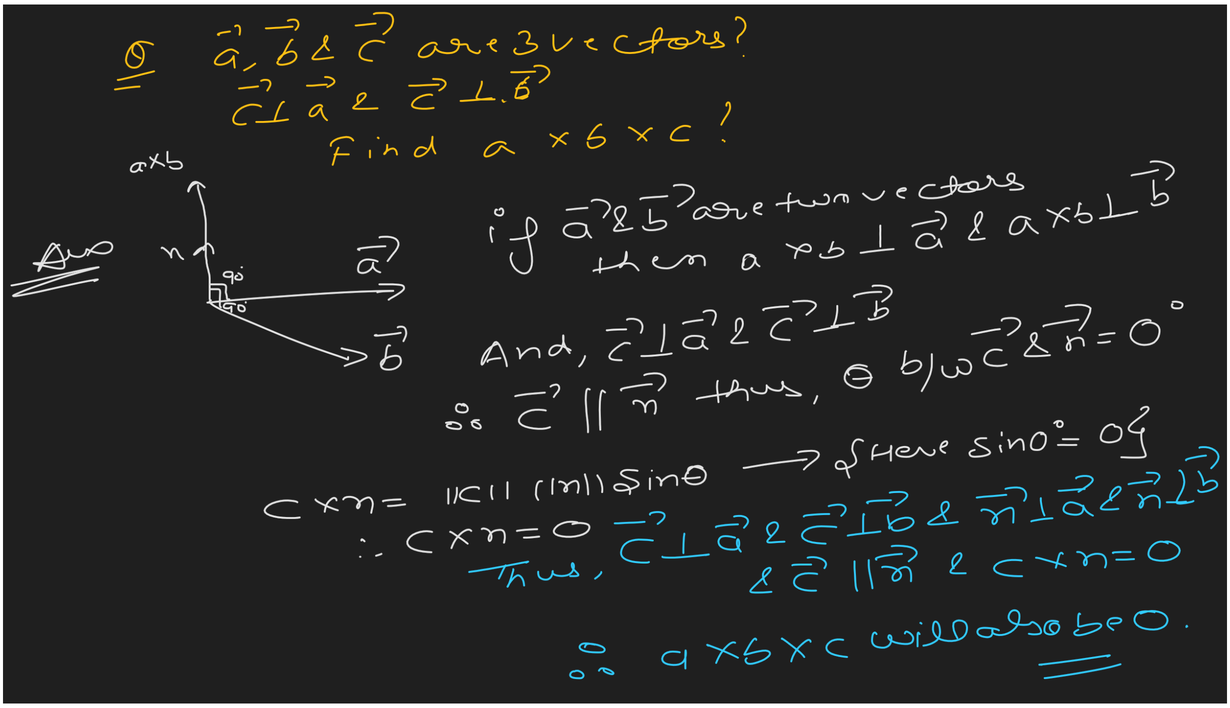
Out[58]:

a, **b** and **c** are three vectors such that **c** is perpendicular to both **a** and **b**
What is the value of **a** × **b** × **c**?

A (1, 1, 1)	B (0, 0, 0)
C (1, 1, 0)	D (0, 0, 1)

In [59]:
```
## Solution
Image("Cross_prd_qs.png")
```

Out[59]:

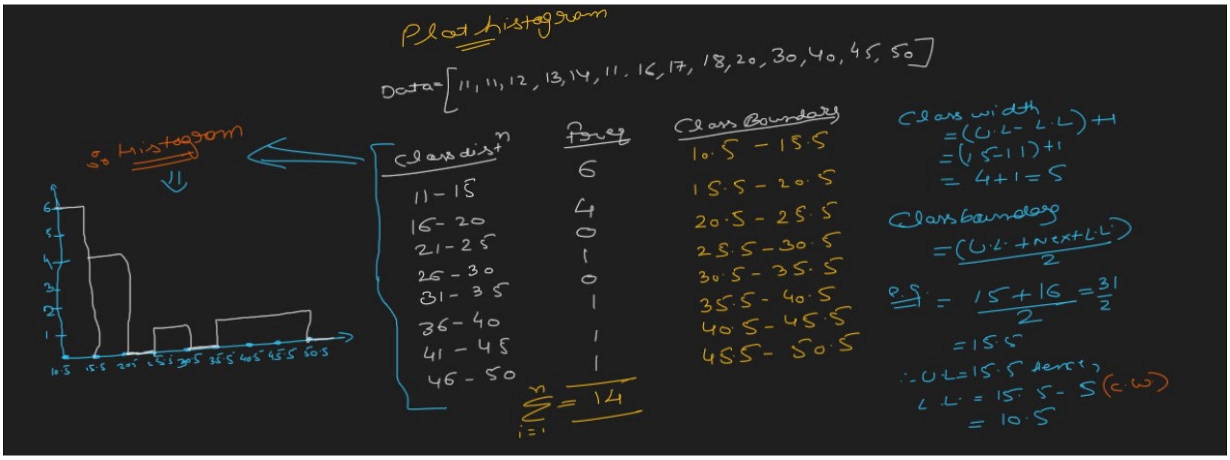


Generate_histogram

How to plot the histogram of a data?

In [60]:
```
Image("Plot_Histogram.jpg",width=1200,height=1200)
```

Out[60]:



In [61]:
```
Image("Histogram_RelFreq_ProbDen_Cum_Freq.png",width=1000,height=1000)
```

Out[61]:

PDF (Prob. density fun) ⇓ Continuous variables

(Probability mass fun) PMF ⇓ Discrete variables

Discrete variables → Uniformly distributed; Non-uniformly distributed

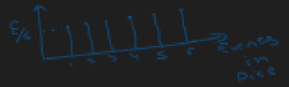
Continuous variables → normally distributed (means the data is normal); non-normal distributed (means data is non-normal)

Uniformly distributed: [1,2,3,4,5,6] or [0.2, 0.4, 0.6, 0.8]

Non-uniformly distributed: [1, 3, 2, 7, 8, 17]; [0.2, 0.9, 0.7, 0.1]

eg. [0.13, 0.14, 0.15, 0.20, 0.25, 0.372 ....]

In PMF, we have equi-probable probs.

eg. 

eg. $E_1 = 1/6$; $E_3 = 1/6$; $E_5 = 1/6$
$E_2 = 1/6$; $E_4 = 1/6$; $E_6 = 1/6$

So, Plot histogram; Rel. freq; Cum freq; Prob. density; KDE??

App Histogram

eg. [11, 11, 12, 13, 14, 11, 16, 17, 18, 20, 30, 40, 45, 50]

Step-1 Bins or Intervals = 10 → We can use different way of finding suitable bins or # of bins / intervals. Also known as n_h.

(i) sqrt(n) → size of data

(ii) $\log_2(n+1)$

(iii) $2n^{(1/3)}$

eg. if n = 10,000 ∴ # of bins = $\sqrt{10000}$ = 100

or $\log_2(10001) \approx 9.2$

or $2(10000)^{1/3} = 43$

So, lets say n_h = 10

now, bins_width = ??

⇒ Again, there are various ways for calculating the bins_width.

(i) bins_width = (max value − min values)/(n_h)

now, we got both n_h & h. Let's calculate UL & LL:

L.L₁ = min_value

U.L₁ = (min_value + h) → bin_width

L.L₂ = U.L₁

U.L₂ = LL₂ + bin_width

L.L₃ = U.L₂

U.L₃ = L.L₃ + bin_width

⋮

Similarly, we can calculate U.L & L.L for every bin.

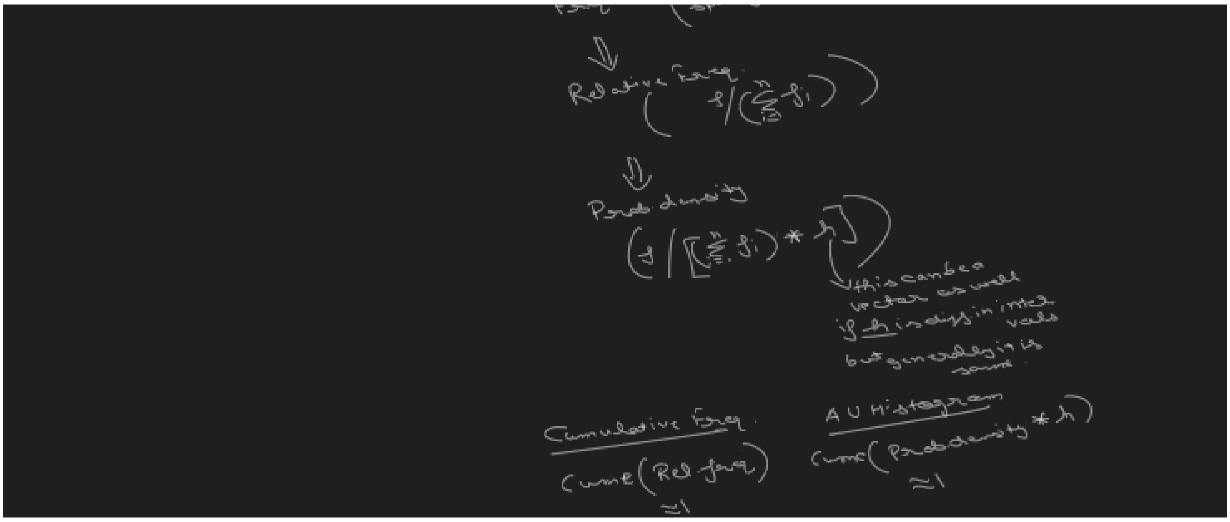
eg. bins = 10 = n_h

bin_width = h = 3.9

∴ class interval ⇒ L.L₁ = 11, U.L₁ = 11 + 3.9 = 14.9

	Freq	R.F.	Prob. Density
∴ C.I.₁ = [11 − 14.9]	6	6/14 = 0.428	6/(14 × 3.9) = 0.1098
C.I.₂ = [14.9 − 18.8]	3	3/14 = 0.214	3/(14×3.9) = 0.0549
C.I.₃ = [18.8 − 22.7]	1	1/14 = 0.071	1/(14×3.9) = 0.0183
C.I.₄ = [22.7 − 26.6]	0	0/14 = 0	0
C.I.₅ = [26.6 − 30.5]	1	1/14 = 0.071	1/(14×3.9) = 0.0183
C.I.₆ = [30.5 − 34.4]	0	0/14 = 0	0
C.I.₇ = [34.4 − 38.3]	0	0/14 = 0	0
C.I.₈ = [38.3 − 42.21]	1	1/14 = 0.071	1/(14×3.9) = 0.0183
C.I.₉ = [42.21 − 46.11]	1	1/14 = 0.071	1/(14×3.9) = 0.0183
C.I.₁₀ = [46.11 − 50.01]	1	1/14 = 0.071	1/(14×3.9) = 0.0183
	$\sum_{i=1}^{n} = 14$		

So, Bins and Bin width
⇓
class intervals
⇓
Frequency (# of data values in a specific interval)



In [62]:
```python
def plot_hist(data,number_of_bins=10):
    """
    Description : This function is created for plotting the histogram of data.

    Input: It accepts below parameters:
        1. data : For which histogram to be plotted
        2. number_of_bins : number of intervals or bins to be formed. By default = 1

    Return: Plot the graph and returns the dataframe with values
    """
    bins = number_of_bins

    # Sort the data in ascending order
    hist_data_srt = np.sort(data)

    # Calculating bin width
    bin_width = (max(hist_data_srt)+0.01 - min(hist_data_srt))/bins
    bin_width = np.round(bin_width,3)

    # Generate the class_limits list
    class_limits = []
    class_limits.append(hist_data_srt.min())
    for i in range(1,(bins*2),1):
        if i%2 != 0:
            class_limits.append(class_limits[-1]+bin_width)
        else:
            class_limits.append(class_limits[-1])

    # Calculate the upper_limit of class_boundary
    ul = (class_limits[1] + class_limits[2])/2

    # Calulate the difference between class_boundary and class_distn raw data value
    diff = ul - class_limits[1]

    # Substracting the diff from raw class values
    cb = [raw_val[1]-diff if raw_val[0]%2 == 0 else raw_val[1]+diff for raw_val in e

    # Bucketing the class_distribution and class_boundaries rows
    intrvals = [i for i in range(len(cb)) if i%2 == 0]
    class_distn_cb = []
    for idx in intrvals:
        class_distn_cb.append([cb[idx],cb[idx+1]])

    class_distn = []
    for idx in intrvals:
        class_distn.append([class_limits[idx],class_limits[idx+1]])

    # Calculating the Frequency of data in the range
```

