Statistical Analysis, Simulation, and Modeling 2023

Semester-end project assignments

Section I: Best-fit model selection

Libraries:

Step-1: Empirical data Analysis

To visually assess potential irregularities within the three sample datasets, such as significant outlier points, histograms were plotted normally and zoomed within their 95% percentile values.

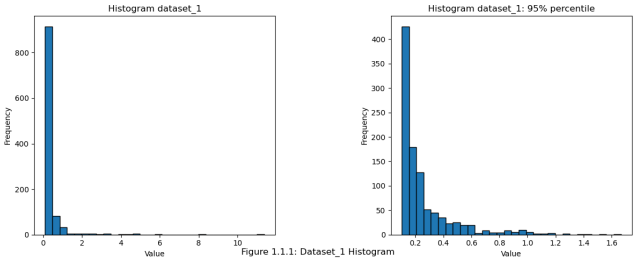
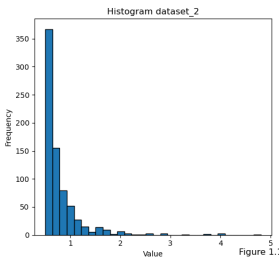


Figure 1.1.1: Dataset_1 Histogram



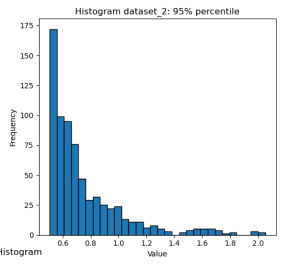
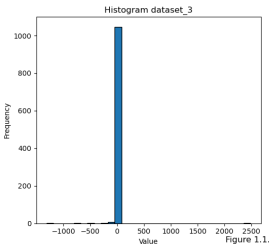


Figure 1.1.2: Dataset_2 Histogram

From **Figure 1.1.1**, we can see that although dataset_1's most values clustered between 0 to 1, a few outliers around 10 exist. However, judging by its height and the extent of its deviation, the outliers are not worth noting, as they are not significant enough to represent the aspects of the sample's distribution.

From **Figure 1.1.2**, we can see that although dataset_2's most values clustered between 0 to 1 and some around 2, a few outliers around 5 exist. However, judging by its height and the extent of its deviation, the outliers are not worth noting, as they are not significant enough to represent the aspects of the sample's distribution.



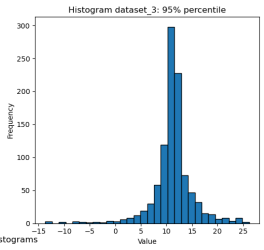
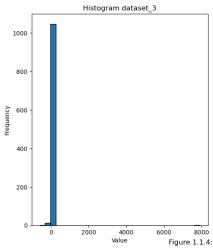


Figure 1.1.3: Dataset_3 Histograms



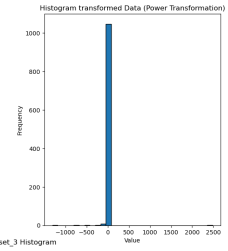


Figure 1.1.4: Transformed dataset_3 Histogram

From **Figure 1.1.3**, we can see two issues with the dataset_3. Firstly, an extreme outlier exists around 8000, which might represent that we are dealing with a heavy-tailed distribution. As the dataset_3 contains such an extreme outlier, it raises the risk of numerical stability within the calculation of CDF. So, as a pre-processing approach for the sample, we can use power transformation to stabilize its variance. One notable power transformer that accepts negative numbers is the Yeo-Johnson transformation from **Scipy.stats** library. For improved stability, the transformation is applied twice to the sample. The resulting histogram, can be seen in **Figure 1.1.4.**

Secondly, as the sample contains negative values, it contradicts the multiple reference distribution's parameter restrictions.

Dataset_3 model rejections:

Half-Normal distribution (Rejected): Reason - requires all positive data points ($x > 0$).

Log-Normal distribution (Rejected): Reason - requires all positive data points ($x > 0$).

Pareto distribution (Rejected): Reason - requires k to be greater than 0 and less than every data points in the sample. As the minimum of the sample is less than 0, it creates confliction within its requirements.

Gamma distribution (Rejected): Reason - requires all positive data points ($x > 0$).

Dataset_3 Remaining models: Levy, Exponential, Gumbel, Cauchy distribution

Step-2: MLE calculation:

As a necessity for both qualitative and quantitative analysis, MLE parameters for the reference distributions require calculation. While MLE calculation utilizes straightforward negative log-likelihood minimization as its standard approach, as we have the boundary restrictions for the distribution parameters, simulated annealing is favorable for our specific case. Additionally, minimization risks getting trapped in local minima depending on its initial value, and alternatively, simulated annealing finds the global minima within its given boundary. The simulated annealing function, dual_annealing from the Scipy.stats library is used for MLE calculation for all remaining distributions for all samples using the simulated annealing function, **dual_annealing**, from the **Scipy.stats** library. However, not to misunderstand, dual_annealing still finds the MLE by minimizing the negative log likelihood function.

Parameter Boundaries for the reference distributions:

bounds_half_normal = [(epsilon, 1000)]

bounds_lognormal = [(-1000, 1000), (epsilon, 1000)]

bounds_levy = [(-1000, min(dataset) - epsilon), (epsilon, 10000)]

bounds_exponential = [(epsilon, 10000)]

bounds_pareto = [(epsilon, min(dataset_3)), (epsilon, 1000)]

bounds_gamma = [(epsilon, 1000), (epsilon, 1000)]

bounds_gumbel = [(-1000, 1000), (epsilon, 1000)]

bounds_cauchy = [(-1000, 1000), (epsilon, 1000)]

To avoid zero division error, a small offset value, epsilon (1e-10), is used instead of zero, and 1000 is used instead of infinity to avoid simulated annealing to explore irrelevant areas, unless bigger one is required out of necessity.

Estimated Parameters:

Distributions	Dataset_1	Dataset_2	Dataset_3
Half-Normal: θ	1.67530137	1.34965799	Rejected
Log-Normal: μ	-1.53094778	-0.31187334	Rejected
Log-Normal: σ	0.74368751	0.37717188	Rejected
Lévy: μ	0.09620193	0.49298647	-1311.34712466
Lévy: σ	0.03372367	0.07236848	1317.72619658
Exponential: λ	2.95446144	1.2468024	826.15545084
Pareto: α	1.30144637	2.63467525	Rejected
Pareto: k	0.10032668	0.50086095	Rejected
Gamma: α	1.25751257	5.63837458	Rejected
Gamma: β	0.26915927	0.14224875	Rejected
Gumbel: α	0.67766999	1.10213206	15.23242735
Gumbel: β	1.11487943	0.93070378	16.59401446
Cauchy: a	0.15255642	0.62091525	11.31372072
Cauchy: b	0.05152416	0.09857946	1.41829099

Table 1.2.1: MLE distribution parameters

Step-3: Qualitative Analysis:

Probability Distribution Functions:

The PDF plots are created by inputting the MLE parameters and evenly spaced values within the dataset's range with the same length of data points using **np.linspace**.

From **Figure 1.3.1**, we can see the dataset_1 histogram plotted with the distribution model PDF plots.

Dataset_1 model rejections: Half-Normal, Gumbel distribution

Dataset_1 remaining models: Log-Normal, Levy, Exponential, Pareto, Gamma, Cauchy distribution

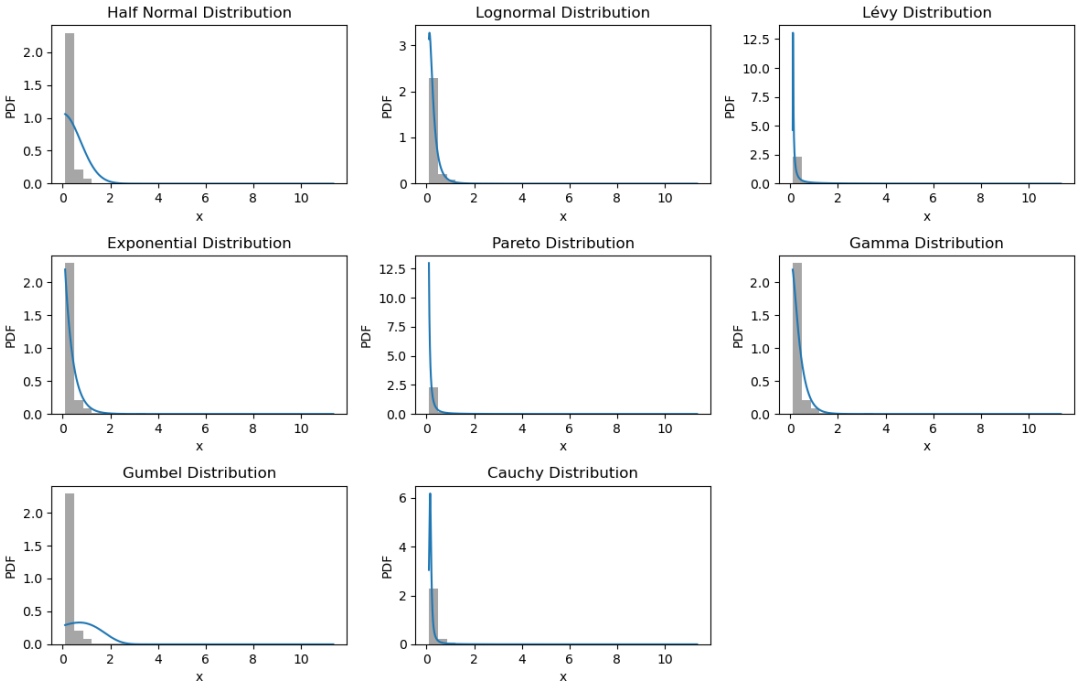


Figure 1.3.1: Dataset_1 distribution PDFs

From **Figure 1.3.2**, we can see the dataset_2 histogram plotted with the distribution model PDF plots.

