Data Science Analysis Assignment - 6

In [13]:

```
#importing required libraries
from scipy.stats import norm

import pandas as pd
import numpy as np
from scipy import optimize

import seaborn as sb
import matplotlib.pyplot as plt
import emcee
import corner
```

Q1.

In [2]:

```
#given data
eins_p = 1.74
newt_p = eins_p/2

eddi_t = 1.61
eddi_e = 0.40

crom_t = 1.98
crom_e = 0.16

#calcualting bayes factor
bf_eddi = norm.pdf(eddi_t, eins_p, eddi_e)/norm.pdf(eddi_t, newt_p, eddi_e)
bf_crom = norm.pdf(crom_t, eins_p, crom_e)/norm.pdf(crom_t, newt_p, crom_e)

#printing the data
print("Bayes Factor from Eddington's measurement is %s." %(bf_eddi))
print("Bayes Factor from Crommelin's measurement is %s." %(bf_crom))
```

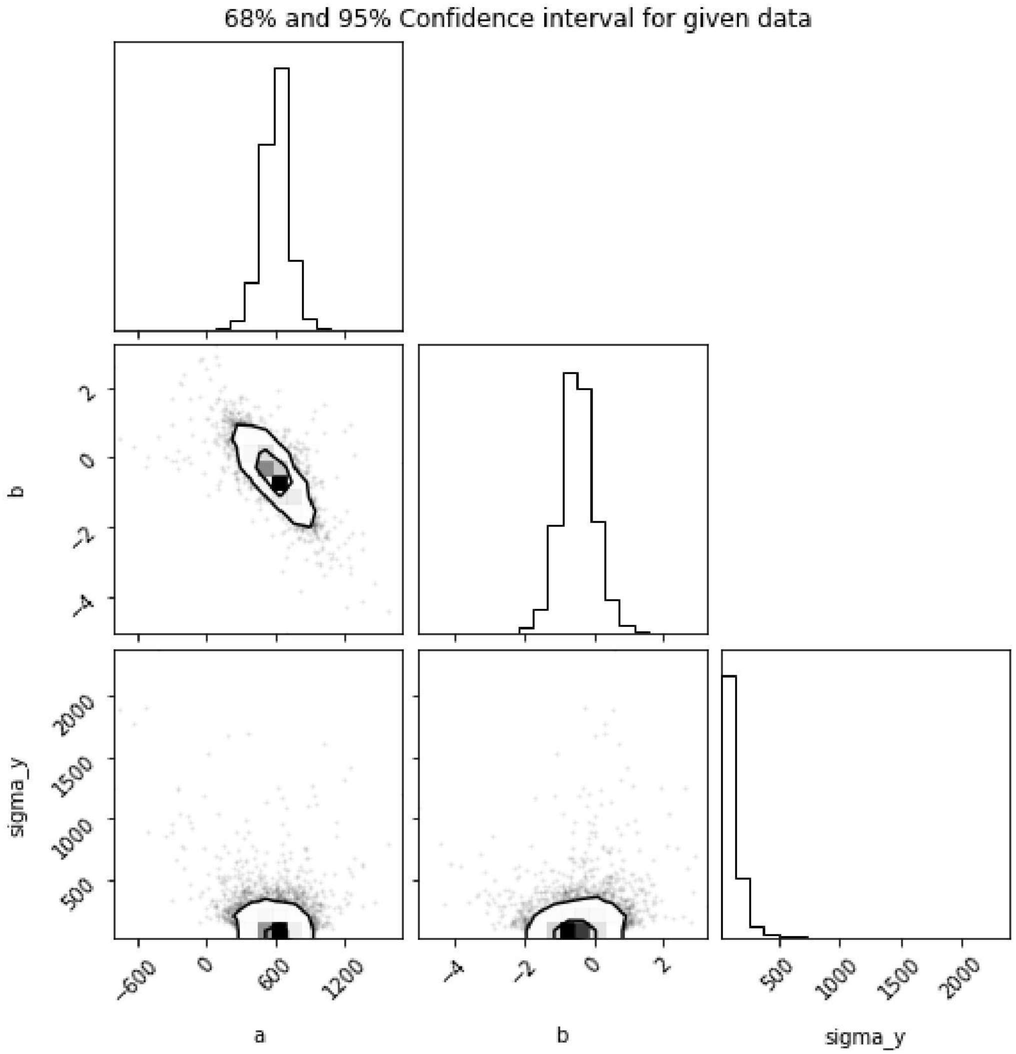
```
Bayes Factor from Eddington's measurement is 5.25109958796716.
Bayes Factor from Crommelin's measurement is 9172292802.960836.
```

Q2.