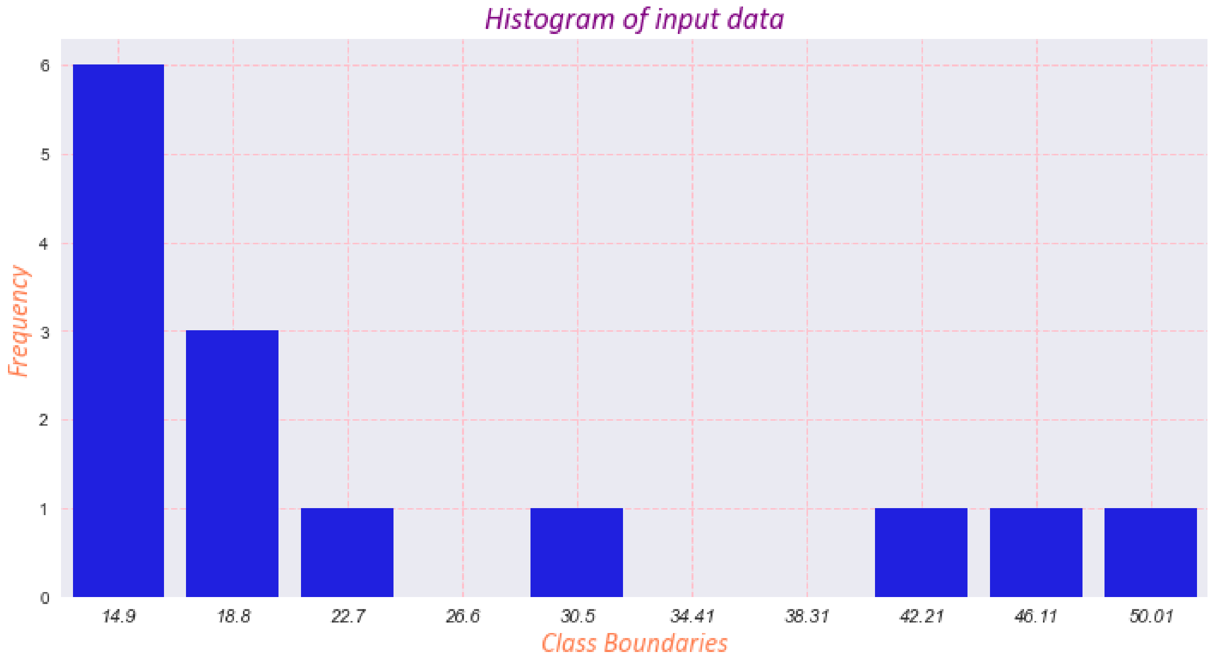
```
    freq = []
    for i in range(len(class_distn)):
        freq.append(len([val for val in hist_data_srt if (val >= class_distn[i][0])

    # Preparaing the data in the dataframe
    class_distn_df = pd.DataFrame([class_distn]).T
    class_distn_df.columns = ['Class_Distribution']

    freq_df = pd.DataFrame(freq)
    freq_df.columns = ['Frequency']

    class_distn_cb_df = pd.DataFrame([class_distn_cb]).T
    class_distn_cb_df.columns = ['Class_Boundaries']

    class_data_df = pd.concat([class_distn_df,freq_df,class_distn_cb_df],axis=1)
    class_data_df['Class_Boundaries'] = class_data_df['Class_Boundaries'].astype(str
    class_data_df['Class_Boundaries'] = class_data_df['Class_Boundaries'].apply(lamb

    ### Calculating relative frequencies for every interval
    class_data_df['Relative_Freq'] = class_data_df['Frequency'].apply(lambda val: va

    ### Calculating probability density for every interval
    class_data_df['Prob_Density'] = class_data_df['Frequency'].apply(lambda val: val

    # Bins_intervals for export
    bins_intervals = []
    for val in class_data_df['Class_Boundaries'].values:
        bins_intervals.append(np.round(np.float(val.replace("[",'').replace("]",'').
        bins_intervals.append(np.round(np.float(val.replace("[",'').replace("]",'').

    bins_intervals = np.unique(bins_intervals)

    ## Prob Density for export
    prob_density_out = class_data_df['Prob_Density'].values

    # Plotting the graph
    if number_of_bins>20:
        rot=90
        tick_size=8
        f_size=(15,7)
    else:
        rot=0
        tick_size=11
        f_size=(12,6)

    with plt.style.context('seaborn'):
        plt.figure(figsize=f_size)
        sns.barplot(x=bins_intervals[1:],y=class_data_df['Frequency'].values,color='
        plt.title("Histogram of input data",fontdict={'family':'calibri','size':18,'
        plt.xlabel("Class Boundaries",fontdict={'family':'calibri','size':16,'style'
        plt.ylabel("Frequency",fontdict={'family':'calibri','size':16,'style':'obliq
        plt.grid(which='major',color='pink',linestyle='--')
        plt.xticks(size=tick_size,style='oblique',rotation=rot)
        plt.show()

    return class_data_df, bins_intervals, prob_density_out
```

In [63]:

```
hist_data = [11,11,12,13,14,11,16,17,18,20,30,40,45,50]
```

In [64]:

```
hist_data_results, bins_intervals, bins_prob_density = plot_hist(hist_data,number_of
hist_data_results
```



Out[64]:

	Class_Distribution	Frequency	Class_Boundaries	Relative_Freq	Prob_Density
0	[11, 14.901]	6	[11.0 - 14.901]	0.428571	0.109862
1	[14.901, 18.802]	3	[14.901 - 18.802]	0.214286	0.054931
2	[18.802, 22.703]	1	[18.802 - 22.703]	0.071429	0.018310
3	[22.703, 26.604]	0	[22.703 - 26.604]	0.000000	0.000000
4	[26.604, 30.505]	1	[26.604 - 30.505]	0.071429	0.018310
5	[30.505, 34.406]	0	[30.505 - 34.406]	0.000000	0.000000
6	[34.406, 38.307]	0	[34.406 - 38.307]	0.000000	0.000000
7	[38.307, 42.208]	1	[38.307 - 42.208]	0.071429	0.018310
8	[42.208, 46.108999999999995]	1	[42.208 - 46.108999999999995]	0.071429	0.018310
9	[46.108999999999995, 50.00999999999999]	1	[46.108999999999995 - 50.00999999999999]	0.071429	0.018310

In [65]:

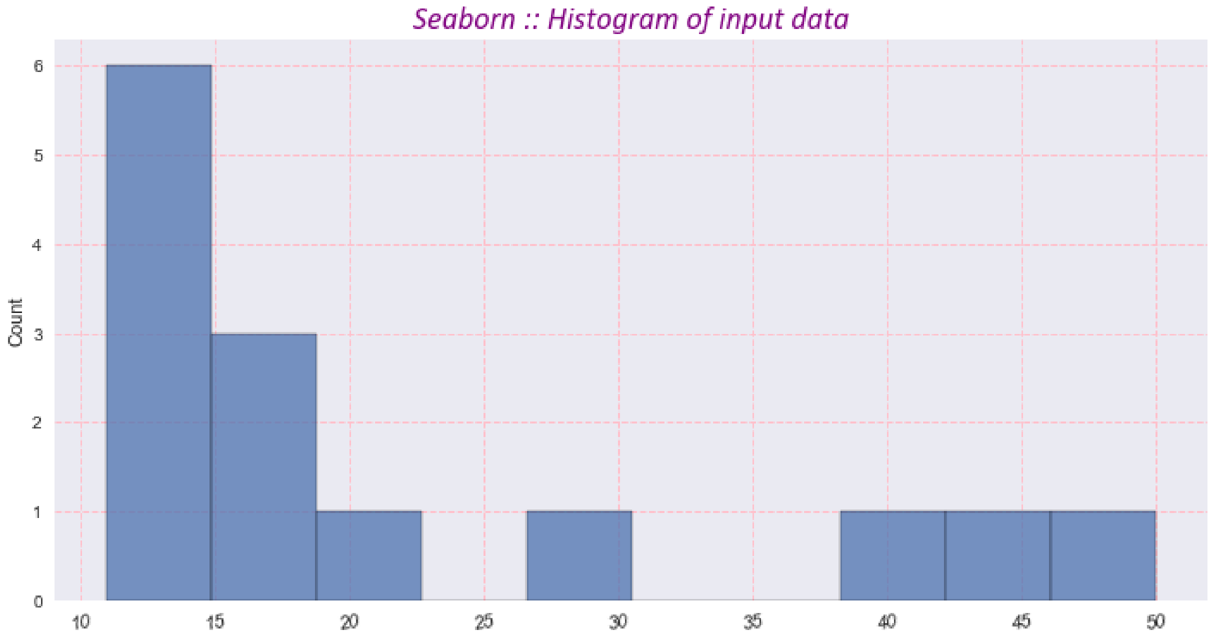
```
bins_intervals, bins_prob_density
```

Out[65]:

```
(array([11.  , 14.9 , 18.8 , 22.7 , 26.6 , 30.5 , 34.41, 38.31, 42.21,
        46.11, 50.01]),
 array([0.10986194, 0.05493097, 0.01831032, 0.        , 0.01831032,
        0.        , 0.        , 0.01831032, 0.01831032, 0.01831032]))
```

In [66]:

```
# Seaborn histogram
with plt.style.context('seaborn'):
    plt.figure(figsize=(12,6))
    sns.histplot(hist_data,bins=10)
    plt.title("Seaborn :: Histogram of input data",fontdict={'family':'calibri','siz
    plt.xticks(size=11,style='oblique',rotation=10)
    plt.grid(which='major',color='pink',linestyle='--')
    plt.show()
```



Relative_Frequency,Probability_Density_and_Cumulative_Frequen

Compare the Self implementated and Matplotlib/Seaborn Relative Frequency, Probability Density and Cumulative Frequencies

In [67]:
```
hist_data
```

Out[67]: [11, 11, 12, 13, 14, 11, 16, 17, 18, 20, 30, 40, 45, 50]

In [68]:
```
hist_data_results
```

Out[68]:

	Class_Distribution	Frequency	Class_Boundaries	Relative_Freq	Prob_Density
0	[11, 14.901]	6	[11.0 - 14.901]	0.428571	0.109862
1	[14.901, 18.802]	3	[14.901 - 18.802]	0.214286	0.054931
2	[18.802, 22.703]	1	[18.802 - 22.703]	0.071429	0.018310
3	[22.703, 26.604]	0	[22.703 - 26.604]	0.000000	0.000000
4	[26.604, 30.505]	1	[26.604 - 30.505]	0.071429	0.018310
5	[30.505, 34.406]	0	[30.505 - 34.406]	0.000000	0.000000
6	[34.406, 38.307]	0	[34.406 - 38.307]	0.000000	0.000000
7	[38.307, 42.208]	1	[38.307 - 42.208]	0.071429	0.018310
8	[42.208, 46.108999999999995]	1	[42.208 - 46.108999999999995]	0.071429	0.018310
9	[46.108999999999995, 50.00999999999999]	1	[46.108999999999995 - 50.00999999999999]	0.071429	0.018310

In [69]:
```
## Mean & Std-dev of relative frequencies
sigma_rel_freq = np.std(hist_data_results['Relative_Freq'],ddof=1)
mean_rel_freq = np.mean(hist_data_results['Relative_Freq'])

sigma_rel_freq, mean_rel_freq
```

Out[69]: (0.13127665478181164, 0.09999999999999998)

In [70]:
```
## Relative frequencies sum-up to 1
print(np.sum(hist_data_results['Relative_Freq']))

## Area under the histogram integrates to 1
print(np.sum(hist_data_results['Prob_Density'] * 5))
```

```
0.9999999999999998
1.281722635221738
```

How np.diff works?