Dataset_2 model rejections: Half-Normal, Exponential, Gumbel distribution

Dataset_2 remaining models: Log-Normal, Levy, Pareto, Gumbel, Cauchy distribution

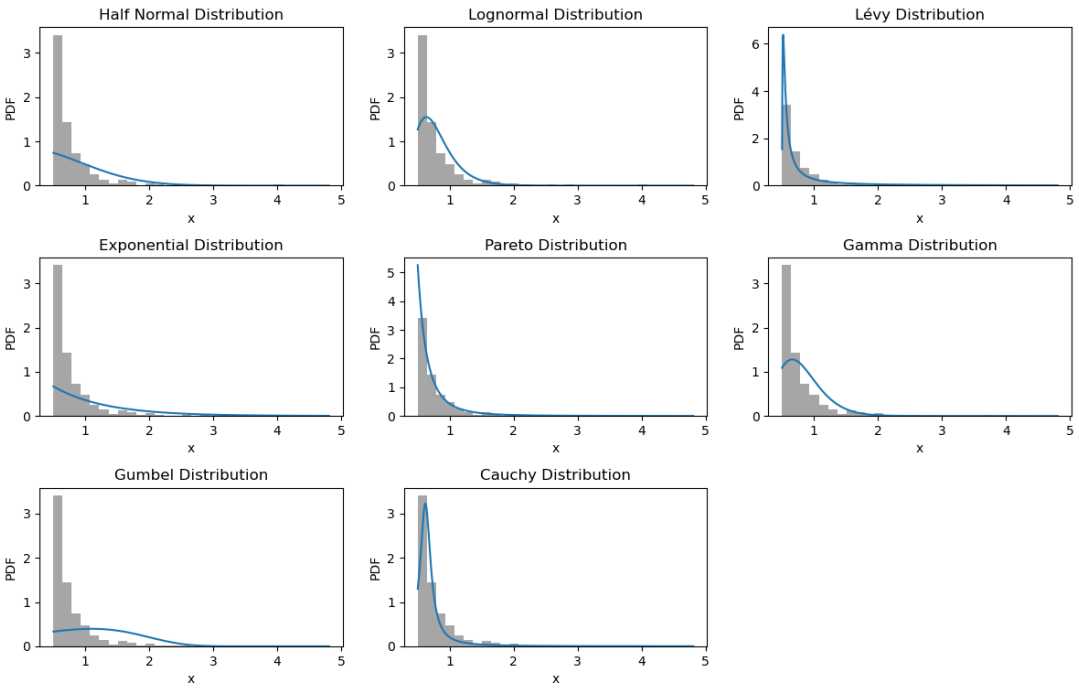


Figure 1.3.2: Dataset_2 distribution PDFs

From **Figure 1.3.3**, we can see the dataset_3 histogram plotted with the distribution model PDF plots.

Dataset_3 model rejections: Exponential distribution

Dataset_3 remaining models: Gumbel, Cauchy distribution

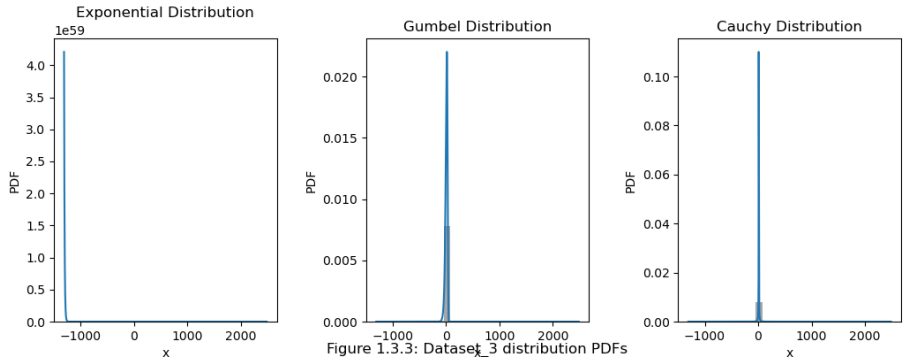


Figure 1.3.3: Dataset_3 distribution PDFs

The rejections made using the PDF distributions were purely visual.

Q-Q plots:

The **statsmodels** library **sm.qqplot_2samples** function was utilized combined with MLE parameters and theoretical quantiles evenly spaced within the datasets' range using **np.linspace**. The resulting 2x3 grid of QQ plots illustrates the comparison of theoretical quantiles against those of datasets, with a reference line at 45 degrees for clarity. A deviation from this line in the plots indicates potential mismatches between the distributions and datasets, guiding the rejection of less suitable fits.

From **Figure 1.3.4**, we can see the dataset_1 quantiles plotted against theoretical quantiles created using the distribution model PDF formulas combined with **np.linspace**.

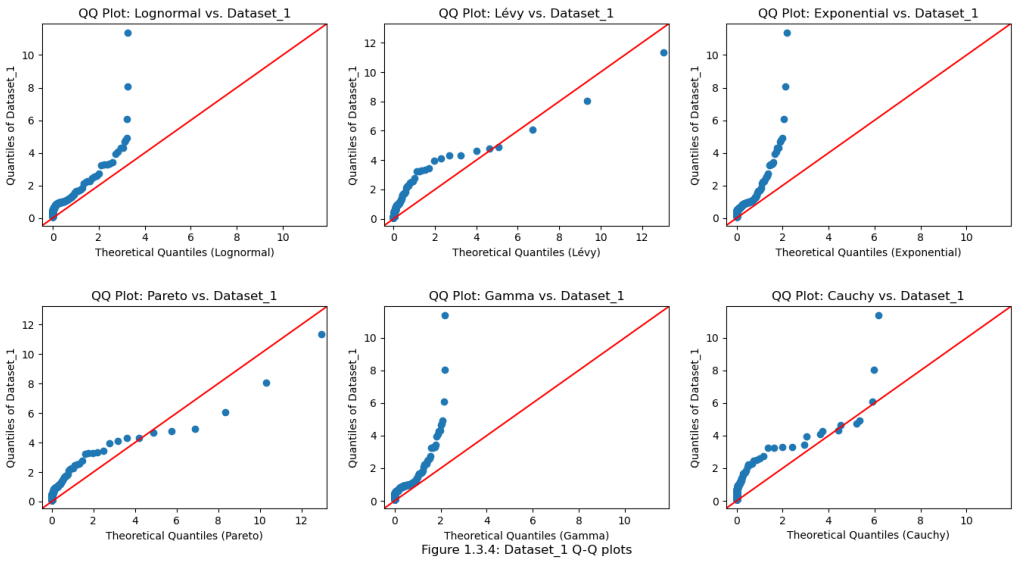


Figure 1.3.4: Dataset_1 Q-Q plots

Dataset_1 model rejections: Exponential, Gamma distribution

Dataset_1 remaining models: Log-Normal, Levy, Pareto, Cauchy distribution

From **Figure 1.3.5**, we can see the dataset_2 quantiles plotted against theoretical quantiles created using the distribution model PDF formulas combined with **np.linspace**.

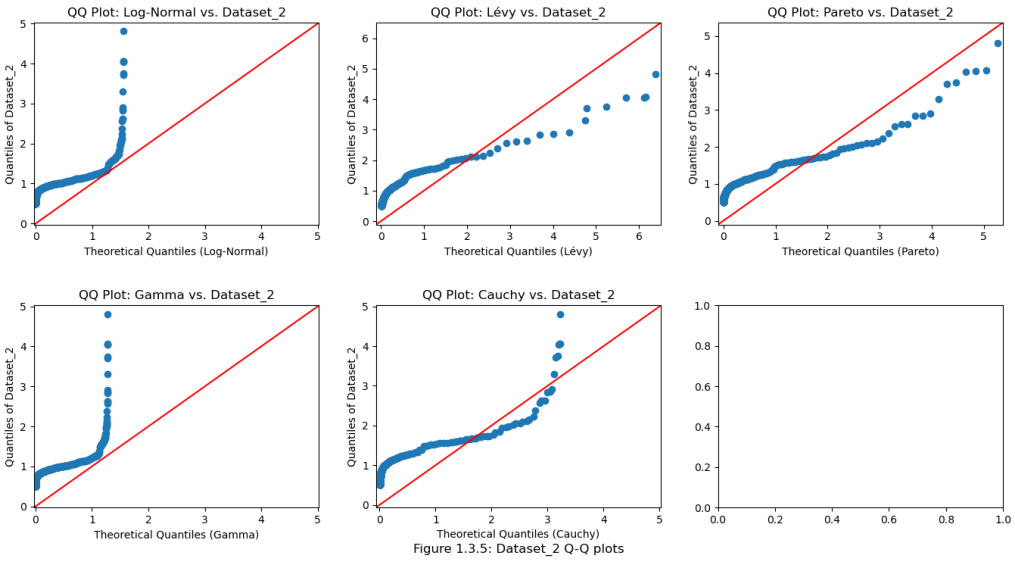


Figure 1.3.5: Dataset_2 Q-Q plots

Dataset_2 model rejections: Log-Normal, Gamma distribution

Dataset_2 remaining models: Levy, Pareto, Cauchy distribution

From **Figure 1.3.6**, we can see the dataset_3 quantiles plotted against theoretical quantiles created using the distribution model PDF formulas combined with **np.linspace**.

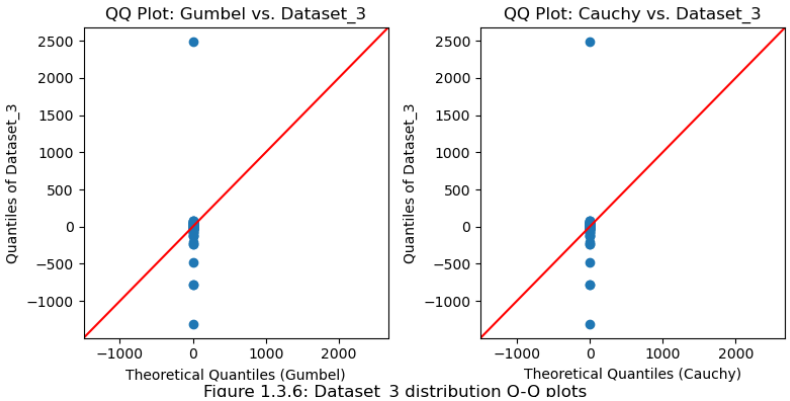


Figure 1.3.6: Dataset_3 distribution Q-Q plots

Dataset_3 model rejections: None

Dataset_3 remaining models: Gumbel, Cauchy distribution

Step-4: Quantitative analysis:

Specification test (Goodness of Fit):

For my specification test, although Anderson Darling is ideal for testing heavy-tailed distributions, due to the lack of library options, the alternative Kolmogorov Smirnov test. The test uses the **kstest** function from **Scipy.stats** library, which uses empirical data and CDF as input. For the CDF formula, as CDF is essentially the integral of PDF, using the reference distributions and quad function from **scipy.integrate** library successfully derived CDF and used the kstest as the model rejection method.

Distributions	Dataset_1	Dataset_2	Dataset_3
Critical Value	**0.0416548525659681**	**0.049623949371356**	**0.0416548525659681**
Half-Normal:	Rejected	Rejected	Rejected
Log-Normal:	0.150755989231507	0.15713223042886	Rejected
Lévy:	0.118046905754623	Rejected	Rejected
Exponential:	Rejected	Rejected	Rejected
Pareto:	0.0197849645404437	0.0199864398043846	Rejected
Gamma:	Rejected	Rejected	Rejected
Gumbel:	Rejected	Rejected	1.0
Cauchy:	0.24783500252594	0.218834661180852	0.0148168773974032

Table 1.4.1: KS-statistics and critical values

Dataset_1 model rejections: Log-Normal, Levy, Cauchy distribution

Dataset_1 remaining models: Pareto distribution

Dataset_2 model rejections: Log-Normal, Cauchy distribution

Dataset_2 remaining models: Pareto distribution

Dataset_3 model rejections: Gumbel distribution

Dataset_3 remaining models: Cauchy distribution

KS-test based model rejection all depends on the ks-statistic for the distribution and the critical value of the dataset. The distribution model is rejected when ks-statistic is greater than critical value, as the ks-statistic should converge to 0 if the empirical data is from the distribution.