In [8]:

```
#reading data from csv file using pandas after ignoring first 4 data points
df = pd.read_csv("D:\CLASSES\SEM 4\Data Science Analysis\A6\q2_data.csv")[:4]

#storing the data into arrays
x = df['x'].values
y = df['y'].values
sigma_y = df['error in y'].values

#total no. of model parameter
num_dim = 3
#burn period for chain stabilization
num_burn_period = 1000
#MCMC steps
num_steps = 5000
#MCMC walkers
num_walkers = 50

#defining log-prior function same as the one in JVDP's blog article
def log_prior(theta):
    alpha, beta, sigma = theta
    if sigma < 0:
        return -np.inf   ##log(0)##
    else:
        return -1.5 * np.log(1 + beta ** 2) - np.log(sigma)

# Defining log-likelihood function same as the one in  JVDP's blog article
def log_likelihood(theta, x, y):
    alpha, beta, sigma = theta
    y_model = alpha + beta * x
    return -0.5 * np.sum(np.log(2 * np.pi * sigma ** 2) + (y - y_model) ** 2 / sigma ** 2)

# Defining the log of posterior probability
def log_posterior(theta, x, y):
    return log_prior(theta) + log_likelihood(theta, x, y)

np.random.seed(1)

#initializing some random values as initial guess for MCMC algorithm
initial_guess = np.random.random((num_walkers, num_dim))

#Running the MCMC algorithm
sampler = emcee.EnsembleSampler(num_walkers, num_dim, log_posterior, args=(x, y))
```

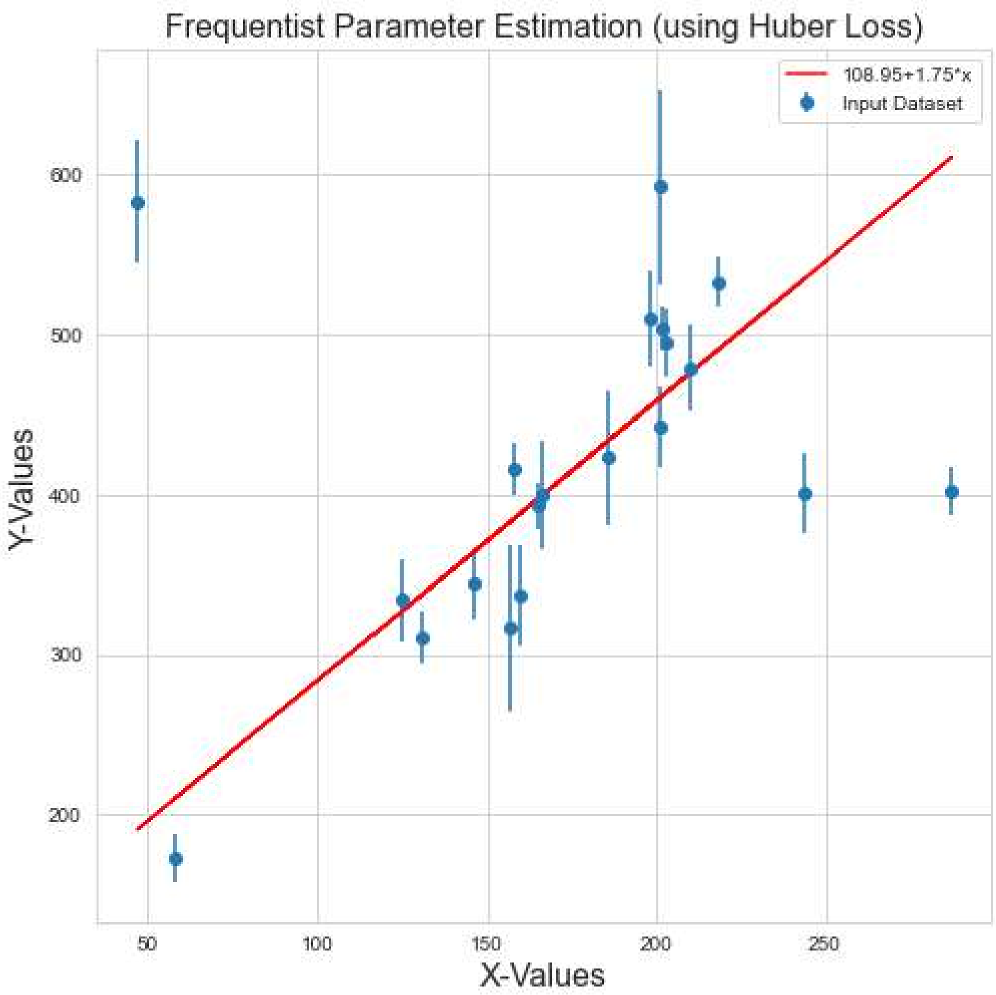
```
sampler.run_mcmc(initial_guess, num_steps)
samples = sampler.get_chain(discard=num_burn_period, thin=15, flat=True)

#Plotting the corner plot
fig = corner.corner(samples, levels=(0.68, 0.95), labels=["a", "b", "sigma_y"])
fig.suptitle("68% and 95% Confidence interval for given data")
plt.show();
```



Q3.