In [71]:
```
"""
Calculate the n-th discrete difference along the given axis.

The first difference is given by ``out[i] = a[i+1] - a[i]`` along the given axis, hi
recursively.
"""
hist_diff = np.diff(hist_data,axis=-1)
hist_data, hist_diff
```

Out[71]:
```
([11, 11, 12, 13, 14, 11, 16, 17, 18, 20, 30, 40, 45, 50],
 array([ 0,  1,  1,  1, -3,  5,  1,  1,  2, 10, 10,  5,  5]))
```

How Matplotlib generates probability density?

In [72]:
```
"""
****** Matplotlib histogram and density ******
If density == ``True``, it draw and return a probability density: each bin will disp
counts * the bin width
    (``density = counts / (sum(counts) * np.diff(bins))``),

so that the area under the histogram integrates to 1
    (``np.sum(density * np.diff(bins)) == 1``).
"""
mat_plt_lib_prob_den=plt.hist(hist_data,density=True)
plt.close()
```

In [73]:
```
## Matplotlib generated probability density and class limits
print("Number of objects returned by Matplotlib:", len(mat_plt_lib_prob_den),'\n')

print("Probability Density: {}{}".format(np.round(mat_plt_lib_prob_den[0],4),'\n'))
print("Class Limits from data: {}{}".format(mat_plt_lib_prob_den[1],'\n'))
```

```
Number of objects returned by Matplotlib: 3

Probability Density: [0.1099 0.0549 0.0183 0.     0.0183 0.     0.     0.0183 0.0183
 0.0183]

Class Limits from data: [11.  14.9 18.8 22.7 26.6 30.5 34.4 38.3 42.2 46.1 50. ]
```

In [74]:
```
## Difference in class limits
mat_plt_lib_clas_lims_diff = np.diff(mat_plt_lib_prob_den[1])
mat_plt_lib_clas_lims_diff
```

Out[74]:
```
array([3.9, 3.9, 3.9, 3.9, 3.9, 3.9, 3.9, 3.9, 3.9, 3.9])
```

In [75]:
```
## Probability density
mat_prob_den_manually = np.round([6/(14*3.9), 3/(14*3.9), 1/(14*3.9), 0/(14*3.9), 1/
                                  1/(14*3.9), 1/(14*3.9)],4)
mat_prob_den_manually
```

Out[75]:
```
array([0.1099, 0.0549, 0.0183, 0.    , 0.0183, 0.    , 0.0183, 0.0183,
       0.0183])
```

In [76]:
```
## Area under the histogram integrates to 1
mat_plt_lib_auc = np.sum(mat_plt_lib_prob_den[0] * np.diff(mat_plt_lib_prob_den[1]))
mat_plt_lib_auc
```

Out[76]:
```
0.9999999999999998
```

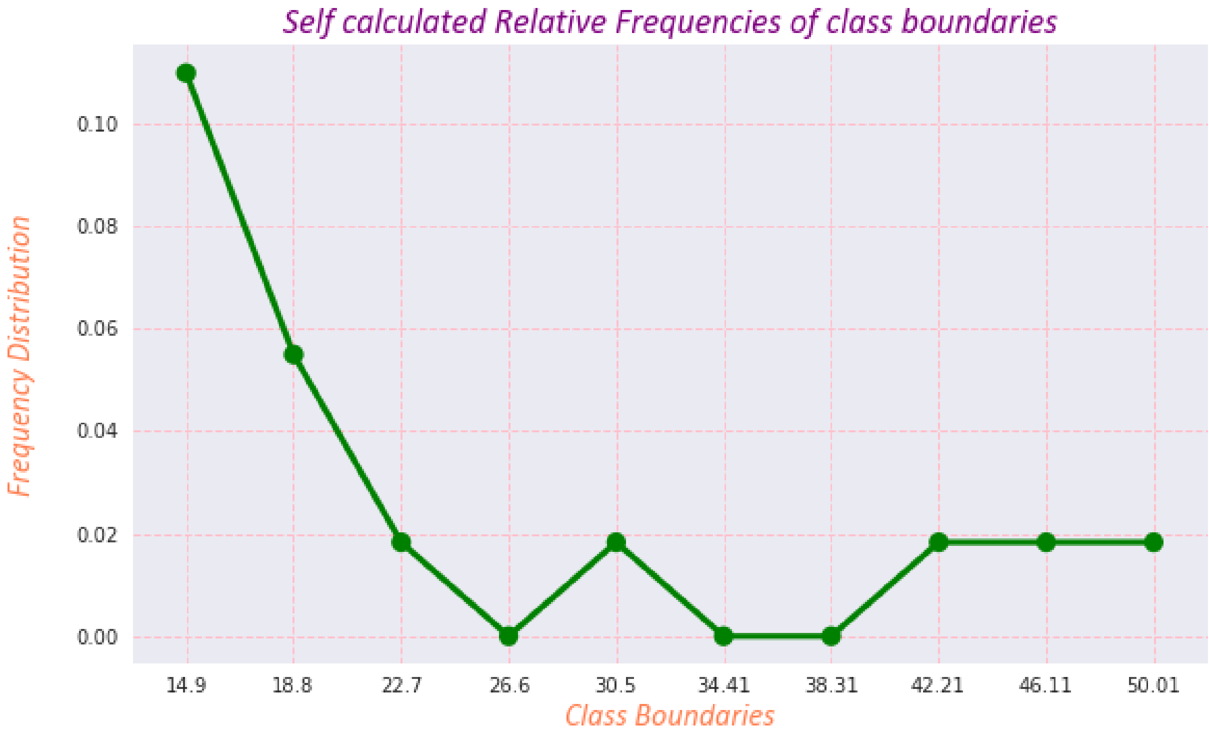
In [77]:
```
## Matplotlib Cumulative Probability Densities
mat_plt_lib_cum_prob_densities = np.cumsum(mat_plt_lib_prob_den[0] * np.diff(mat_plt
mat_plt_lib_cum_prob_densities
```

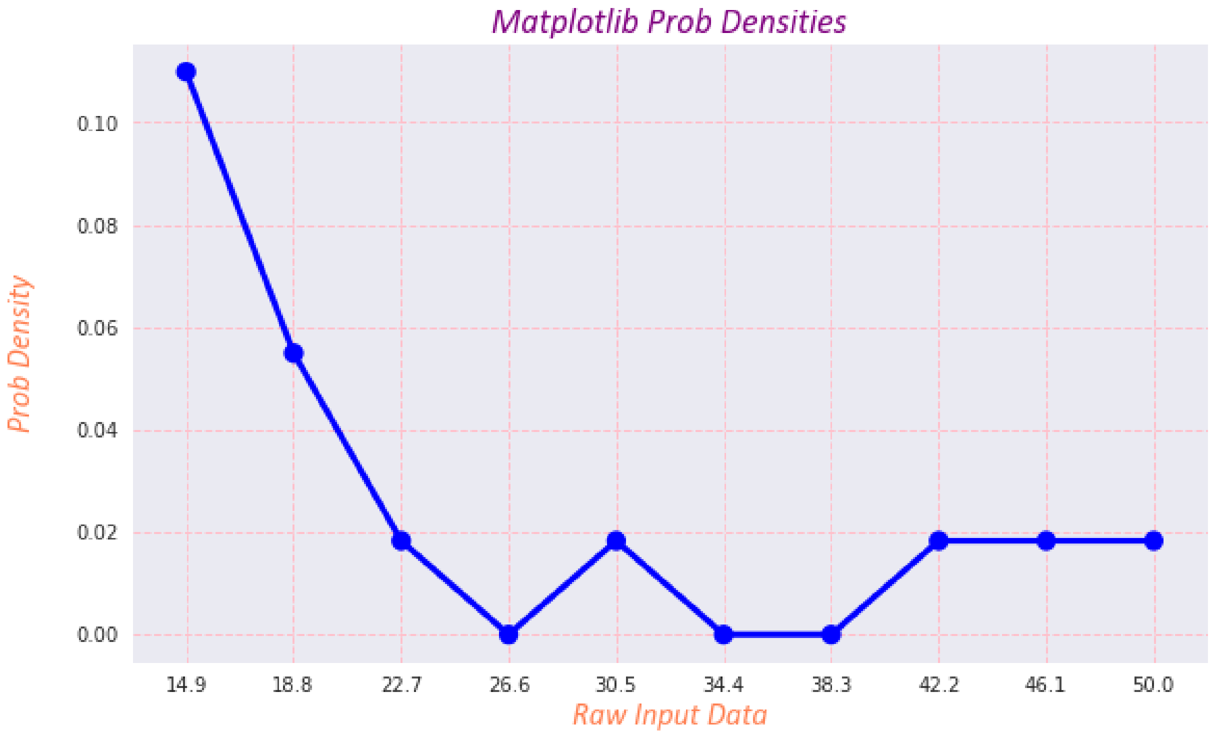
Out[77]:
```
array([0.42857143, 0.64285714, 0.71428571, 0.71428571, 0.78571429,
       0.78571429, 0.78571429, 0.85714286, 0.92857143, 1.        ])
```

In [78]:
```
## Self calculated cumulative relative frequencies
hist_data_results['Cum_Rel_Freq'] = np.cumsum(hist_data_results['Relative_Freq'])
hist_data_results['Cum_Rel_Freq']
```

Out[78]:
```
0    0.428571
1    0.642857
2    0.714286
3    0.714286
4    0.785714
5    0.785714
6    0.785714
7    0.857143
8    0.928571
9    1.000000
Name: Cum_Rel_Freq, dtype: float64
```

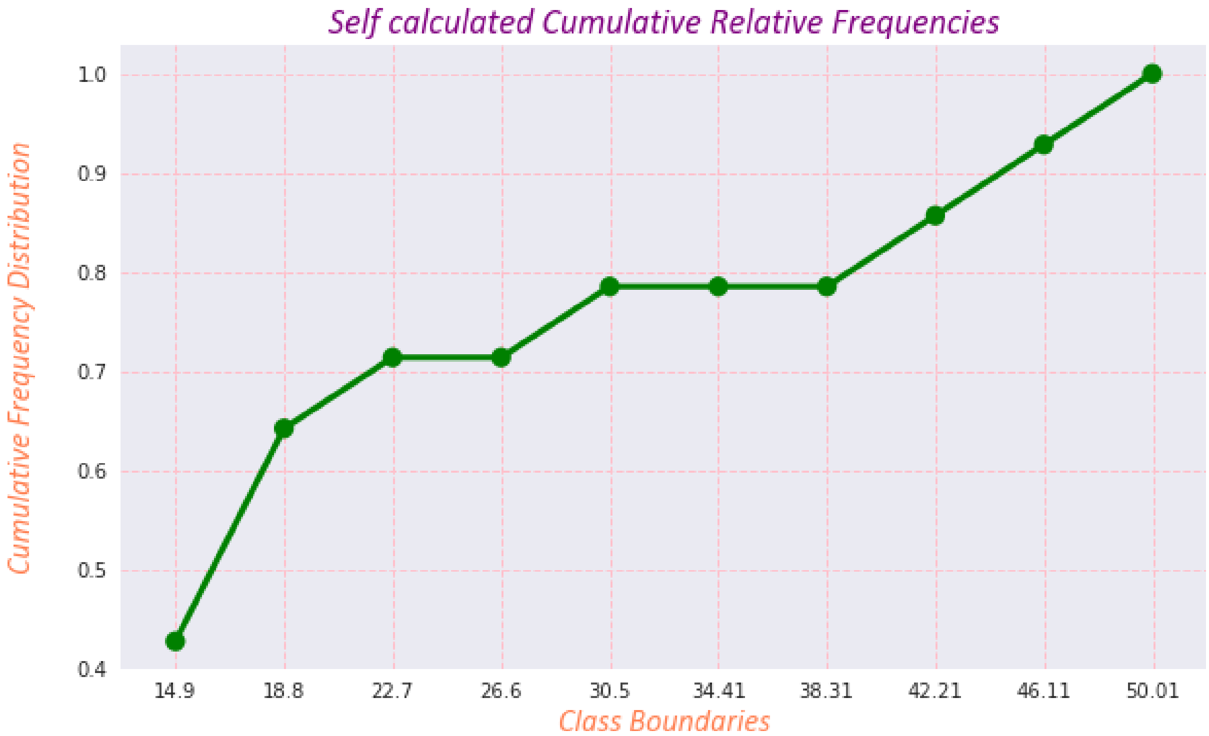
In [79]:
```
## Comparison b/w Self calculated Relative Frequencies & Matplotlib generated Probab
with plt.style.context('seaborn'):
    fig,ax = plt.subplots(nrows=2,ncols=1,figsize=(10,13))
    sns.pointplot(x=bins_intervals[1:],y=bins_prob_density,label='Relative Freq',col
    ax[0].grid(which='major',linestyle='--',color='pink')
    ax[0].set_title("Self calculated Relative Frequencies of class boundaries",
                    fontdict={'family':'calibri','size':18,'style':'oblique','color'
    ax[0].set_xlabel("Class Boundaries",fontdict={'family':'calibri','size':16,'styl
    ax[0].set_ylabel("Frequency Distribution\n",fontdict={'family':'calibri','size':

    sns.pointplot(x=mat_plt_lib_prob_den[1][1:],y=mat_plt_lib_prob_den[0],
                  label='Matplotlib Prob Density',color='blue',ax=ax[1])
    ax[1].grid(which='major',linestyle='--',color='pink')
    ax[1].set_title("Matplotlib Prob Densities",fontdict={'family':'calibri','size':
    ax[1].set_xlabel("Raw Input Data",fontdict={'family':'calibri','size':16,'style'
    ax[1].set_ylabel("Prob Density\n",fontdict={'family':'calibri','size':16,'style'
```