Model selection:

Following the specification test, when multiple models are still under consideration (have not been rejected yet), AIC and BIC serve as criteria to identify the best-fitting model among them. AIC

(Akaike Information Criterion) and BIC (Bayesian Information Criterion) evaluate models by considering their complexity, using the number of model parameters as the punishment mechanism. When AIC and BIC scores are calculated lower, they represent better-fitting models. However, in some cases, even after AIC and BIC assessments, it is possible to be left with multiple models. If the difference from the minimum AIC model is less than 2, evidence is not enough for model rejection. If the difference is within 4, there is considerable evidence against the model, but the model can still utilized. Although in my datasets cases, there is no need for AIC and BIC calculation, I want to demonstrate the results.

Dataset_1 Pareto criterion scores: AIC = -1684.9452844782181, BIC = -1675.0075837215343

Dataset_2 Pareto criterion scores: AIC = -416.38862839751573, BIC = -407.15115043048127

Dataset_3 Cauchy criterion scores: AIC = 6188.460920906726, BIC = 6198.398621663409

Step-5: Model parameter confidence interval:

In this class, we learned about three types of confidence intervals, namely percentile confidence interval, BC (bias-corrected) confidence interval, and BCa (bias-corrected accelerated) confidence interval. Each of these confidence intervals has its strengths and drawbacks. Although the percentile confidence interval is easy to calculate, it is not as accurate in asymmetric data. As for BC, it appropriately deals with skewed data but is weak against datasets with huge outliers. Despite being the best-representing confidence interval, the BCa method is computationally very demanding to calculate.

From **Figures 1.5.1, 1.5.2, and 1.5.3,** you can notice that BC and BCa lines overlap, resulting in visual indistinguishability. This similarity between BC and BCa intervals indicates that, in this particular case, the bootstrap method provides a reliable estimate of the confidence interval, and the additional bias correction and acceleration do not introduce significant changes. The unreliability of the percentile confidence intervals can be seen in the Pareto parameter k calculation. The percentile confidence tends to generalize the interval towards the skewed side.

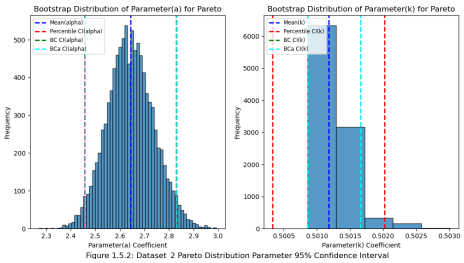


Figure 1.5.2: Dataset 2 Pareto Distribution Parameter 95% Confidence Interval

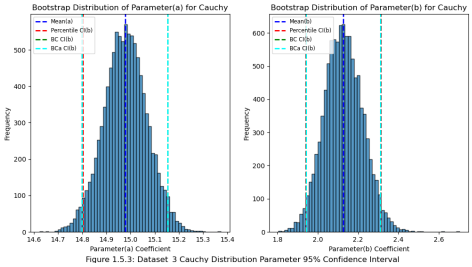


Figure 1.5.3: Dataset 3 Cauchy Distribution Parameter 95% Confidence Interval

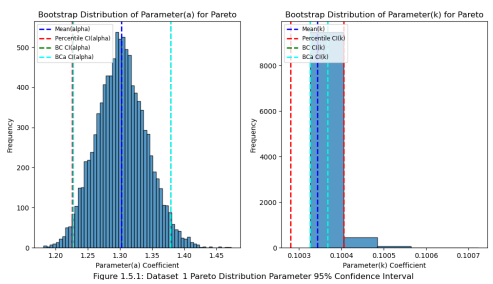


Figure 1.5.1: Dataset 1 Pareto Distribution Parameter 95% Confidence Interval

Section II: Glivenko Cantelli theorem

The Glivenko-Cantelli theorem essentially describes the convergence behavior of the ECDF to the true CDF. The Kolmogorov-Smirnov statistic serves as a good example of showing this behavior. As KS statistic is defined as the maximum absolute difference between the ECDF and CDF. Once the KS statistic converges to 0, it would imply that the ECDF has converged to the true CDF.

For this section to illustrate this behavior, I have chosen two different heavy-tailed distributions namely, Log-Normal and Cauchy distribution. The reason for choosing heavy-tailed distribution is that it has a high chance of producing extreme outliers which would be a great candidate to display how the Glivenko Cantelli convergence behaves.

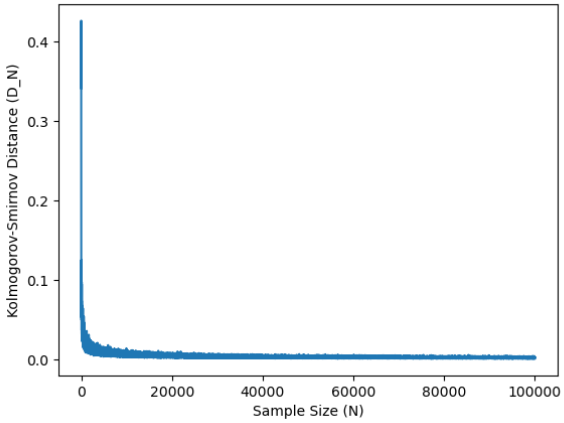


Figure 2.1: Convergence of D_N Empirical Lognorm vs Lognorm

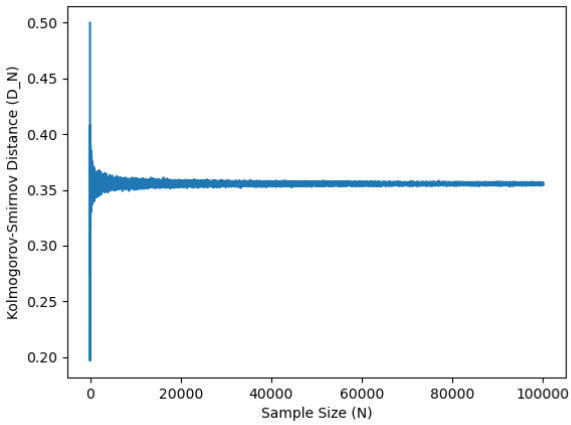


Figure 2.2: Convergence of D_N Empirical Lognorm vs Cauchy

As previously mentioned, the main goal of the Glivenko-Cantelli theorem is to achieve convergence of the empirical cumulative distribution function (ECDF) to its true cumulative distribution function (CDF). This experiment began by generating 10,000 random variables from a Log-Normal distribution. With these Log-Normal random variables as our empirical data, we computed their ECDF.

As we perform bootstrap resampling with increasing sample sizes, the random variables progressively display the behavior of their distributions, eventually converging toward 0. In Figure 2.2, as the Cauchy distribution is not its true distribution, it does not converge to 0. Instead, it converges to 0.35. This value signifies that it does not accurately represent the true CDF of the empirical data.

Section III: PBCM method

The Parametric Bootstrap Cross-Fitting Method (PBCM) is a method in which designed to assess and compare conventionally unrelated models. This method involves clustering and evaluating the goodness of fit, in my case, ks-statistic of various distribution methods. By combining the techniques of parametric bootstrap resampling and cross-fitting, PBCM provides a way to assess commonality or equivalence between models, especially in scenarios where traditional assumptions may not hold.

For calculating the GOF difference with the PBMC method, we tackled with 5 step approach.

1. Get a nonparametric bootstrap from my observed data, which I have chosen as my dataset_2.
2. With all the resample data created using dataset_2, fit each bootstrap dataset into my distribution models, namely model A(Log-Normal) and model B(Pareto), creating MLE parameters for them.
3. Then using the fitted parameters, I feed it into a random variable generator for the distributions and create a parametric bootstrap sample for each model.
4. Depending on which distribution model random variable generator we used to create the parametric bootstrap sample, we deem it as the "true" fit model. The deemed "true" model dataset calculates GOF for both models and stores their differences. We then do it with the assumption of the other model being true.
5. Finally, these steps are repeated 1000 times to create the histogram for the "true" model GOF difference distribution.

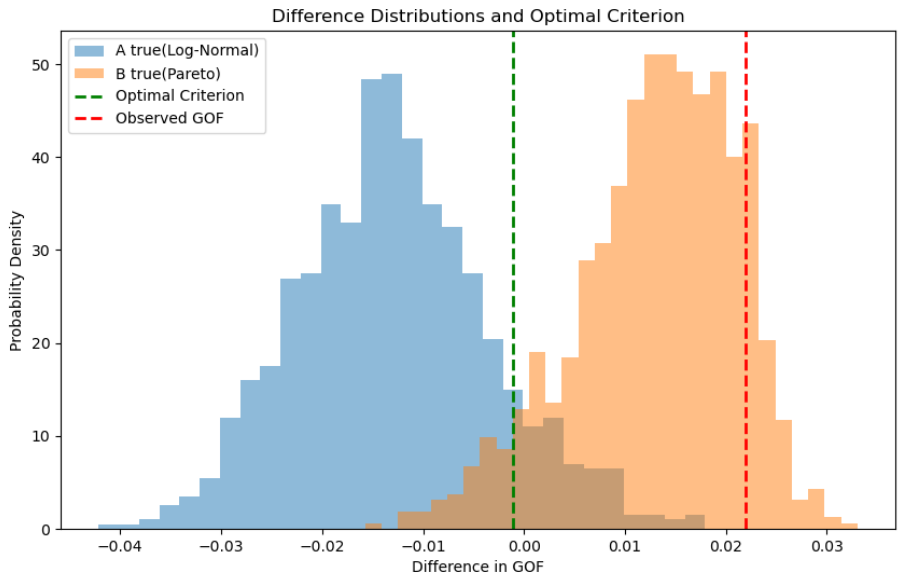


Figure 3.1: GOF difference distribution

After the steps, one would be left with results that resembles the **Figure 3.1.** Foremost, we can see that the initial GOF, we got from the observed dataset was more trended towards the model B. Although both models are quite similar, intersecting in values, the optimal criterion which would fit both models are quite off from it.