In [15]:

```
#reading data and storing x, y and sigma_y values as arrays
data = pd.read_csv("D:\CLASSES\SEM 4\Data Science Analysis\A6\q2_data.csv")
X = data['x']
Y = data['y']
sigma_y = data['error in y']

#Calculating huber loss and total huber loss
#We use this method as it is better than using squared loss when outliners are to be identified
def huber_loss(t, c = 3):
    return ((abs(t) < c) * 0.5 * t ** 2
            + (abs(t) >= c) * -c * (0.5 * c - abs(t)))

def total_huber_loss(theta, x = X, y = Y, e = sigma_y, c = 3):
    return huber_loss((y - theta[0] - theta[1] * x) / e, c).sum()

#Calculating best fit parameters so that the total huber loss is minimized
best_theta_freq = optimize.fmin(total_huber_loss, [0, 0], disp = False)
[intercept, slope] = best_theta_freq
y_vals = intercept + slope * X

# Plotting the original data along with errorbars and also the best-fit line
fig1 = plt.figure(figsize = (8,8))
sb.set_style('whitegrid')

plt.errorbar(x = X, y = Y, yerr = sigma_y, fmt = "o", label = "Input Dataset")
plt.plot(X, y_vals, 'r',label=f"{intercept:.2f}+{slope:.2f}*x")
plt.legend()

plt.title("Frequentist Parameter Estimation (using Huber Loss)",fontsize = 16)
plt.xlabel("X-Values",fontsize = 16)
plt.ylabel("Y-Values",fontsize = 16)

plt.show()
```



In [22]:

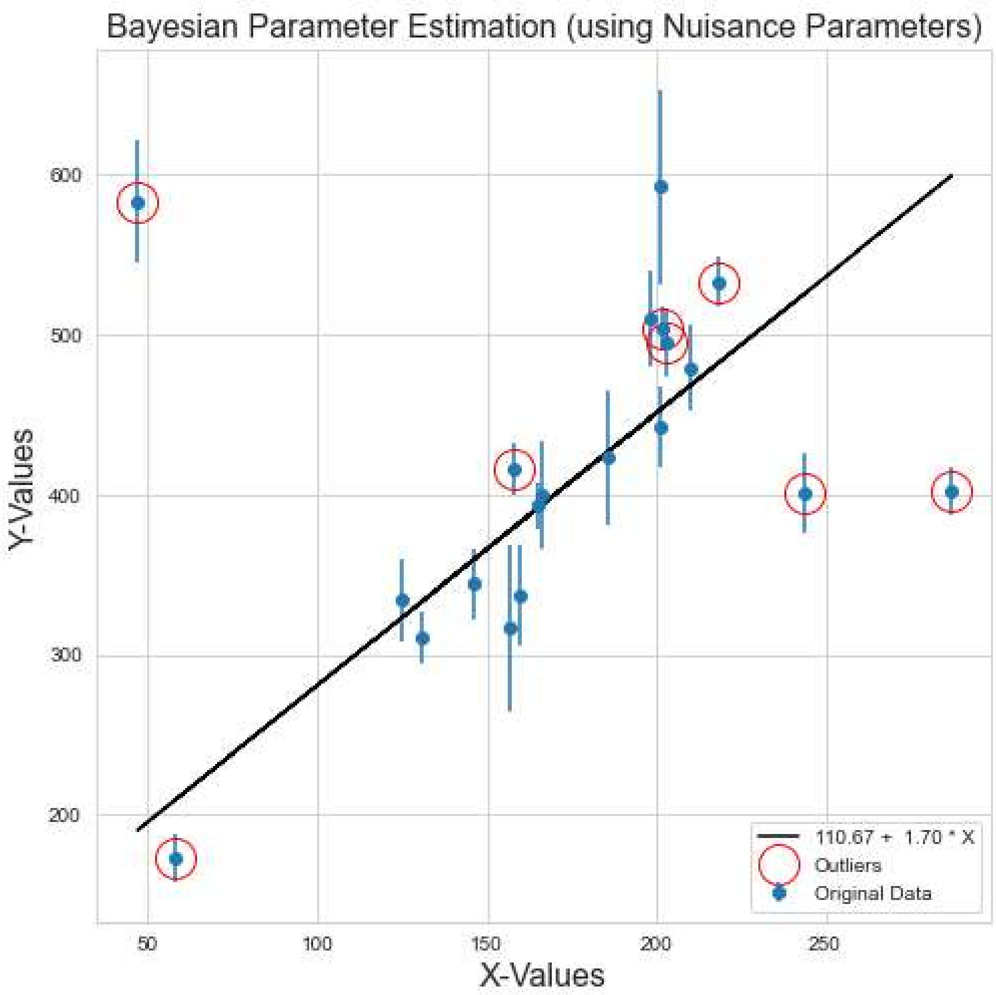
```
##Best-fit Linear Paramters using Bayesian Parametric Estimation##
#Defining log-prior probabilities
def log_prior(theta):
    if (all(theta[2:] > 0) and all(theta[2:] < 1)):
        #g_i needs to be between 0 and 1
        return 0
    else:
        #recall log(0) = -inf
        return -np.inf

#defining log likelihood of data including nuisance parameters
```

```python
def log_likelihood(theta, x, y, e, sigma_B):
    dy = y - theta[0] - theta[1] * x

    #g<0 or g>1 leads to NaNs in logarithm
    g = np.clip(theta[2:], 0, 1)

    logL1 = np.log(g) - 0.5 * np.log(2 * np.pi * e**2) - 0.5 * (dy / e) ** 2
    logL2 = np.log(1 - g) - 0.5 * np.log(2 * np.pi * sigma_B ** 2) - 0.5 * (dy / sigma_B) ** 2
    return np.sum(np.logaddexp(logL1, logL2))

#defining posterior probability
def log_posterior(theta, x, y, e, sigma_B):
    return log_prior(theta) + log_likelihood(theta, x, y, e, sigma_B)

#no.of parameters in the model
num_dim = 2 + len(X)
#burn period for chain stabilization
num_burn_period = 1000
#MCMC steps
num_steps = 10000
#MCMC walkers
num_walkers = 50

np.random.seed(0)

#Setting random Normal varying numbers as initial guess
initial_guess = np.zeros((num_walkers, num_dim))
initial_guess[:, :2] = np.random.normal(best_theta_freq, 1, (num_walkers, 2))
initial_guess[:, 2:] = np.random.normal(0.5, 0.1, (num_walkers, num_dim - 2))

# Running the MCMC algorithm
sampler = emcee.EnsembleSampler(num_walkers, num_dim, log_posterior, args=[X, Y, sigma_y, 50])
sampler.run_mcmc(initial_guess, num_steps)

#shape is (nwalkers, nsteps, ndim)
sample = sampler.chain
sample = sampler.chain[:, num_burn_period:, :].reshape(-1, num_dim)
```

```
<ipython-input-22-bfd0ebe342ea>:19: RuntimeWarning: divide by zero encountered in log
  logL2 = np.log(1 - g) - 0.5 * np.log(2 * np.pi * sigma_B ** 2) - 0.5 * (dy / sigma_B) ** 2
<ipython-input-22-bfd0ebe342ea>:18: RuntimeWarning: divide by zero encountered in log
  logL1 = np.log(g) - 0.5 * np.log(2 * np.pi * e**2) - 0.5 * (dy / e) ** 2
```