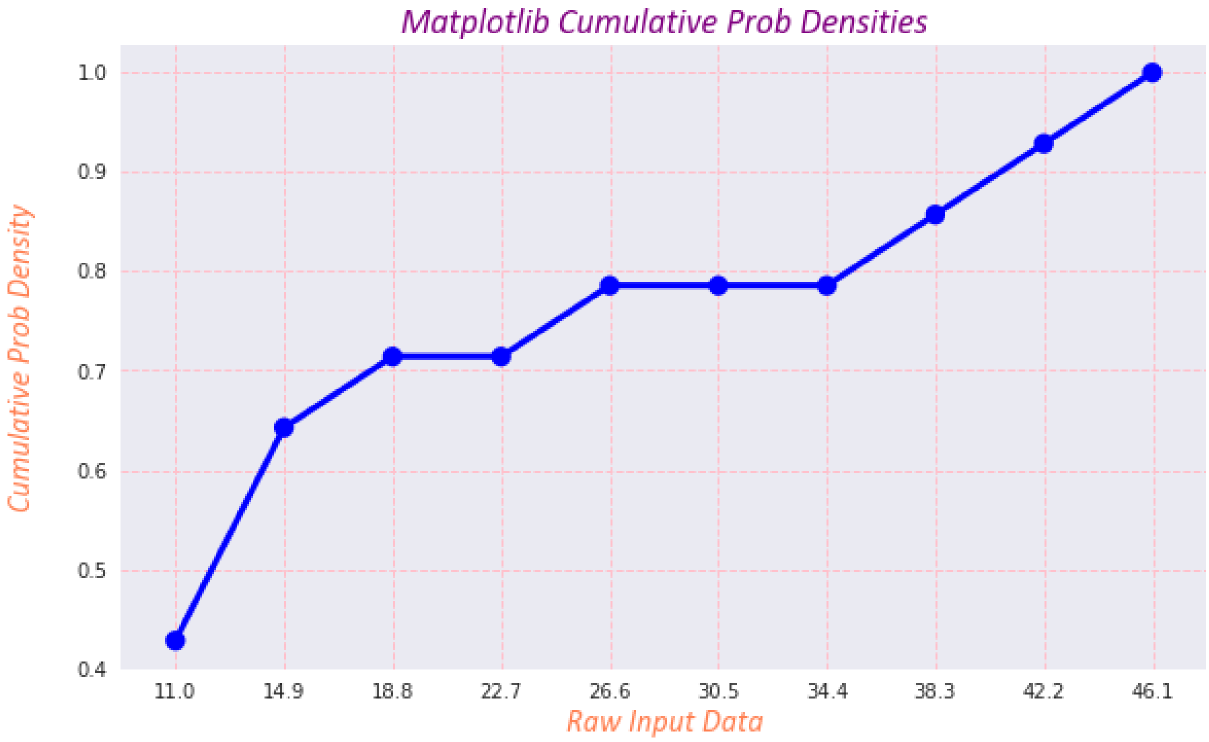




In [80]:

```
## Comparison b/w Self calculated Cumulative Relative Frequencies & Matplotlib gener
with plt.style.context('seaborn'):
    fig,ax = plt.subplots(nrows=2,ncols=1,figsize=(10,13))
    sns.pointplot(x=bins_intervals[1:],y=hist_data_results['Cum_Rel_Freq'],
                  label='Cumulative Relative Freq',color='green',ax=ax[0])
    ax[0].grid(which='major',linestyle='--',color='pink')
    ax[0].set_title("Self calculated Cumulative Relative Frequencies",
                    fontdict={'family':'calibri','size':18,'style':'oblique','color'
    ax[0].set_xlabel("Class Boundaries",fontdict={'family':'calibri','size':16,'styl
    ax[0].set_ylabel("Cumulative Frequency Distribution\n",fontdict={'family':'calib

    sns.pointplot(y=mat_plt_lib_cum_prob_densities,x=mat_plt_lib_prob_den[1][0:-1],
                  label='Matplotlib Cumulative Prob Density',color='blue',ax=ax[1])
    ax[1].grid(which='major',linestyle='--',color='pink')
    ax[1].set_title("Matplotlib Cumulative Prob Densities",fontdict={'family':'calib
    ax[1].set_xlabel("Raw Input Data",fontdict={'family':'calibri','size':16,'style'
    ax[1].set_ylabel("Cumulative Prob Density\n",fontdict={'family':'calibri','size'
```





Bingo!! Above graphs totally matched with the Matplotlib!!

Kernel_Density_Estimator

In [81]:
```
Image("KDE.png",width=1200,height=1200)
```

Out[81]:

KDE

Gaussian Kernels

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

if $\mu = 0$ & $\sigma = 1$

$$\therefore f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-0)^2}{2(1)^2}}$$

$$\left\{ \therefore f(x) = \frac{1}{\sqrt{2\pi}} e^{-[0.5(x)^2]} \right\}$$

$$\left\{ KDE = f_k(x) = \frac{1}{n.h} \sum_{i=1}^{n} \left[\frac{(x - x_i)}{h} \right] \right\}$$

Q How to apply KDE in practice??

Ans inp_data = [. . . . .] ⇒ (Represent x_i)

n = len(inp_data)

X = np.linspace[min(inp_data), max(inp_data), 50]

$$Kde = f_k(x) = \frac{1}{nh} \sum_{i=1}^{n} \left[\frac{(X - x_i)}{h} \right]$$

→ band_width

Calculate

```
res=0
for x in X:
    for xi in inp_data:
        res += (x - xi) / h
    res /= (n.h)
```

⇒ Run for every xi in inp_data

Run 50 times for every x in X

this we can actually feed into gaussian kernels.

eg $f(x) = \frac{1}{\sqrt{2\pi}} e^{-(0.5 . x^2)}$

this can actually be treated as:-

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\left[0.5\left(\frac{x - x_i}{h}\right)^2\right]}$$

this will return a value.

$\therefore f(x) =$ ...... = res

then, res /= (n.h)

this comes from KDE fun

In [82]:

```
hist_data_results
```

Out[82]:

	Class_Distribution	Frequency	Class_Boundaries	Relative_Freq	Prob_Density	Cum_Rel_Freq
0	[11, 14.901]	6	[11.0 - 14.901]	0.428571	0.109862	0.428571
1	[14.901, 18.802]	3	[14.901 - 18.802]	0.214286	0.054931	0.642857
2	[18.802, 22.703]	1	[18.802 - 22.703]	0.071429	0.018310	0.714286
3	[22.703, 26.604]	0	[22.703 - 26.604]	0.000000	0.000000	0.714286
4	[26.604, 30.505]	1	[26.604 - 30.505]	0.071429	0.018310	0.785714
5	[30.505, 34.406]	0	[30.505 - 34.406]	0.000000	0.000000	0.785714
6	[34.406, 38.307]	0	[34.406 - 38.307]	0.000000	0.000000	0.785714
7	[38.307, 42.208]	1	[38.307 - 42.208]	0.071429	0.018310	0.857143
8	[42.208, 46.108999999999995]	1	[42.208 - 46.108999999999995]	0.071429	0.018310	0.928571
9	[46.108999999999995, 50.00999999999999]	1	[46.108999999999995 - 50.00999999999999]	0.071429	0.018310	1.000000

In [83]:

```
def gauss_kernels(x):
    """
    Description: This function is created for generating the gaussian kernels.
    Input: It accepts one parameter:
        1. data: Value for which gaussian kernels to be generated
    Return: Gaussian Kernel
    """
    left_half = 1/math.sqrt(2*math.pi)
    second_half = np.exp((-0.5)*(x**2))
    gauss = left_half * second_half
    return gauss

def prob_distn_func(x_linspaced,inp_data,h=1):
    """
    Description: This function is created for performing the kernel density estimati
    Input: It accepts 3 input parameters:
        1. x_linspaced: int/float
            It represents x in the KDE formula
        2. inp_data: np.array
            It represents x_i in the KDE formula
        3. h: int/float
            It represents bandwidth of gaussian kernels
    """
    ## Total elements in the data
    n = len(inp_data)
    if len(inp_data) == 0:
        return 0

    ## Performing KDE estimation
    kde_value = 0
    for idx,x_i in enumerate(inp_data):
        kde_value += gauss_kernels(np.divide((x_linspaced - x_i),h))
    kde_value /= (n*h)
    return kde_value

def density_plot(inp_data,use_external_h=False,h=0.05,bins=10):
    """
    Description: This function is created for generating the KDE plot.
    Input: It accepts 5 input parameters:
        1. inp_data: np.array
```