Appendix:
Section 1: Dataset_1 model Selection

In [33]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
import scipy
import csv
import pandas as pd
import seaborn as sns
import random
import statsmodels.api as sm
from scipy.integrate import quad
from scipy.special import gamma as gamma_function
from scipy.optimize import dual_annealing
from scipy.stats import kstest, yeojohnson
```

In [39]:
```
folder_path = '/Users/biligee/Documents/Classwork/semester_4/Stats/Final_

dataset_1 = np.genfromtxt('data1.csv', delimiter=',')
```

In [67]:
```
# Create a figure with 1 row and 2 columns of subplots
plt.figure(figsize=(12, 5))

# Subplot 1
plt.subplot(1, 2, 1)
plt.hist(dataset_1, bins=30, edgecolor='black')
plt.title('Histogram dataset_1')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Subplot 2
plt.subplot(1, 2, 2)
plt.hist(dataset_1, bins=30, range=(np.percentile(dataset_1, 2.5), np.per
plt.title('Histogram dataset_1: 95% percentile')
plt.xlabel('Value')
plt.ylabel('Frequency')

plt.text(0.5, 0.05, 'Figure 1.1.1: Dataset_1 Histogram', ha='center', va=

# Adjust layout to prevent overlapping
plt.tight_layout()


# Save the figure
filename = os.path.join(folder_path, 'figure_1.1.1.png')
plt.savefig(filename)
# Display the plot
plt.show()
```

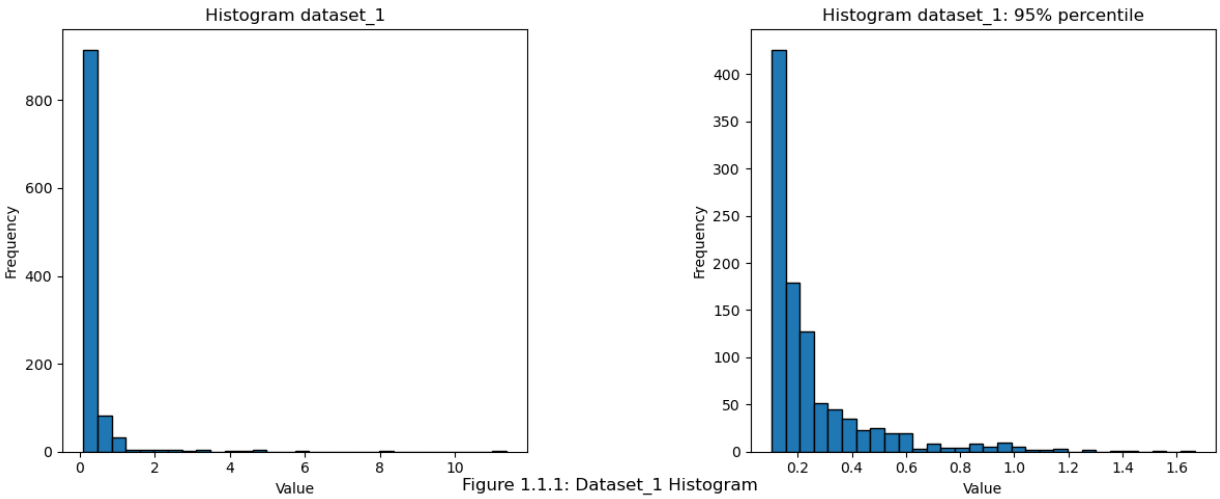


Figure 1.1.1: Dataset_1 Histogram

HalfNormal Distribution: $\frac{2e^{-\frac{x^2\theta^2}{\pi}}\theta}{\pi}$

Lognormal Distribution: $\frac{e^{-\frac{(-\mu+\ln x)^2}{2\sigma^2}}}{\sqrt{2\pi}\cdot x\cdot\sigma}$

Lévy Distribution: $\frac{e^{-\frac{\sigma}{2(x-\mu)}}\left(\frac{\sigma}{x-\mu}\right)^{\frac{3}{2}}}{\sqrt{2\pi}\sigma}$

Exponential Distribution: $e^{-x\lambda}\lambda$

Pareto Distribution: $k^{\alpha}\cdot x^{-1-\alpha}\cdot\alpha$

Gamma Distribution: $\frac{e^{-\frac{x}{\beta}}x^{-1+\alpha}\beta^{-\alpha}}{\Gamma(\alpha)}$

Gumbel Distribution: $\frac{e^{-e^{\frac{x-\alpha}{\beta}}+\frac{x-\alpha}{\beta}}}{\beta}$

Cauchy Distribution: $\frac{1}{b\pi\left(1+\frac{(-a+x)^2}{b^2}\right)}$

In [4]:
```
def half_normal(x, theta):
    if not (np.all(x > 0) and np.all(theta > 0)):
        raise ValueError("Both x and theta must be greater than 0")
    return (2 * np.exp(-x**2 * theta**2 / np.pi) * theta) / np.pi

def lognormal(x, mu, sigma):
    if not (np.all(x > 0) and np.all(sigma > 0)):
        raise ValueError("Both x and sigma must be greater than 0")
    return (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))) / (np.sqrt(2 *

def levy(x, mu, sigma):
    if not (np.all(x > mu) and np.all(sigma > 0)):
        raise ValueError("x must be greater than mu, and sigma must be gr
    return (np.exp(-sigma / (2 * (x - mu))) * (sigma / (x - mu))**(3/2))

def exponential(x, lam):
    if not np.all(lam > 0):
        raise ValueError("Lambda must be greater than 0")
    return np.exp(-x * lam) * lam

def pareto(x, k, alpha):
    if not (np.all(x >= k) and np.all(k > 0) and np.all(alpha > 0)):
        raise ValueError("x must be greater than or equal to k, and both
    return k**alpha * x**(-1 - alpha) * alpha

def gamma_distribution(x, alpha, beta):
    if not (np.all(x > 0) and np.all(alpha > 0) and np.all(beta > 0)):
        raise ValueError("x, alpha, and beta must all be greater than 0")
    return (np.exp(-x / beta) * x**(alpha - 1) * beta**(-alpha)) / gamma_

def gumbel(x, alpha, beta):
    if not np.all(beta > 0):
        raise ValueError("Beta must be greater than 0")
    return np.exp(-np.exp((x - alpha) / beta) + (x - alpha) / beta) / bet


def cauchy_distribution(x, a, b):
    if not np.all(b > 0):
        raise ValueError("Scale parameter (b) must be greater than 0")
    return 1 / (b * np.pi * (1 + ((-a + x)**2 / b**2)))
```

In [5]:
```
epsilon = 1e-10

def nnlf_half_normal(params, data):
    theta = params[0]
    if not (np.all(data > 0) and np.all(theta > 0)):
        return np.inf
    half_normal_values = half_normal(data, theta)
    valid_values = np.isfinite(half_normal_values)
    log_likelihood = -np.sum(np.log(np.where(half_normal_values[valid_val
    return log_likelihood

def nnlf_lognormal(params, data):
    mu, sigma = params
    if not (np.all(data > 0) and np.all(sigma > 0)):
        return np.inf
```

```
    lognormal_values = lognormal(data, mu, sigma)
    valid_values = np.isfinite(lognormal_values)
    log_likelihood = -np.sum(np.log(np.where(lognormal_values[valid_value
    return log_likelihood

def nnlf_levy(params, data):
    mu, sigma = params
    if not (np.all(data > mu) and np.all(sigma > 0)):
        return np.inf
    levy_values = levy(data, mu, sigma)
    valid_values = np.isfinite(levy_values)
    log_likelihood = -np.sum(np.log(np.where(levy_values[valid_values] ==
    return log_likelihood

def nnlf_exponential(params, data):
    lam = params[0]
    if not (np.all(lam > 0)):
        return np.inf
    exponential_values = exponential(data, lam)
    valid_values = np.isfinite(exponential_values)
    log_likelihood = -np.sum(np.log(np.where(exponential_values[valid_val
    return log_likelihood

def nnlf_pareto(params, data):
    k, alpha = params
    if not (np.all(data >= k) and np.all(k > 0) and np.all(alpha > 0)):
        return np.inf
    pareto_values = pareto(data, k, alpha)
    valid_values = np.isfinite(pareto_values)
    log_likelihood = -np.sum(np.log(np.where(pareto_values[valid_values]
    return log_likelihood

def nnlf_gamma(params, data):
    alpha, beta = params
    if not (np.all(data > 0) and np.all(alpha > 0) and np.all(beta > 0)):
        return np.inf
    gamma_values = gamma_distribution(data, alpha, beta)
    valid_values = np.isfinite(gamma_values)
    log_likelihood = -np.sum(np.log(np.where(gamma_values[valid_values] =
    return log_likelihood

def nnlf_gumbel(params, data):
    alpha, beta = params
    if not np.all(beta > 0):
        return np.inf
    gumbel_values = gumbel(data, alpha, beta)
    valid_values = np.isfinite(gumbel_values)
    log_likelihood = -np.sum(np.log(np.where(gumbel_values[valid_values]

    return log_likelihood

def nnlf_cauchy(params, data):
    a, b = params
    if not np.all(b > 0):
        return np.inf
    cauchy_values = cauchy_distribution(data, a, b)
    valid_values = np.isfinite(cauchy_values)
```

```
    log_likelihood = -np.sum(np.log(np.where(cauchy_values[valid_values]
    return log_likelihood
```

In [46]:

```
def half_normal_cdf(x, theta):
    if not (np.all(x > 0) and np.all(theta > 0)):
        raise ValueError("Both x and theta must be greater than 0")
    integrand = lambda t: half_normal(t, theta)
    result, _ = quad(integrand, 0, x)
    return result

def lognormal_cdf(x, mu, sigma):
    if not (np.all(x > 0) and np.all(sigma > 0)):
        raise ValueError("Both x and sigma must be greater than 0")
    integrand = lambda t: lognormal(t, mu, sigma)
    result, _ = quad(integrand, 0, x)
    return result

def levy_cdf(x, mu, sigma):
    if not (np.all(x > mu) and np.all(sigma > 0)):
        raise ValueError("x must be greater than mu, and sigma must be gr
    integrand = lambda t: levy(t, mu, sigma)
    result, _ = quad(integrand, mu, x)
    return result

def exponential_cdf(x, lam):
    if not np.all(lam > 0):
        raise ValueError("Lambda must be greater than 0")
    integrand = lambda t: exponential(t, lam)
    result, _ = quad(integrand, 0, x)
    return result

def pareto_cdf(x, k, alpha):
    if not (np.all(x >= k) and np.all(k > 0) and np.all(alpha > 0)):
        raise ValueError("x must be greater than or equal to k, and both
    integrand = lambda t: pareto(t, k, alpha)
    result, _ = quad(integrand, k+epsilon, x)
    return result

def gamma_cdf(x, alpha, beta):
    if not (np.all(x > 0) and np.all(alpha > 0) and np.all(beta > 0)):
        raise ValueError("x, alpha, and beta must all be greater than 0")
    integrand = lambda t: gamma_distribution(t, alpha, beta)
    result, _ = quad(integrand, 0, x)
    return result

def gumbel_cdf(x, alpha, beta):
    if not np.all(beta > 0):
        raise ValueError("Beta must be greater than 0")
    integrand = lambda t: gumbel(t, alpha, beta)
    result, _ = quad(integrand, -np.inf, x)
    return result

def cauchy_cdf(x, a, b):
    if not np.all(b > 0):
        raise ValueError("Scale parameter (b) must be greater than 0")
    integrand = lambda t: cauchy_distribution(t, a, b)
```