In [26]:

```python
#best fit parameters obtained
best_theta_bayes = np.mean(sample[:, :2], 0)

# Taking the mean value of the g1 and g2 nuisance parameters and setting the appropriate value of g to identify outliners
g1 = sample[:, 2].mean()
g2 = sample[:, 3].mean()

g = np.mean(sample[:, 2:], 0)

outliers= (g < (g1+g2)/2)

#Plotting the original data and the best-fit line
figure1 = plt.figure(figsize = (8,8))

plt.errorbar(x = X, y = Y, yerr = sigma_y, fmt='o', label = "Original Data")

plt.plot(X, best_theta_bayes[0] + best_theta_bayes[1] * X, color ='black', label = f"{best_theta_bayes[0]:.2f} + {best_theta_bayes
plt.plot(X[outliers], Y[outliers], 'ro', ms=20, mfc='none', mec='red', label="Outliers")

plt.legend(loc="lower right")

plt.title('Bayesian Parameter Estimation (using Nuisance Parameters)',fontsize = 16)
plt.xlabel("X-Values",fontsize = 16)
plt.ylabel("Y-Values",fontsize = 16)
plt.show();
```



In [25]:

```
#Plotting the original data and the best-fit lines obtained by the Maximum Likelihood estimation and Bayesian estimation
figure2 = plt.figure(figsize=(8,8))

plt.errorbar(x = X, y = Y, yerr = sigma_y, fmt = "o")
plt.plot(X, y_vals, label = f"Frequentist Approach : {intercept:.2f} + {slope:.2f}*x")
plt.plot(X, best_theta_bayes[0]+best_theta_bayes[1]*X, label=f"Bayesian Approach : {best_theta_bayes[0]:.2f} + {best_theta_bayes[1

#plt.legend(loc="lower center")
plt.xlabel("X-Values", fontsize = 16)
plt.ylabel("Y-Values", fontsize = 16)
```

```
plt.title("Frequentist and Bayesian Best-Fit lines",fontsize = 16)
plt.show();
```