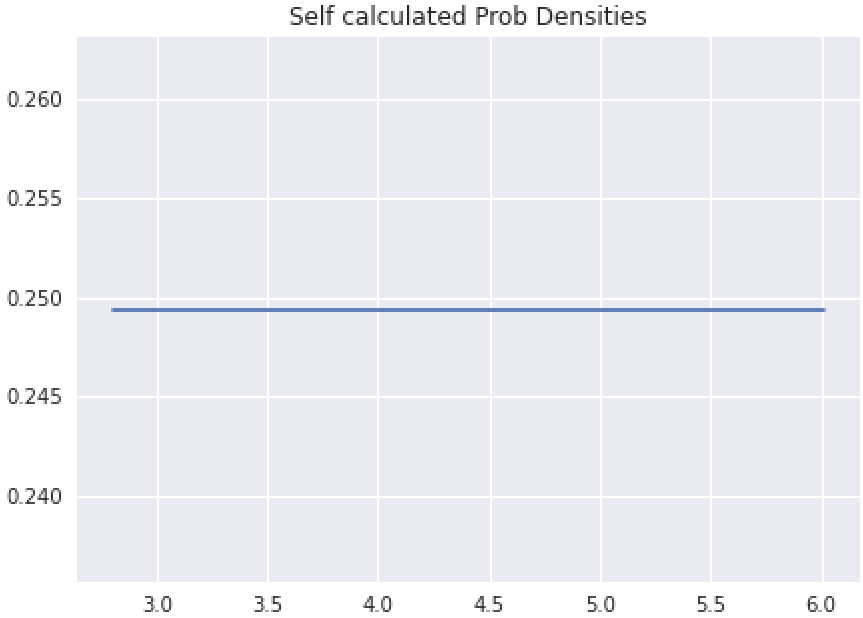
```
            Data for which density plot to be generated and it represents x_i in the
        2. use_external_h: boolean
            Flag for User-defined bandwidth of gaussian kernels
        3. h: int/float
            It represents bandwidth of gaussian kernels
        4. bins: integer
            Bins for plotting the histogram
    """
    if use_external_h:
        bandwidth=h
    else:
        ## Calculating the bandwidth = C * (n)^(-1/5)
        ##### Here, C = 1.05 * stddev(data) and n = len(data)
        bandwidth= 1.05 * np.std(inp_data) * (len(inp_data)**(-1/5))

    ## Generating linearly spaced x's
    x_linspace=np.linspace(min(inp_data),max(inp_data),50)

    ## Applying the Density Estimator
    y_prob_densities=[prob_distn_func(x_linspace[i],inp_data,bandwidth) for i in ran

    ## Plotting the Histogram and Density Estimation
    with plt.style.context('seaborn'):
        plt.figure(figsize=(8,6))
        plt.hist(inp_data,bins=bins,density=True,color='lightblue',label='Histogram'
        plt.plot(x_linspace.tolist(),y_prob_densities,color='black',linestyle='-',la
        plt.grid(which='major',linestyle='--',color='pink')
        plt.xlabel("Data Values",fontdict={'family':'calibri','size':17,'style':'obl
        plt.title("KDE Estimation plot",fontdict={'family':'calibri','size':18,'styl
        plt.legend()
```

PMF:Discrete_Variable

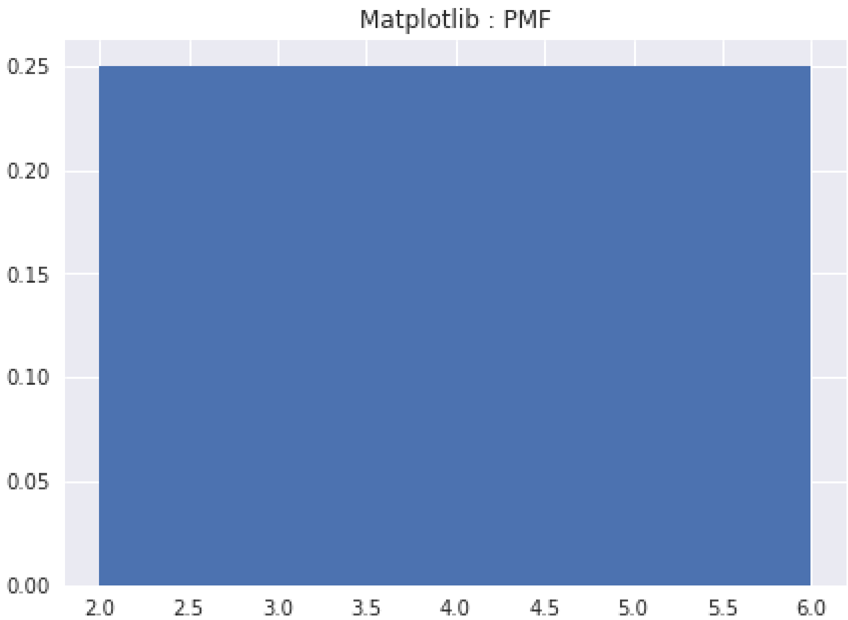
In [84]:
```
data_dv=[2,3,4,5,6]
```

In [85]:
```
a_dv,b_dv,p_dv = plot_hist(data=data_dv,number_of_bins=5)
```

In [86]:
```
with plt.style.context('seaborn'):
    plt.figure(figsize=(7,5))
    plt.plot(b_dv[1:],p_dv)
    plt.title("Self calculated Prob Densities");
```

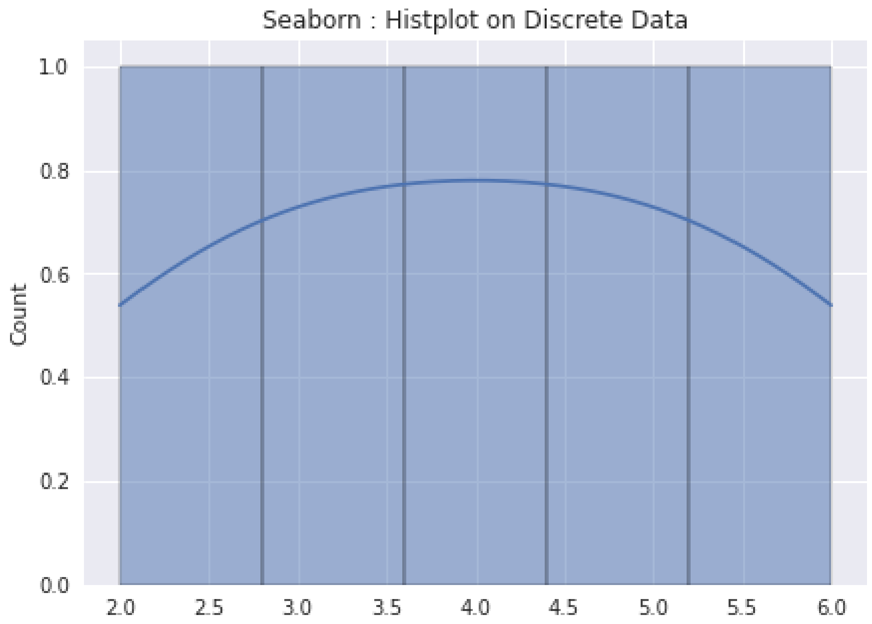


In [87]:
```
with plt.style.context('seaborn'):
    plt.figure(figsize=(7,5))
    plt.hist(data_dv,bins=5,density=True)
    plt.title("Matplotlib : PMF");
```

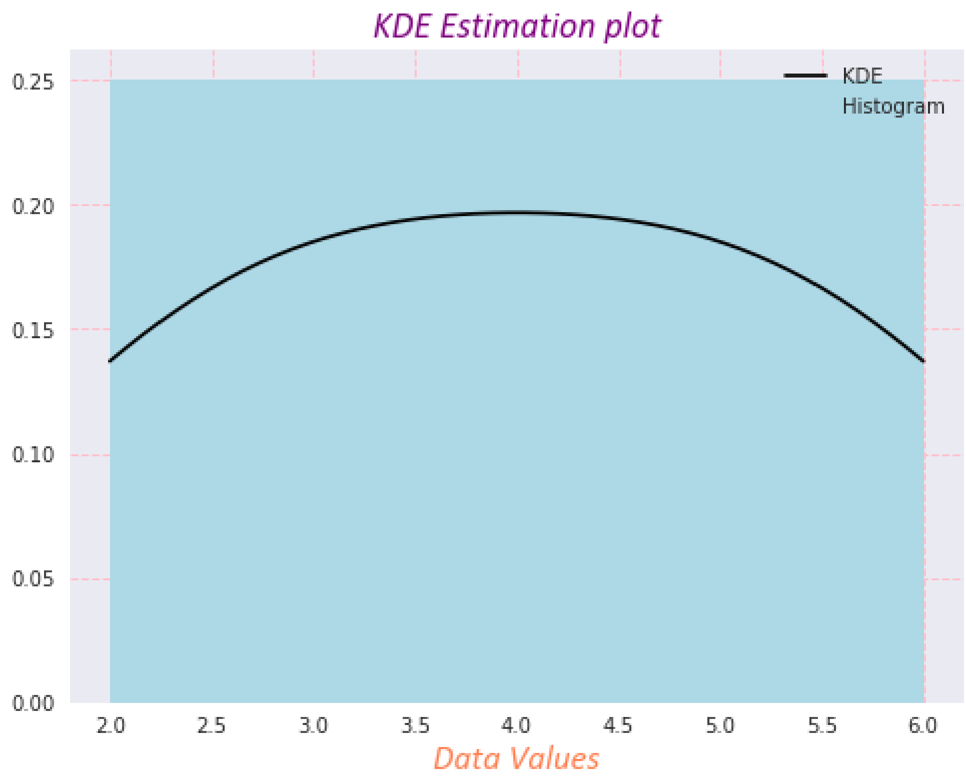


As we know in PMF, we have equi-probable values.

In [88]:
```
with plt.style.context('seaborn'):
    plt.figure(figsize=(7,5))
    sns.histplot(data_dv,bins=5,kde=True)
    plt.title("Seaborn : Histplot on Discrete Data");
```



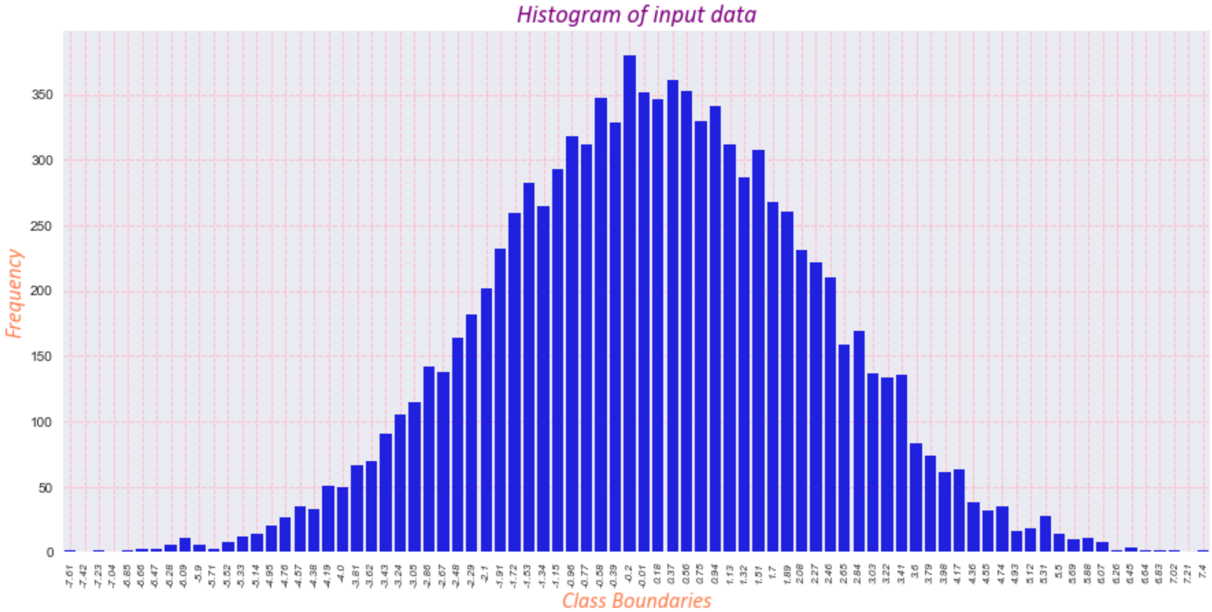
In [89]:
```
density_plot(data_dv,bins=5)
```



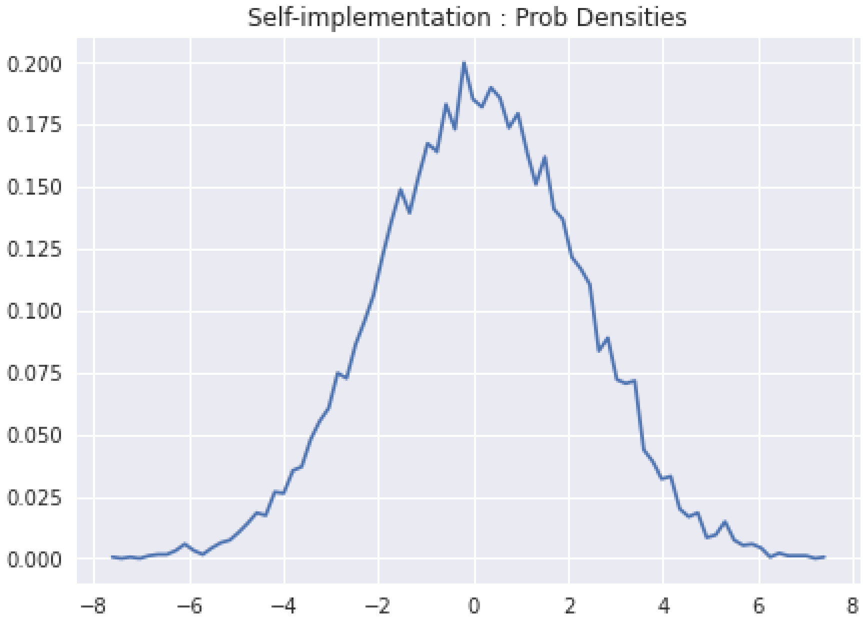
PDF:Continuous_Random_Variable(Gaussian)