```
    result, _ = quad(integrand, -np.inf, x)
    return result

half_normal_cdf = np.vectorize(half_normal_cdf)
lognormal_cdf = np.vectorize(lognormal_cdf)
levy_cdf = np.vectorize(levy_cdf)
exponential_cdf = np.vectorize(exponential_cdf)
pareto_cdf = np.vectorize(pareto_cdf)
gamma_cdf = np.vectorize(gamma_cdf)
gumbel_cdf = np.vectorize(gumbel_cdf)
cauchy_cdf = np.vectorize(cauchy_cdf)
```

In [47]:
```
def jackknife_mean(dataset):
    return sum(dataset)/len(dataset)

def jackknife(dataset, statistic_function):
    n = len(dataset)
    jackknife_estimates = []

    for i in range(n):
        leave_one_out_data = np.delete(dataset, i)
        statistic_value = statistic_function(leave_one_out_data)
        jackknife_estimates.append(statistic_value)


    return jackknife_estimates
```

In [48]:
```
def reject_or_not(ks_statistic, critical):
    if ks_statistic > critical:
        return "Reject the null hypothesis"
    else:
        return "Fail to reject the null hypothesis"
```

In [49]:
```
# Half-Normal Distribution
bounds_half_normal = [(epsilon, 100)]  # Replace np.inf with a large valu

# Lognormal Distribution
bounds_lognormal = [(-100, 100), (epsilon, 100)]  # Replace np.inf with a

# Lévy Distribution
bounds_levy = [(-100, min(dataset_1) + epsilon), (epsilon, 100)]  # Repla

# Exponential Distribution
bounds_exponential = [(epsilon, 100)]  # Replace np.inf with a large valu

# Pareto Distribution
bounds_pareto = [(epsilon, min(dataset_1)), (epsilon, 100)]  # Replace No

# Gamma Distribution
bounds_gamma = [(epsilon, 10), (epsilon, 10)]  # Replace None and np.inf

# Gumbel Distribution
bounds_gumbel = [(-100, 100), (epsilon, 100)]  # Replace np.inf with a la

# Cauchy Distribution
bounds_cauchy = [(-100, 100), (epsilon, 100)]  # Replace np.inf with a la


# Define bounds for other distributions in a similar manner

result_half_normal = dual_annealing(nnlf_half_normal, bounds=bounds_half_
result_lognormal = dual_annealing(nnlf_lognormal, bounds=bounds_lognormal
result_levy = dual_annealing(nnlf_levy, bounds=bounds_levy, args=(dataset
result_exponential = dual_annealing(nnlf_exponential, bounds=bounds_expon
result_pareto = dual_annealing(nnlf_pareto, bounds=bounds_pareto, args=(d
result_gamma = dual_annealing(nnlf_gamma, bounds=bounds_gamma, args=(data
result_gumbel = dual_annealing(nnlf_gumbel, bounds=bounds_gumbel, args=(d
result_cauchy = dual_annealing(nnlf_cauchy, bounds=bounds_cauchy, args=(d

# Access the optimized parameters
optimized_params_half_normal = result_half_normal.x
optimized_params_lognormal = result_lognormal.x
optimized_params_levy = result_levy.x
optimized_params_exponential = result_exponential.x
optimized_params_pareto = result_pareto.x
optimized_params_gamma = result_gamma.x
optimized_params_gumbel = result_gumbel.x
optimized_params_cauchy = result_cauchy.x

# Print or use the optimized parameters as needed
print("Optimized Parameters - Half-Normal:", optimized_params_half_normal
print("Optimized Parameters - Lognormal:", optimized_params_lognormal)
print("Optimized Parameters - Lévy:", optimized_params_levy)
print("Optimized Parameters - Exponential:", optimized_params_exponential
print("Optimized Parameters - Pareto:", optimized_params_pareto)
print("Optimized Parameters - Gamma:", optimized_params_gamma)
print("Optimized Parameters - Gumbel:", optimized_params_gumbel)
print("Optimized Parameters - Cauchy:", optimized_params_cauchy)
```

```
/var/folders/rm/7h6vlsd94v9b6hq6m0qjz4680000gn/T/ipykernel_54288/64895940
8.py:34: RuntimeWarning: overflow encountered in exp
  return np.exp(-np.exp((x - alpha) / beta) + (x - alpha) / beta) / beta
Optimized Parameters - Half-Normal: [1.67530137]
Optimized Parameters - Lognormal: [-1.53094778  0.74368751]
Optimized Parameters - Lévy: [0.09620193 0.03372367]
Optimized Parameters - Exponential: [2.95446144]
Optimized Parameters - Pareto: [0.10032668 1.30144637]
Optimized Parameters - Gamma: [1.25751257 0.26915927]
Optimized Parameters - Gumbel: [0.67766999 1.11487943]
Optimized Parameters - Cauchy: [0.15255642 0.05152416]
```

In [10]:
```
# Example usage:
x_values = np.linspace(min(dataset_1), max(dataset_1), len(dataset_1))

# Specify parameters for each distribution with limitations


# Calculate values for each distribution
half_normal_values = half_normal(x_values, optimized_params_half_normal)
lognormal_values = lognormal(x_values, optimized_params_lognormal[0],opti
levy_values = levy(x_values, optimized_params_levy[0], optimized_params_l
exponential_values = exponential(x_values, optimized_params_exponential)
pareto_values = pareto(x_values, optimized_params_pareto[0], optimized_pa
gamma_values = gamma_distribution(x_values, optimized_params_gamma[0], op
gumbel_values = gumbel(x_values, optimized_params_gumbel[0], optimized_pa
cauchy_values = cauchy_distribution(x_values, optimized_params_cauchy[0],
```

In [56]:
```
# Plot the PDFs
plt.figure(figsize=(12, 8))

plt.subplot(3, 3, 1)
plt.plot(x_values, half_normal_values, label='Half Normal')
plt.hist(dataset_1, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Half Normal Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 2)
plt.plot(x_values, lognormal_values, label='Lognormal')
plt.hist(dataset_1, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Lognormal Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 3)
plt.plot(x_values, levy_values, label='Lévy')
plt.hist(dataset_1, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Lévy Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 4)
plt.plot(x_values, exponential_values, label='Exponential')
plt.hist(dataset_1, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Exponential Distribution')
```

```
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 5)
plt.plot(x_values, pareto_values, label='Pareto')
plt.hist(dataset_1, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Pareto Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 6)
plt.plot(x_values, gamma_values, label='Gamma')
plt.hist(dataset_1, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Gamma Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 7)
plt.plot(x_values, gumbel_values, label='Gumbel')
plt.hist(dataset_1, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Gumbel Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 8)
plt.plot(x_values, cauchy_values, label='Cauchy')
plt.hist(dataset_1, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Cauchy Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.text(0.5, 0.03, 'Figure 1.3.1: Dataset_1 distribution PDFs', ha='cent

# Adjust layout to prevent overlapping
plt.tight_layout()


# Save the figure
filename = os.path.join(folder_path, 'figure_1.3.1.png')
plt.savefig(filename)

plt.show()
```

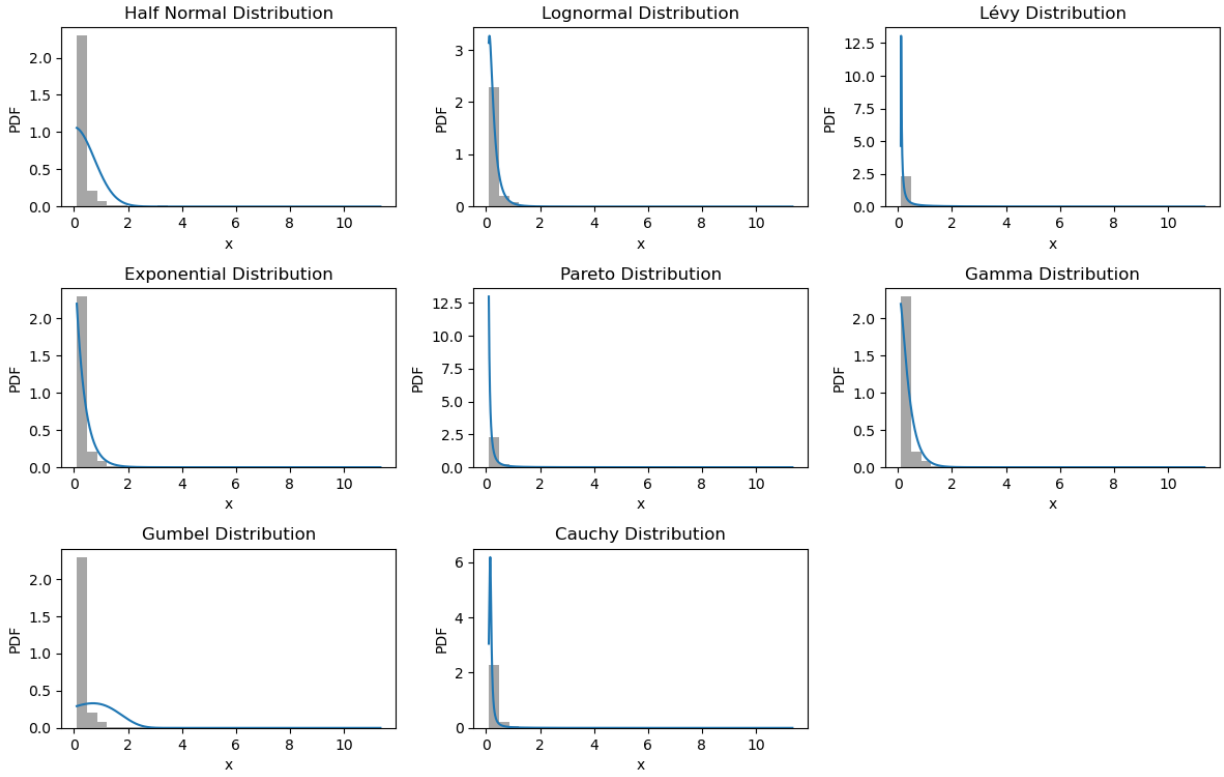


Figure 1.3.1: Dataset_1 distribution PDFs

In [66]:
```
# Create a QQ plot for each distribution against dataset_1
distributions = [lognormal_values, levy_values, exponential_values, paret
distribution_names = ['Lognormal', 'Lévy', 'Exponential', 'Pareto', 'Gamm

fig, axes = plt.subplots(2, 3, figsize=(16, 8))

for i, (dist_values, dist_name) in enumerate(zip(distributions, distribut
    row, col = divmod(i, 3)
    ax = axes[row, col]
    sm.qqplot_2samples(dist_values, dataset_1, ax=ax, line='45')
    ax.set_title(f'QQ Plot: {dist_name} vs. Dataset_1')
    ax.set_xlabel(f'Theoretical Quantiles ({dist_name})')
    ax.set_ylabel(f'Quantiles of Dataset_1')

# Reduce the horizontal gap between subplots
plt.subplots_adjust(left=0.1, bottom=0.1, right=0.9, top=0.9, wspace=0.2,

plt.text(0.5, 0.03, 'Figure 1.3.4: Dataset_1 Q-Q plots', ha='center', va=

# Save the figure
filename = os.path.join(folder_path, 'figure_1.3.4.png')
plt.savefig(filename)

plt.show()
```

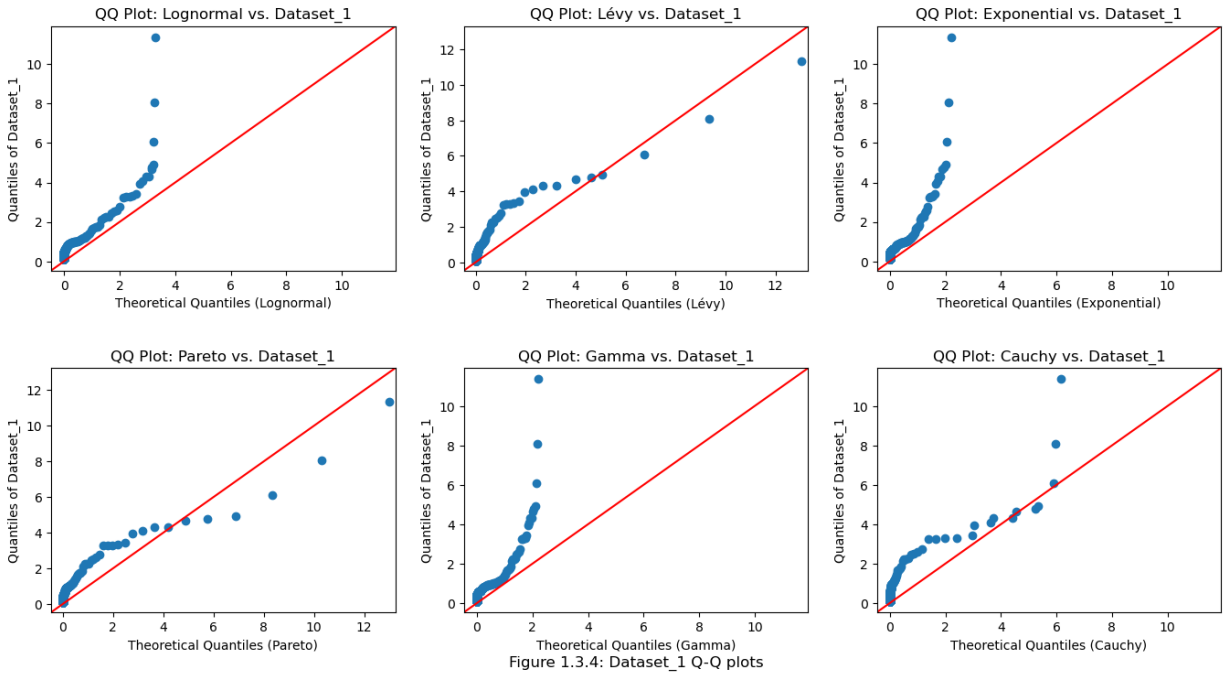


Figure 1.3.4: Dataset_1 Q-Q plots

In [13]:
```
critical_value = np.sqrt(-np.log(0.05/2)/ (2*len(dataset_1)))

print("KS-Test Critical-Value:",critical_value)
print("\n")

# Log-Normal distribution
ks_statistic, ks_p_value = kstest(dataset_1, lambda x: lognormal_cdf(x, o
print("KS Test for Log-Normal:", ks_statistic, ks_p_value)
print("Rejection Note:", reject_or_not(ks_statistic, critical_value))
print("\n")

# Levy distribution
ks_statistic, ks_p_value = kstest(dataset_1, lambda x: levy_cdf(x,  optim
print("KS Test for Levy:", ks_statistic, ks_p_value)
print("Rejection Note:", reject_or_not(ks_statistic, critical_value))
print("\n")

# Pareto distribution
ks_statistic, ks_p_value = kstest(dataset_1, lambda x: pareto_cdf(x,  opt
print("KS Test for Pareto:", ks_statistic, ks_p_value)
print("Rejection Note:", reject_or_not(ks_statistic, critical_value))
print("\n")

# Cauchy distribution
ks_statistic, ks_p_value = kstest(dataset_1, lambda x: cauchy_cdf(x,  opt
print("KS Test for Cauchy:", ks_statistic, ks_p_value)
print("Rejection Note:", reject_or_not(ks_statistic, critical_value))
print("\n")
```

```
KS-Test Critical-Value: 0.04165485256596809


KS Test for Log-Normal: 0.1507559892315071 1.478500285397238e-21
Rejection Note: Reject the null hypothesis


KS Test for Levy: 0.11804690575462295 2.3061170338409826e-13
Rejection Note: Reject the null hypothesis


KS Test for Pareto: 0.01978496454044365 0.7918792622158823
Rejection Note: Fail to reject the null hypothesis


KS Test for Cauchy: 0.24783500252593982 5.34408049155206e-58
Rejection Note: Reject the null hypothesis
```

In [14]:

```
aic_pareto = 2 * nnlf_pareto(optimized_params_pareto, dataset_1) + 2 * le
bic_pareto = 2 * nnlf_pareto(optimized_params_pareto, dataset_1) + np.log
print("Pareto AIC score:", aic_pareto)
print("Pareto BIC score:", bic_pareto)
print("\n")
```

```
Pareto AIC score: -1684.9452844782181
Pareto BIC score: -1675.0075837215343
```

In [15]:
```
from tqdm.notebook import tqdm as tqdm # Import tqdm for notebooks

# Bootstrapping
n_iterations = 10000
bootstrap_alpha = []
bootstrap_k = []

for _ in tqdm(range(n_iterations), desc="Bootstrapping"):
    bootstrap_sample = np.random.choice(dataset_1, size=len(dataset_1), r
    bounds_pareto = [(epsilon, min(bootstrap_sample)), (epsilon, 100)]  #
    values = dual_annealing(nnlf_pareto, bounds=bounds_pareto, args=(boot
    bootstrap_k.append(values.x[0])
    bootstrap_alpha.append(values.x[1])

# Calculate confidence intervals
confidence_intervals_alpha = np.percentile(bootstrap_alpha, [2.5, 97.5],
confidence_intervals_k = np.percentile(bootstrap_k, [2.5, 97.5], axis=0)

mean_alpha = np.mean(bootstrap_alpha)
mean_k = np.mean(bootstrap_k)
print("Bootstrapped Estimate a:", mean_alpha)
print("Bootstrapped Estimates k:", mean_k)
print("\n")

print("95% Confidence Intervals a:", confidence_intervals_alpha)
print("95% Confidence Intervals k:", confidence_intervals_k)
print("\n")
```

```
Bootstrapping:   0%|          | 0/10000 [00:00<?, ?it/s]
Bootstrapped Estimate a: 1.3022483299450218
Bootstrapped Estimates k: 0.10034308439351818


95% Confidence Intervals a: [1.22849811 1.38074869]
95% Confidence Intervals k: [0.10032668 0.10044766]
```

In [16]:
```
bootstrap_standard_error_alpha = 0
for i in range(n_iterations):
    bootstrap_standard_error_alpha = bootstrap_standard_error_alpha + (bo
bootstrap_standard_error_alpha = np.sqrt(bootstrap_standard_error_alpha/(



bootstrap_standard_error_k = 0
for i in range(n_iterations):
    bootstrap_standard_error_k = bootstrap_standard_error_k + (bootstrap_
bootstrap_standard_error_k = np.sqrt(bootstrap_standard_error_k/(n_iterat

# 1.96 is z-value for 95% confidence interval
print("PARAMETER(Alpha):")
print("-----------------------------------------------")

lower_CI_alpha = mean_alpha - 1.96 * bootstrap_standard_error_alpha
upper_CI_alpha = mean_alpha + 1.96 * bootstrap_standard_error_alpha
```

```
print("Percentile 95% bounds for Parameter(alpha)")
print(lower_CI_alpha)
print(upper_CI_alpha)
print("\n")

bootstrap_bias_alpha = 0
for i in range(len(bootstrap_alpha)):
    bootstrap_bias_alpha = bootstrap_bias_alpha + (mean_alpha - bootstrap

bootstrap_bias_alpha = bootstrap_bias_alpha / len(bootstrap_alpha)
p_0_alpha = np.count_nonzero(bootstrap_alpha >= mean_alpha)/len(bootstrap

print("P_0 for Parameter(a)")
print(p_0_alpha)
print("\n")
print("Bootstrap bias for Parameter(a)")
print(bootstrap_bias_alpha)
print("\n")


z_0_alpha = scipy.stats.norm.ppf(p_0_alpha)

print("Z_0 for Parameter(a)")
print(z_0_alpha)
print("\n")


bc_lower_CI_alpha = np.percentile(bootstrap_alpha, 100 * scipy.stats.norm
bc_upper_CI_alpha = np.percentile(bootstrap_alpha, 100 * scipy.stats.norm

print("Bias corrected 95% bounds for Parameter(a)")
print(bc_lower_CI_alpha)
print(bc_upper_CI_alpha)
print("\n")

estimates_alpha = jackknife(bootstrap_alpha,jackknife_mean)
jackknife_estimate_alpha = sum(jackknife(bootstrap_alpha,jackknife_mean))

standard_error_rate_alpha = 0
se_up_alpha = 0
se_down_alpha = 0

for i in range(len(bootstrap_alpha)):
    se_up_alpha = se_up_alpha + (estimates_alpha[i] - jackknife_estimate_
    se_down_alpha = se_down_alpha + (estimates_alpha[i] - jackknife_estim

standard_error_rate_alpha = (1/6) * (se_up_alpha / se_down_alpha**1.5)

print("Rate of Change of Standard Error Parameter(a) (Jackknife accelerat
print("\n")

bca_up_upper_alpha = z_0_alpha + scipy.stats.norm.ppf(0.05/2)
bca_down_upper_alpha = 1- standard_error_rate_alpha*(z_0_alpha +scipy.sta
bca_up_lower_alpha = z_0_alpha + scipy.stats.norm.ppf(1 - 0.05/2)
bca_down_lower_alpha = 1- standard_error_rate_alpha*(z_0_alpha +scipy.sta
bca_upper_CI_alpha = np.percentile(bootstrap_alpha, 100*scipy.stats.norm.
bca_lower_CI_alpha = np.percentile(bootstrap_alpha, 100*scipy.stats.norm.
```