In [90]:
```
data_cv = np.random.normal(size=10000)*2.1
```

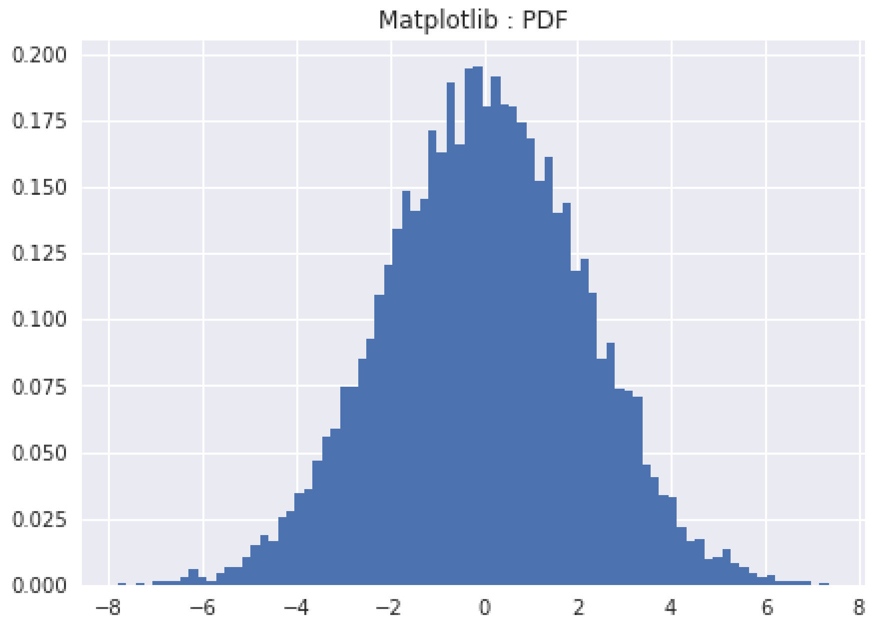
In [91]:
```
a_cv,b_cv,p_cv = plot_hist(data=data_cv,number_of_bins=80)
```



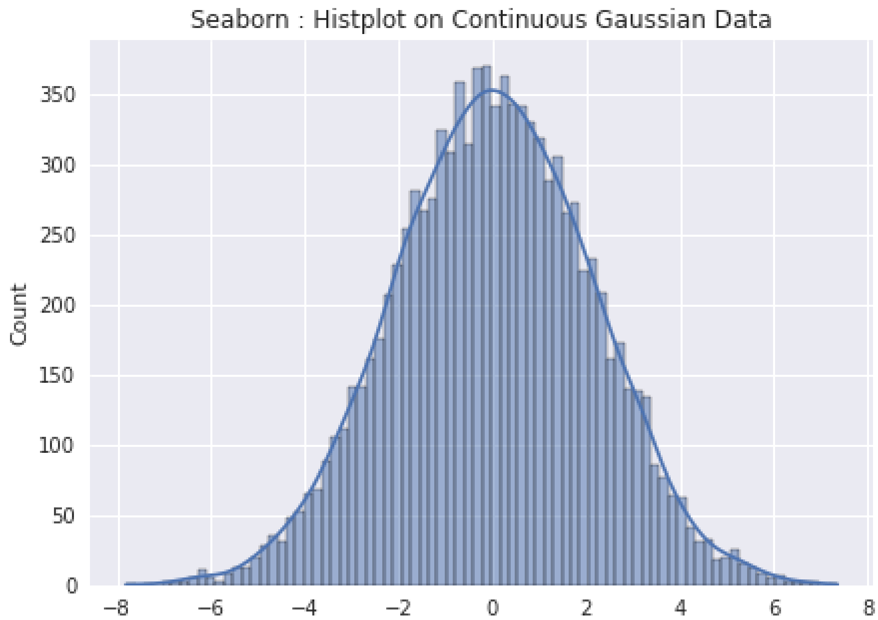
In [92]:
```
with plt.style.context('seaborn'):
    plt.figure(figsize=(7,5))
    plt.plot(b_cv[1:],p_cv)
    plt.title("Self-implementation : Prob Densities");
```



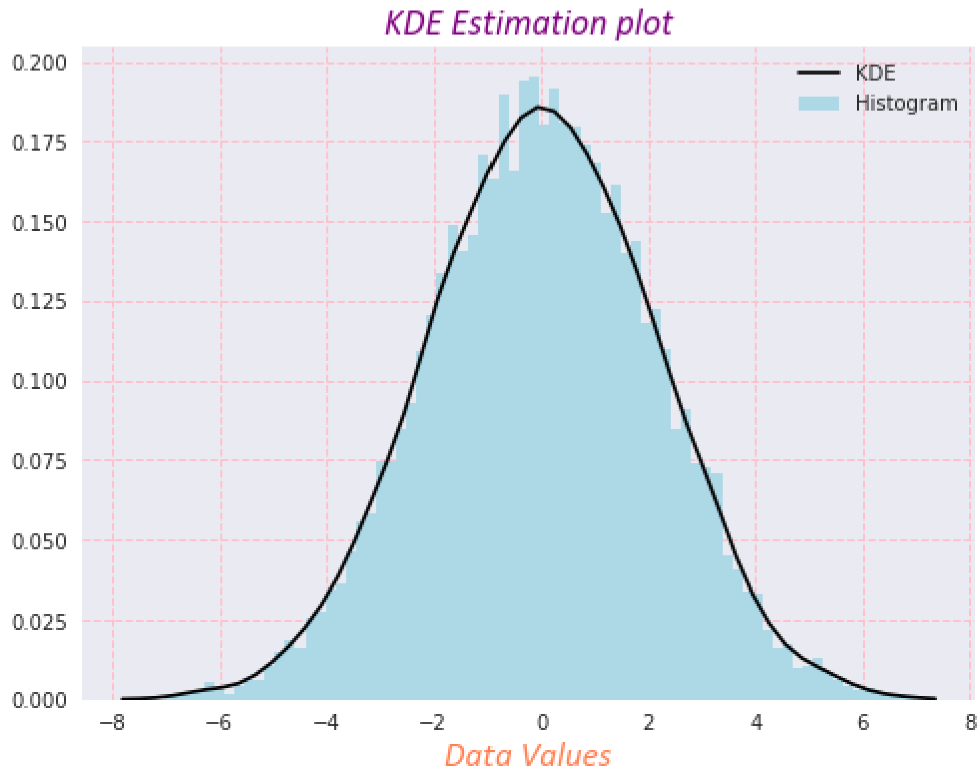
In [93]:
```
with plt.style.context('seaborn'):
    plt.figure(figsize=(7,5))
    plt.hist(data_cv,bins=80,density=True)
    plt.title("Matplotlib : PDF");
```



In [94]:
```
with plt.style.context('seaborn'):
    plt.figure(figsize=(7,5))
    sns.histplot(data_cv,bins=80,kde=True)
    plt.title("Seaborn : Histplot on Continuous Gaussian Data");
```



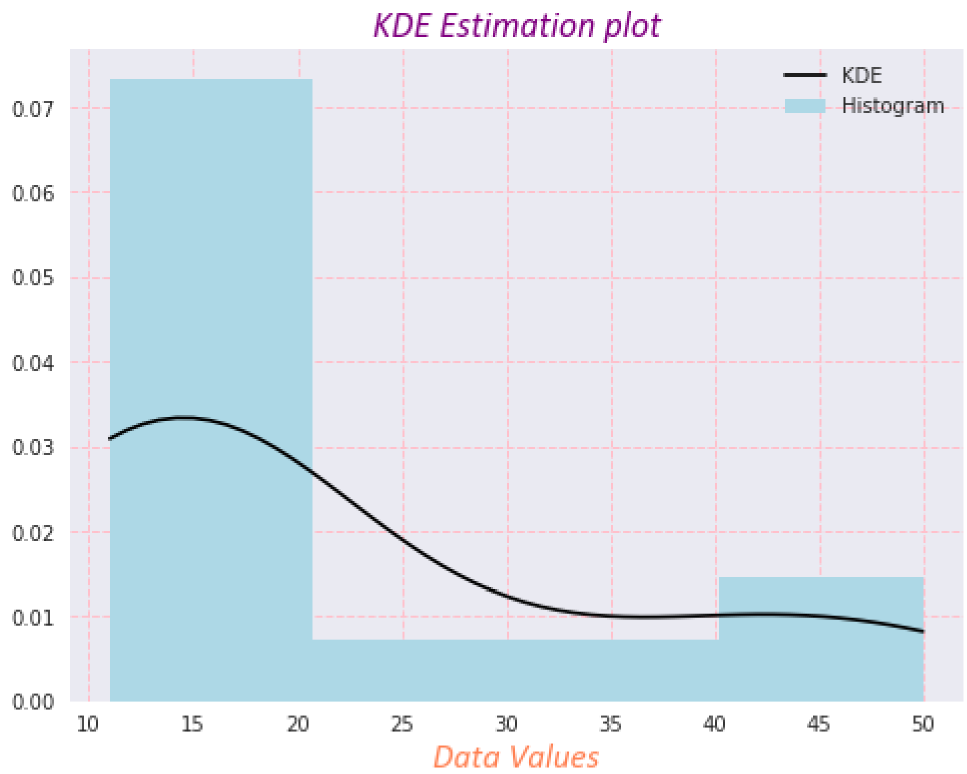
In [95]:
```
density_plot(data_cv,bins=80)
```



Above shows the normal bell-shaped curve.