```
print("Bias corrected accelerated 95% bounds for Parameter(a)")
print(bca_upper_CI_alpha)
print(bca_lower_CI_alpha)
print("\n")

print("PARAMETER(K):")
print("-----------------------------------------------")

lower_CI_k = mean_k - 1.96 * bootstrap_standard_error_k
upper_CI_k = mean_k + 1.96 * bootstrap_standard_error_k
print("Percentile 95% bounds for Parameter(k)")
print(lower_CI_k)
print(upper_CI_k)
print("\n")

bootstrap_bias_k = 0
for i in range(len(bootstrap_k)):
    bootstrap_bias_k = bootstrap_bias_k + (mean_k - bootstrap_k[i])

bootstrap_bias_k = bootstrap_bias_k / len(bootstrap_k)
p_0_k = np.count_nonzero(bootstrap_k >= mean_k)/len(bootstrap_k)

print("P_0 for Parameter(k)")
print(p_0_k)
print("\n")
print("Bootstrap bias for Parameter(k)")
print(bootstrap_bias_k)
print("\n")


z_0_k = scipy.stats.norm.ppf(p_0_k)

print("Z_0 for Parameter(k)")
print(z_0_k)
print("\n")


bc_lower_CI_k = np.percentile(bootstrap_k, 100 * scipy.stats.norm.cdf(2 *
bc_upper_CI_k = np.percentile(bootstrap_k, 100 * scipy.stats.norm.cdf(2 *

print("Bias corrected 95% bounds for Parameter(k)")
print(bc_lower_CI_k)
print(bc_upper_CI_k)
print("\n")

estimates_k = jackknife(bootstrap_k,jackknife_mean)
jackknife_estimate_k = sum(jackknife(bootstrap_k,jackknife_mean)) / len(b

standard_error_rate_k = 0
se_up_k = 0
se_down_k = 0

for i in range(len(bootstrap_k)):
    se_up_k = se_up_k + (estimates_k[i] - jackknife_estimate_k)**3
    se_down_k = se_down_k + (estimates_k[i] - jackknife_estimate_k)**2
```

```
standard_error_rate_k = (1/6) * (se_up_k / se_down_k**1.5)

print("Rate of Change of Standard Error Parameter(k) (Jackknife accelerat
print("\n")

bca_up_upper_k = z_0_k + scipy.stats.norm.ppf(0.05/2)
bca_down_upper_k = 1- standard_error_rate_k*(z_0_k +scipy.stats.norm.ppf(
bca_up_lower_k = z_0_k + scipy.stats.norm.ppf(1 - 0.05/2)
bca_down_lower_k = 1- standard_error_rate_k*(z_0_k +scipy.stats.norm.ppf(
bca_upper_CI_k = np.percentile(bootstrap_k, 100*scipy.stats.norm.cdf(z_0_
bca_lower_CI_k = np.percentile(bootstrap_k, 100*scipy.stats.norm.cdf(z_0_

print("Bias corrected accelerated 95% bounds for Parameter(k)")
print(bca_upper_CI_k)
print(bca_lower_CI_k)
print("\n")
```

```
PARAMETER(Alpha):
-----------------------------------------------
Percentile 95% bounds for Parameter(alpha)
1.2258533222352457
1.378643337654798


P_0 for Parameter(a)
0.491


Bootstrap bias for Parameter(a)
2.7824409443155675e-16


Z_0 for Parameter(a)
-0.02256156839022472


Bias corrected 95% bounds for Parameter(a)
1.227131532423647
1.3786805671358673


Rate of Change of Standard Error Parameter(a) (Jackknife accelerated valu
e): -0.0002395040726381158


Bias corrected accelerated 95% bounds for Parameter(a)
1.2270642603090596
1.3786426392553348


PARAMETER(K):
-----------------------------------------------
Percentile 95% bounds for Parameter(k)
0.10028071415892476
0.1004054546281116
```

```
P_0 for Parameter(k)
0.3663


Bootstrap bias for Parameter(k)
1.3697376566312868e-18


Z_0 for Parameter(k)
-0.34166900533109773


Bias corrected 95% bounds for Parameter(k)
0.1003266787694308
0.10036790888795988


Rate of Change of Standard Error Parameter(k) (Jackknife accelerated valu
e): -0.005377303698663638


Bias corrected accelerated 95% bounds for Parameter(k)
0.1003266787694308
0.10036790888795988
```

In [71]:
```
import matplotlib.pyplot as plt

print("Bootstrap mean a:", mean_alpha)
print("Bootstrap standard error a:", bootstrap_standard_error_alpha)

# Plot for Parameter(a)
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.hist(bootstrap_alpha, bins=60, edgecolor='black', alpha=0.75)
plt.axvline(mean_alpha, color='blue', linestyle='dashed', linewidth=2, la
plt.axvline(lower_CI_alpha, color='red', linestyle='dashed', linewidth=2,
plt.axvline(upper_CI_alpha, color='red', linestyle='dashed', linewidth=2)
plt.axvline(bc_lower_CI_alpha, color='green', linestyle='dashed', linewid
plt.axvline(bc_upper_CI_alpha, color='green', linestyle='dashed', linewid
plt.axvline(bca_lower_CI_alpha, color='cyan', linestyle='dashed', linewid
plt.axvline(bca_upper_CI_alpha, color='cyan', linestyle='dashed', linewid
plt.title('Bootstrap Distribution of Parameter(a) for Pareto')
plt.xlabel('Parameter(a) Coefficient')
plt.ylabel('Frequency')
plt.legend(loc='upper left', bbox_to_anchor=(0, 1), fontsize='small')

# Bootstrap results for Parameter(b)
print("Bootstrap mean k:", mean_k)
print("Bootstrap standard error k:", bootstrap_standard_error_k)

plt.subplot(1, 2, 2)
plt.hist(bootstrap_k, bins=5, edgecolor='black', alpha=0.75)
plt.axvline(mean_k, color='blue', linestyle='dashed', linewidth=2, label=
plt.axvline(lower_CI_k, color='red', linestyle='dashed', linewidth=2, lab
plt.axvline(upper_CI_k, color='red', linestyle='dashed', linewidth=2)
plt.axvline(bc_lower_CI_k, color='green', linestyle='dashed', linewidth=2
plt.axvline(bc_upper_CI_k, color='green', linestyle='dashed', linewidth=2
plt.axvline(bca_lower_CI_k, color='cyan', linestyle='dashed', linewidth=2
plt.axvline(bca_upper_CI_k, color='cyan', linestyle='dashed', linewidth=2
plt.title('Bootstrap Distribution of Parameter(k) for Pareto')
plt.xlabel('Parameter(k) Coefficient')
plt.ylabel('Frequency')
plt.legend(loc='upper left', bbox_to_anchor=(0, 1), fontsize='small')

plt.text(0.5, 0.01, 'Figure 1.5.1: Dataset_1 Pareto Distribution Paramete

# Save the figure
filename = os.path.join(folder_path, 'figure_1.5.1.png')
plt.savefig(filename)

plt.show()
```

```
Bootstrap mean a: 1.3022483299450218
Bootstrap standard error a: 0.03897704474988576
Bootstrap mean k: 0.10034308439351818
Bootstrap standard error k: 3.182154826194791e-05
```

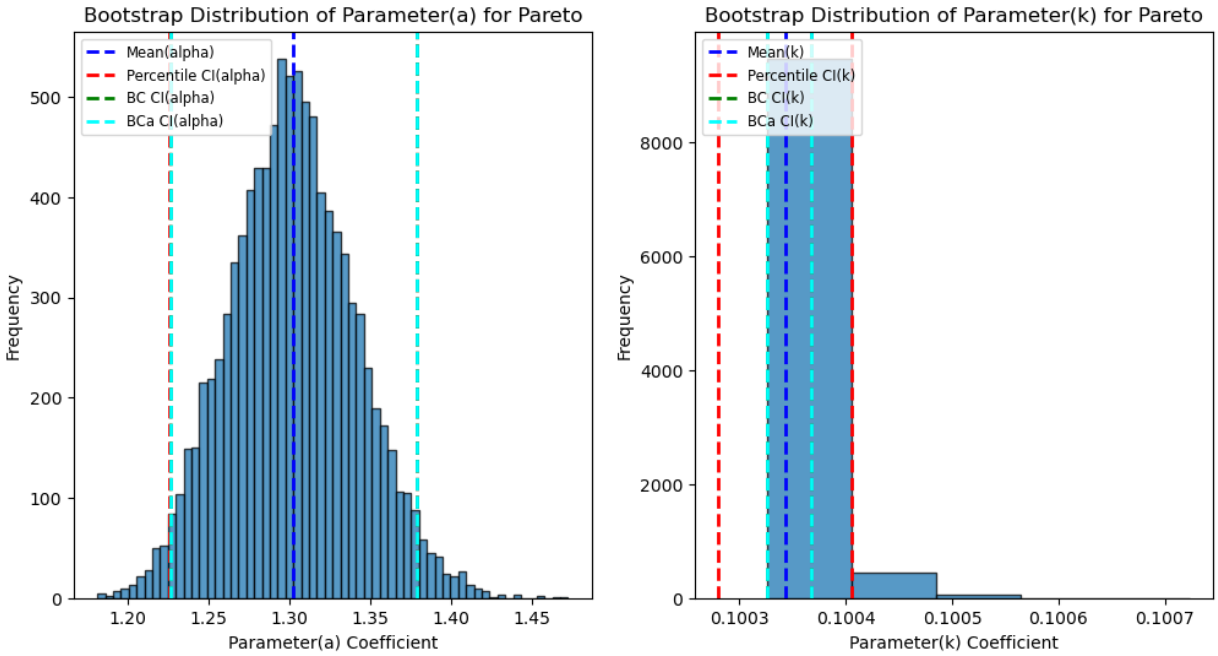


Figure 1.5.1: Dataset_1 Pareto Distribution Parameter 95% Confidence Interval

In [ ]:

Section 1: Datatset_2 model selection

In [19]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
import scipy
import csv
import pandas as pd
import seaborn as sns
import random
import statsmodels.api as sm
from scipy.integrate import quad
from scipy.special import gamma as gamma_function
from scipy.optimize import dual_annealing
from scipy.stats import kstest, yeojohnson
```

In [22]:
```
folder_path = '/Users/biligee/Documents/Classwork/semester_4/Stats/Final_

dataset_2 = np.genfromtxt('data2.csv', delimiter=',')
```

In [41]:
```
# Create a figure with 1 row and 2 columns of subplots
plt.figure(figsize=(12, 5))

# Subplot 1
plt.subplot(1, 2, 1)
plt.hist(dataset_2, bins=30, edgecolor='black')
plt.title('Histogram dataset_2')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Subplot 2
plt.subplot(1, 2, 2)
plt.hist(dataset_2, bins=30, range=(np.percentile(dataset_2, 2.5), np.per
plt.title('Histogram dataset_2: 95% percentile')
plt.xlabel('Value')
plt.ylabel('Frequency')

plt.text(0.5, 0.05, 'Figure 1.1.2: Dataset_2 Histogram', ha='center', va=

# Adjust layout to prevent overlapping
plt.tight_layout()


# Save the figure
filename = os.path.join(folder_path, 'figure_1.1.2.png')
plt.savefig(filename)
# Display the plot
plt.show()
```