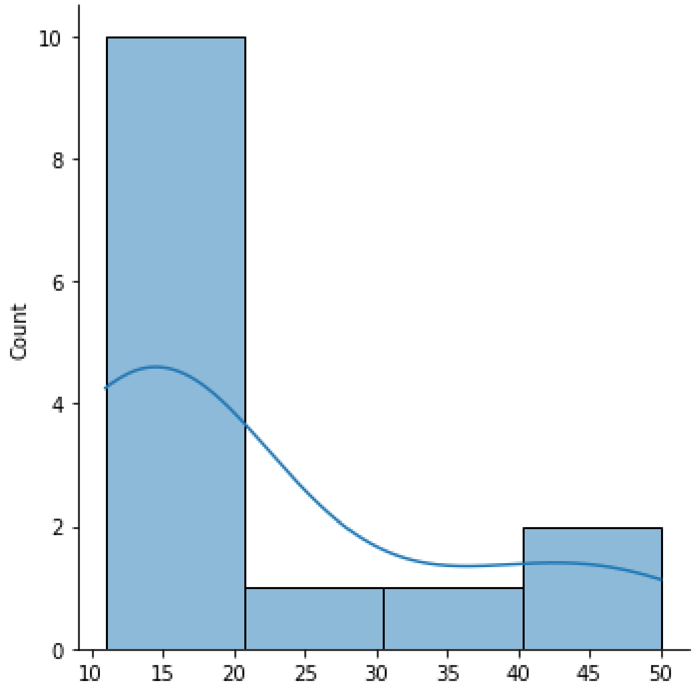
KDE on Small Random Dataset

In [96]:
```
## Self-implemented KDE function
density_plot(hist_data,use_external_h=False,h=5,bins=4)
```



In [97]:
```
## Seaborn Displot
sns.displot(hist_data,kde=True,bins=4)
```

Out[97]: `<seaborn.axisgrid.FacetGrid at 0x2199ca200b8>`

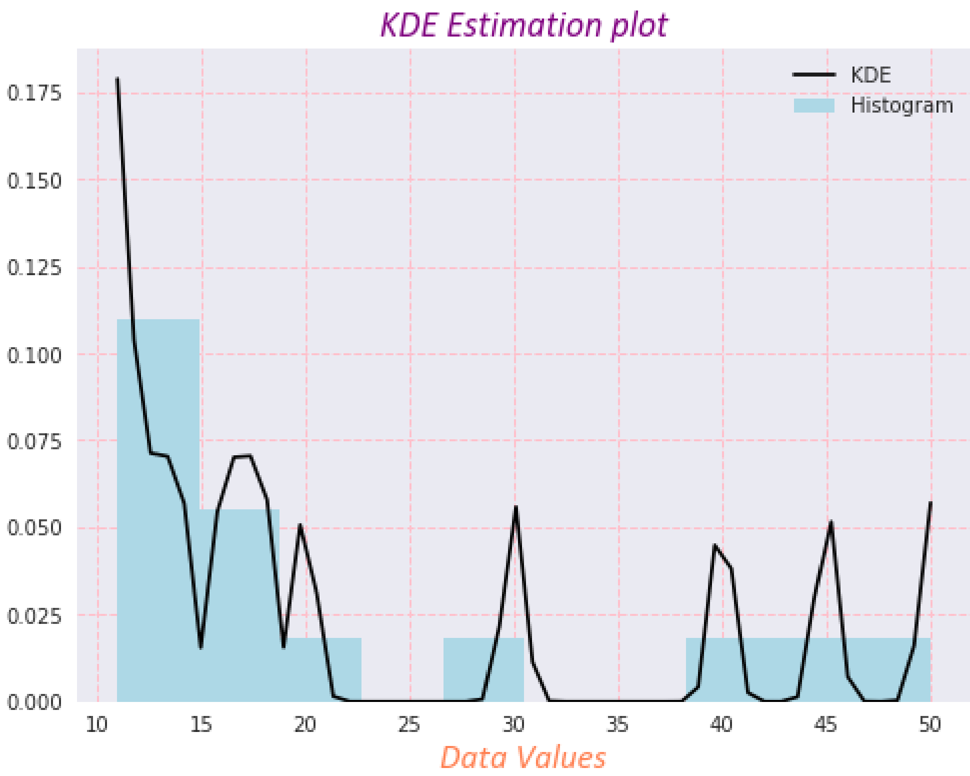


Effect_of_H_in_KDE

Effect of lower or higher value of bandwidth on KDE?

Smaller Value

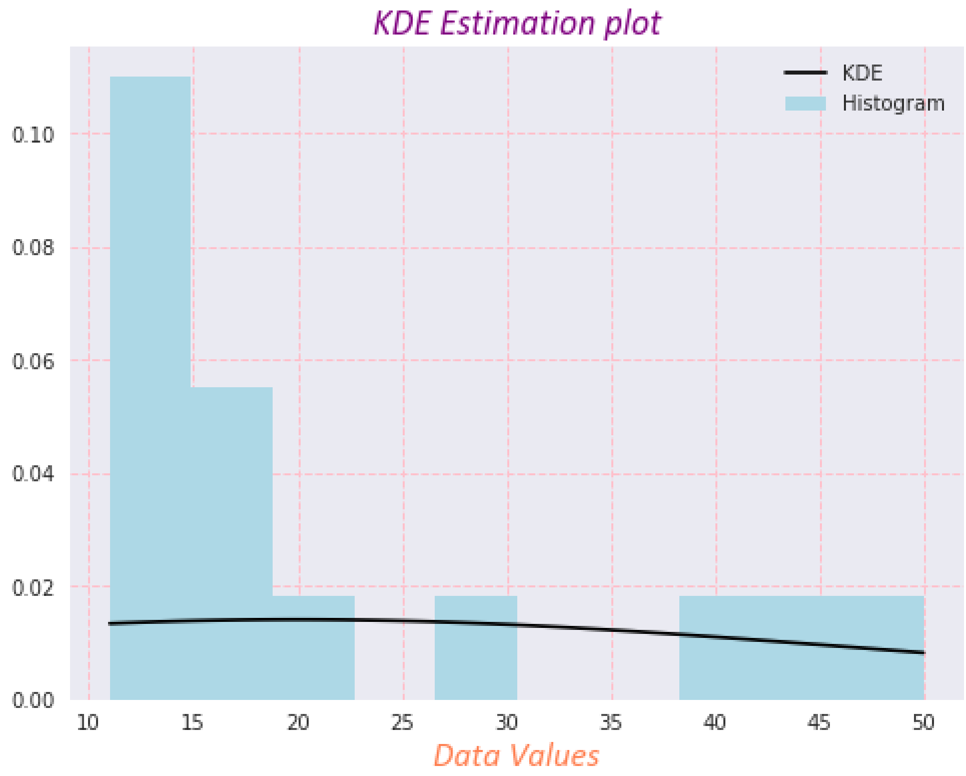
In [98]:
```
density_plot(hist_data,use_external_h=True,h=0.5)
```



Smaller value of h gives squiggly plot.

Higher Value

In [99]:
```
density_plot(hist_data,use_external_h=True,h=25)
```



Larger value of h gives flat plot.

PDF_from_CDF

Example:1

In [100…
```
cdfs = hist_data_results['Cum_Rel_Freq'].values
```

In [101…
```
cdfs, len(cdfs)
```

Out[101…
```
(array([0.42857143, 0.64285714, 0.71428571, 0.71428571, 0.78571429,
        0.78571429, 0.78571429, 0.85714286, 0.92857143, 1.        ]),
 10)
```

In [102…
```
## Manual CDF's from Prob Density
np.cumsum(hist_data_results['Prob_Density']*3.9)
```

Out[102…
```
0    0.428462
1    0.642692
2    0.714103
3    0.714103
4    0.785513
5    0.785513
6    0.785513
7    0.856923
8    0.928333
9    0.999744
Name: Prob_Density, dtype: float64
```

In [103…
```
def cal_pdf_from_cdf(cdfs,var='cv',del_h=1.0):
    """
    Description: This function is created for calculating the PDF's value from CDF.

    Input: It accepts 1 parameter:
        1. cdfs : list/array
            Cumulative Frequencies
        2. var : Defines kind of variable
            By default 'cv'
        3. del_h : Bin_width or Diff b/w intervals or class limits
```

```
            By default 1.0

    Return: It returns the pdf values.
        1. prob_density_from_cdf : list
    """
    prob_density_from_cdf = []

    if var == 'dv':
        prob_density_from_cdf.append(cdfs[0])
        i=0
        while i < len(cdfs)-1:
            pdf = (cdfs[i+1]-cdfs[i])
            prob_density_from_cdf.append(pdf)
            i += 1
    elif var == 'cv':
        prob_density_from_cdf.append(np.round((cdfs[0]/del_h),5))
        i=0
        while i < len(cdfs)-1:
            pdf = (cdfs[i+1]-cdfs[i])/del_h
            pdf = np.round(pdf,5)
            prob_density_from_cdf.append(pdf)
            i += 1
    else:
        return None
    return prob_density_from_cdf
```

In [104…

```
## Self-implemented function :: PDFs from CDF
pdf_from_cdf = cal_pdf_from_cdf(cdfs,var='dv')
pdf_from_cdf
```

Out[104…

```
[0.42857142857142855,
 0.21428571428571425,
 0.0714285714285714,
 0.0,
 0.0714285714285714,
 0.0,
 0.0,
 0.0714285714285714,
 0.0714285714285714,
 0.0714285714285714]
```

In [105…

```
## PDFs from CDFs using np.diff
hist_data_results['Relative_Freq'][0],np.diff(np.cumsum(hist_data_results['Prob_Dens
```

Out[105…

```
(0.42857142857142855,
 array([0.21423078, 0.07141026, 0.        , 0.07141026, 0.        ,
        0.        , 0.07141026, 0.07141026, 0.07141026]))
```

Example:2

In [106…

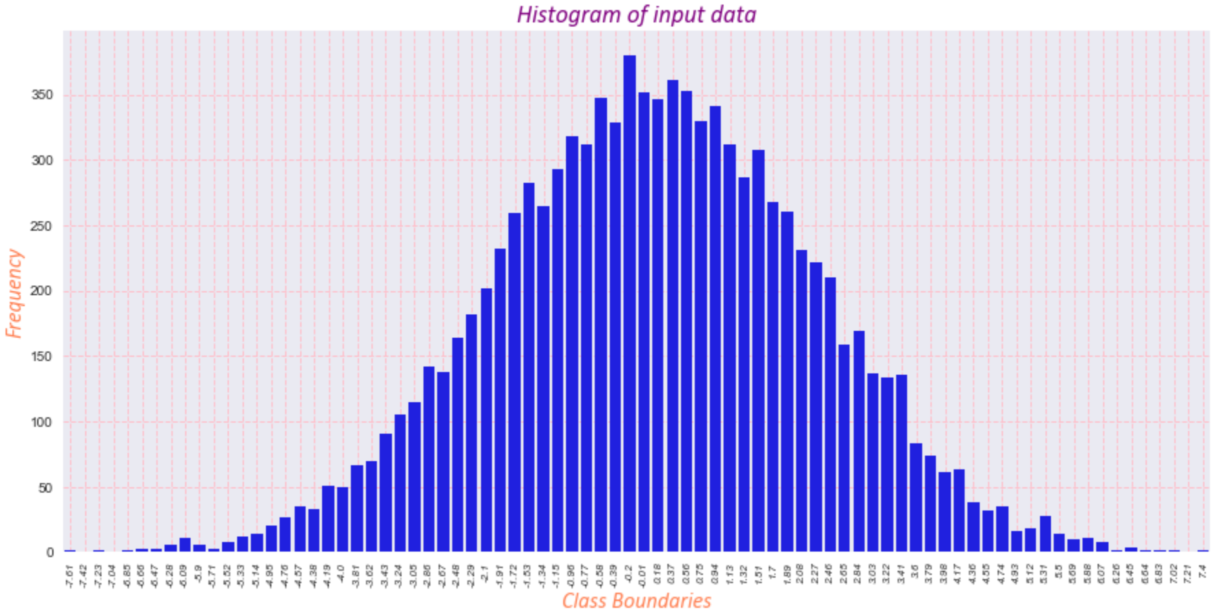
```
data_cv
```

Out[106…

```
array([ 1.27402585, -0.07002967, -2.10500803, ..., -2.8054017 ,
       -0.86908818, -1.27014371])
```

In [107…

```
a_cv,b_cv,p_cv = plot_hist(data=data_cv,number_of_bins=80)
```



In [108…
```
a_cv
```

Out[108…

	Class_Distribution	Frequency	Class_Boundaries	Relative_Freq	Prob_Density
0	[-7.804783794498327, -7.614783794498327]	1	[-7.804783794498327 - -7.614783794498327]	0.0001	0.000526
1	[-7.614783794498327, -7.424783794498326]	0	[-7.614783794498327 - -7.424783794498326]	0.0000	0.000000
2	[-7.424783794498326, -7.234783794498326]	1	[-7.424783794498326 - -7.234783794498326]	0.0001	0.000526
3	[-7.234783794498326, -7.044783794498326]	0	[-7.234783794498326 - -7.044783794498326]	0.0000	0.000000
4	[-7.044783794498326, -6.854783794498325]	2	[-7.044783794498326 - -6.854783794498325]	0.0002	0.001053
...	...	...	...	...	...
75	[6.4452162055016835, 6.635216205501684]	2	[6.4452162055016835 - 6.635216205501684]	0.0002	0.001053
76	[6.635216205501684, 6.825216205501684]	2	[6.635216205501684 - 6.825216205501684]	0.0002	0.001053
77	[6.825216205501684, 7.015216205501685]	2	[6.825216205501684 - 7.015216205501685]	0.0002	0.001053
78	[7.015216205501685, 7.205216205501685]	0	[7.015216205501685 - 7.205216205501685]	0.0000	0.000000
79	[7.205216205501685, 7.3952162055016855]	1	[7.205216205501685 - 7.3952162055016855]	0.0001	0.000526

80 rows × 5 columns

In [109…
```
## Manual Calulation of Prob Density
bin_width = np.round((np.max(data_cv)-np.min(data_cv))/80,4)
print('Bin_width :',bin_width,'\n')
print("Prob Density :\n",a_cv['Frequency']/(a_cv['Frequency'].sum() * bin_width))
```

```
Bin_width : 0.1895

Prob Density :
```

```
0     0.000528
1     0.000000
2     0.000528
3     0.000000
4     0.001055
        ...
75    0.001055
76    0.001055
77    0.001055
78    0.000000
79    0.000528
Name: Frequency, Length: 80, dtype: float64
```

In [110…

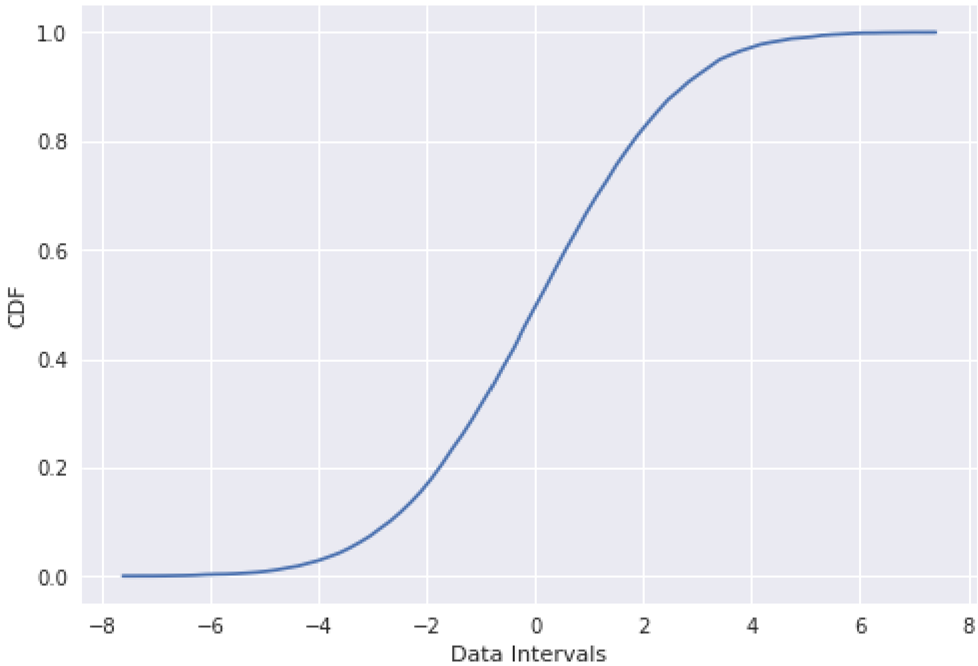
```
## Manual Calulation of CDF ### 0.19 is bins_width
cdfs_cv = np.cumsum((a_cv['Frequency']/(a_cv['Frequency'].sum() * bin_width)) * bin_
cdfs_cv
```

Out[110…

```
0     0.0001
1     0.0001
2     0.0002
3     0.0002
4     0.0004
       ...
75    0.9995
76    0.9997
77    0.9999
78    0.9999
79    1.0000
Name: Frequency, Length: 80, dtype: float64
```

In [111…

```
with plt.style.context('seaborn'):
    plt.plot(b_cv[1:],cdfs_cv)
    plt.xlabel("Data Intervals")
    plt.ylabel("CDF")
```



In [112…

```
## Manual Calculation of PDF from CDF ### 0.19 is bins_width
((a_cv['Frequency']/(a_cv['Frequency'].sum() * bin_width)) * bin_width)/bin_width
```

Out[112…

```
0     0.000528
1     0.000000
2     0.000528
3     0.000000
4     0.001055
        ...
```

```
75    0.001055
76    0.001055
77    0.001055
78    0.000000
79    0.000528
Name: Frequency, Length: 80, dtype: float64
```

In [113…
```
## Calculation of PDF from CDF via Self-implemented function
cal_pdf_from_cdf(cdfs_cv,var='cv',del_h=bin_width)[0:5]
```

Out[113…
```
[0.00053, 0.0, 0.00053, 0.0, 0.00106]
```

Bingo!! All the above result sets matched, so good here!!

Proportional_Sampling

In [114…
```
data = [1,4,-2,6,-1,0]
data_prob = np.square(data)
```

In [115…
```
data, data_prob
```

Out[115…
```
([1, 4, -2, 6, -1, 0], array([ 1, 16,  4, 36,  1,  0], dtype=int32))
```

In [116…
```
def cum_sum(inp_data):
    """
    Description: This function calculates the cumulative sum.
    """
    cum_sum = []
    cum_sum.append(inp_data[0])
    for i in range(1,len(inp_data)):
        cum_sum.append(cum_sum[i-1]+inp_data[i])
    return cum_sum
```

In [117…
```
data_and_probs = {}
cs = cum_sum(data_prob/data_prob.sum())
for idx,val in enumerate(cs):
    data_and_probs[data[idx]]=val

data_and_probs
```

Out[117…
```
{1: 0.017241379310344827,
 4: 0.29310344827586204,
 -2: 0.3620689655172413,
 6: 0.9827586206896551,
 -1: 1.0,
 0: 1.0}
```

In [118…
```
diff = []
diff.append(data_and_probs[1])
vals=np.diff(list(data_and_probs.values()))
for val in vals:
    diff.append(val)
```

In [119…
```
for i,val in enumerate(data):
    print(val,'----',diff[i])
```

```
1 ---- 0.017241379310344827
4 ---- 0.27586206896551724
-2 ---- 0.06896551724137928
6 ---- 0.6206896551724138
-1 ---- 0.017241379310344862
0 ---- 0.0
```

The above result clears all the confusion here, as the probability weights of "4" and "6" are higher thus we have larger differences for these two values which tells us that they are covering more values from 0 to 1.

In [120…

```
def proportional_sampling(data_vals, data_vals_probs):
    """
    Description: This function is performing the proportional sampling.
    """
    data_proportions = data_vals_probs/data_vals_probs.sum()
    cume_sum = cum_sum(data_proportions)
    ## Pick a Random number b/w 0 and 1
    rn_num = uniform(0,1)
    print("Random Number --",rn_num)
    for idx, val in enumerate(cume_sum):
        if rn_num <=  val:
            print("Data value picked --",data_vals[idx])
            return data_vals[idx]
```

In [121…

```
sampled_value = proportional_sampling(data, data_prob)
```

```
Random Number -- 0.07384749525018264
Data value picked -- 4
```

In [122…

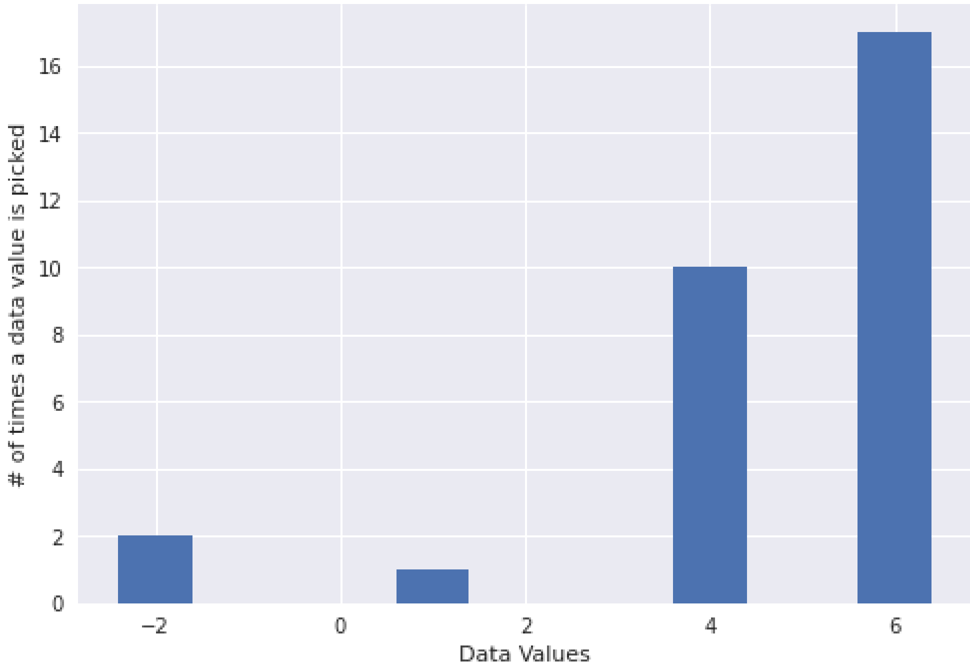
```
data_prop_sampling_result = {}
for i in range(0, 30):
    sampled_value = proportional_sampling(data, data_prob)
    if sampled_value not in data_prop_sampling_result:
        data_prop_sampling_result[sampled_value] = 1
    else:
        data_prop_sampling_result[sampled_value] += 1
```

```
Random Number -- 0.9174058009902165
Data value picked -- 6
Random Number -- 0.501052011437288
Data value picked -- 6
Random Number -- 0.8211076715122755
Data value picked -- 6
Random Number -- 0.29814415956857476
Data value picked -- -2
Random Number -- 0.1288518149907184
Data value picked -- 4
Random Number -- 0.22863852910810123
Data value picked -- 4
Random Number -- 0.9384919691364035
Data value picked -- 6
Random Number -- 0.9442553045916641
Data value picked -- 6
Random Number -- 0.07946361269854318
Data value picked -- 4
Random Number -- 0.18874628869036325
Data value picked -- 4
Random Number -- 0.7205152225942119
Data value picked -- 6
Random Number -- 0.6988383864408874
Data value picked -- 6
Random Number -- 0.6747278740427888
Data value picked -- 6
Random Number -- 0.20856858855216376
Data value picked -- 4
Random Number -- 0.0461866257517457
Data value picked -- 4
Random Number -- 0.7344352564592898
Data value picked -- 6
Random Number -- 0.08066772346306927
Data value picked -- 4
Random Number -- 0.566592464181382
Data value picked -- 6
```

```
Random Number -- 0.24446668896872048
Data value picked -- 4
Random Number -- 0.7108010179397969
Data value picked -- 6
Random Number -- 0.31049459019347503
Data value picked -- -2
Random Number -- 0.8859116534509489
Data value picked -- 6
Random Number -- 0.8860102662218309
Data value picked -- 6
Random Number -- 0.4888701291318224
Data value picked -- 6
Random Number -- 0.2504436141560329
Data value picked -- 4
Random Number -- 0.8840963497342477
Data value picked -- 6
Random Number -- 0.008483182093586894
Data value picked -- 1
Random Number -- 0.8850955194373896
Data value picked -- 6
Random Number -- 0.3916379950745654
Data value picked -- 6
Random Number -- 0.18795095049353694
Data value picked -- 4
```

In [123…

```
with plt.style.context('seaborn'):
    plt.bar(data_prop_sampling_result.keys(),data_prop_sampling_result.values())
    plt.xlabel("Data Values")
    plt.ylabel("# of times a data value is picked")
```



Clearly, 4 and 6 are mostly picked values.

Reference Links

- **Probability Density** & **Relative Frequencies**
  - https://stackoverflow.com/questions/41974615/how-do-i-calculate-pdf-probability-density-function-in-python
  - https://www.quora.com/What-is-the-distinction-between-a-probability-distribution-and-a-relative-frequency-distribution
- **KDE implementation**

- https://medium.com/analytics-vidhya/kernel-density-estimation-kernel-construction-and-bandwidth-optimization-using-maximum-b1dfce127073
- https://www.programmersought.com/article/52286021603/