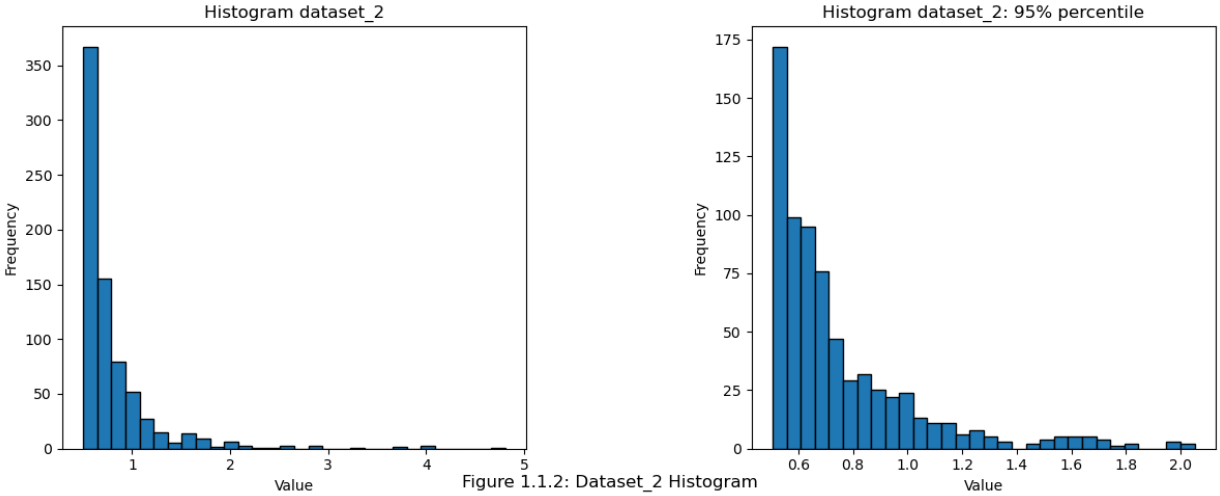


Figure 1.1.2: Dataset_2 Histogram

HalfNormal Distribution: $\frac{2e^{-\frac{x^2\theta^2}{\pi}}\theta}{\pi}$

Lognormal Distribution: $\frac{e^{-\frac{(-\mu+\ln x)^2}{2\sigma^2}}}{\sqrt{2\pi}\cdot x\cdot\sigma}$

Lévy Distribution: $\frac{e^{-\frac{\sigma}{2(x-\mu)}}\left(\frac{\sigma}{x-\mu}\right)^{\frac{3}{2}}}{\sqrt{2\pi}\sigma}$

Exponential Distribution: $e^{-x\lambda}\lambda$

Pareto Distribution: $k^{\alpha}\cdot x^{-1-\alpha}\cdot\alpha$

Gamma Distribution: $\frac{e^{-\frac{x}{\beta}}x^{-1+\alpha}\beta^{-\alpha}}{\Gamma(\alpha)}$

Gumbel Distribution: $\frac{e^{-e^{\frac{x-\alpha}{\beta}}+\frac{x-\alpha}{\beta}}}{\beta}$

Cauchy Distribution: $\frac{1}{b\pi\left(1+\frac{(-a+x)^2}{b^2}\right)}$

In [4]:
```
def half_normal(x, theta):
    if not (np.all(x > 0) and np.all(theta > 0)):
        raise ValueError("Both x and theta must be greater than 0")
    return (2 * np.exp(-x**2 * theta**2 / np.pi) * theta) / np.pi

def lognormal(x, mu, sigma):
    if not (np.all(x > 0) and np.all(sigma > 0)):
        raise ValueError("Both x and sigma must be greater than 0")
    return (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))) / (np.sqrt(2 *

def levy(x, mu, sigma):
    if not (np.all(x > mu) and np.all(sigma > 0)):
        raise ValueError("x must be greater than mu, and sigma must be gr
    return (np.exp(-sigma / (2 * (x - mu))) * (sigma / (x - mu))**(3/2))

def exponential(x, lam):
    if not np.all(lam > 0):
        raise ValueError("Lambda must be greater than 0")
    return np.exp(-x * lam) * lam

def pareto(x, k, alpha):
    if not (np.all(x >= k) and np.all(k > 0) and np.all(alpha > 0)):
        raise ValueError("x must be greater than or equal to k, and both
    return k**alpha * x**(-1 - alpha) * alpha

def gamma_distribution(x, alpha, beta):
    if not (np.all(x > 0) and np.all(alpha > 0) and np.all(beta > 0)):
        raise ValueError("x, alpha, and beta must all be greater than 0")
    return (np.exp(-x / beta) * x**(alpha - 1) * beta**(-alpha)) / gamma_

def gumbel(x, alpha, beta):
    if not np.all(beta > 0):
        raise ValueError("Beta must be greater than 0")
    return np.exp(-np.exp((x - alpha) / beta) + (x - alpha) / beta) / bet


def cauchy_distribution(x, a, b):
    if not np.all(b > 0):
        raise ValueError("Scale parameter (b) must be greater than 0")
    return 1 / (b * np.pi * (1 + ((-a + x)**2 / b**2)))
```

In [5]:
```
epsilon = 1e-10

def nnlf_half_normal(params, data):
    theta = params[0]
    if not (np.all(data > 0) and np.all(theta > 0)):
        return np.inf
    half_normal_values = half_normal(data, theta)
    valid_values = np.isfinite(half_normal_values)
    log_likelihood = -np.sum(np.log(np.where(half_normal_values[valid_val
    return log_likelihood

def nnlf_lognormal(params, data):
    mu, sigma = params
    if not (np.all(data > 0) and np.all(sigma > 0)):
        return np.inf
```

```
    lognormal_values = lognormal(data, mu, sigma)
    valid_values = np.isfinite(lognormal_values)
    log_likelihood = -np.sum(np.log(np.where(lognormal_values[valid_value
    return log_likelihood

def nnlf_levy(params, data):
    mu, sigma = params
    if not (np.all(data > mu) and np.all(sigma > 0)):
        return np.inf
    levy_values = levy(data, mu, sigma)
    valid_values = np.isfinite(levy_values)
    log_likelihood = -np.sum(np.log(np.where(levy_values[valid_values] ==
    return log_likelihood

def nnlf_exponential(params, data):
    lam = params[0]
    if not (np.all(lam > 0)):
        return np.inf
    exponential_values = exponential(data, lam)
    valid_values = np.isfinite(exponential_values)
    log_likelihood = -np.sum(np.log(np.where(exponential_values[valid_val
    return log_likelihood

def nnlf_pareto(params, data):
    k, alpha = params
    if not (np.all(data >= k) and np.all(k > 0) and np.all(alpha > 0)):
        return np.inf
    pareto_values = pareto(data, k, alpha)
    valid_values = np.isfinite(pareto_values)
    log_likelihood = -np.sum(np.log(np.where(pareto_values[valid_values]
    return log_likelihood

def nnlf_gamma(params, data):
    alpha, beta = params
    if not (np.all(data > 0) and np.all(alpha > 0) and np.all(beta > 0)):
        return np.inf
    gamma_values = gamma_distribution(data, alpha, beta)
    valid_values = np.isfinite(gamma_values)
    log_likelihood = -np.sum(np.log(np.where(gamma_values[valid_values] =
    return log_likelihood

def nnlf_gumbel(params, data):
    alpha, beta = params
    if not np.all(beta > 0):
        return np.inf
    gumbel_values = gumbel(data, alpha, beta)
    valid_values = np.isfinite(gumbel_values)
    log_likelihood = -np.sum(np.log(np.where(gumbel_values[valid_values]

    return log_likelihood

def nnlf_cauchy(params, data):
    a, b = params
    if not np.all(b > 0):
        return np.inf
    cauchy_values = cauchy_distribution(data, a, b)
    valid_values = np.isfinite(cauchy_values)
```

```
    log_likelihood = -np.sum(np.log(np.where(cauchy_values[valid_values]
    return log_likelihood
```

In [6]:

```
def half_normal_cdf(x, theta):
    if not (np.all(x > 0) and np.all(theta > 0)):
        raise ValueError("Both x and theta must be greater than 0")
    integrand = lambda t: half_normal(t, theta)
    result, _ = quad(integrand, 0, x)
    return result

def lognormal_cdf(x, mu, sigma):
    if not (np.all(x > 0) and np.all(sigma > 0)):
        raise ValueError("Both x and sigma must be greater than 0")
    integrand = lambda t: lognormal(t, mu, sigma)
    result, _ = quad(integrand, 0, x)
    return result

def levy_cdf(x, mu, sigma):
    if not (np.all(x > mu) and np.all(sigma > 0)):
        raise ValueError("x must be greater than mu, and sigma must be gr
    integrand = lambda t: levy(t, mu, sigma)
    result, _ = quad(integrand, mu+epsilon, x)
    return result

def exponential_cdf(x, lam):
    if not np.all(lam > 0):
        raise ValueError("Lambda must be greater than 0")
    integrand = lambda t: exponential(t, lam)
    result, _ = quad(integrand, 0, x)
    return result

def pareto_cdf(x, k, alpha):
    if not (np.all(x >= k) and np.all(k > 0) and np.all(alpha > 0)):
        raise ValueError("x must be greater than or equal to k, and both
    integrand = lambda t: pareto(t, k, alpha)
    result, _ = quad(integrand, k+epsilon, x)
    return result

def gamma_cdf(x, alpha, beta):
    if not (np.all(x > 0) and np.all(alpha > 0) and np.all(beta > 0)):
        raise ValueError("x, alpha, and beta must all be greater than 0")
    integrand = lambda t: gamma_distribution(t, alpha, beta)
    result, _ = quad(integrand, 0, x)
    return result

def gumbel_cdf(x, alpha, beta):
    if not np.all(beta > 0):
        raise ValueError("Beta must be greater than 0")
    integrand = lambda t: gumbel(t, alpha, beta)
    result, _ = quad(integrand, -np.inf, x)
    return result

def cauchy_cdf(x, a, b):
    if not np.all(b > 0):
        raise ValueError("Scale parameter (b) must be greater than 0")
    integrand = lambda t: cauchy_distribution(t, a, b)
```

```
    result, _ = quad(integrand, -np.inf, x)
    return result

half_normal_cdf = np.vectorize(half_normal_cdf)
lognormal_cdf = np.vectorize(lognormal_cdf)
levy_cdf = np.vectorize(levy_cdf)
exponential_cdf = np.vectorize(exponential_cdf)
pareto_cdf = np.vectorize(pareto_cdf)
gamma_cdf = np.vectorize(gamma_cdf)
gumbel_cdf = np.vectorize(gumbel_cdf)
cauchy_cdf = np.vectorize(cauchy_cdf)
```

In [7]:
```
def jackknife_mean(dataset):
    return sum(dataset)/len(dataset)

def jackknife(dataset, statistic_function):
    n = len(dataset)
    jackknife_estimates = []

    for i in range(n):
        leave_one_out_data = np.delete(dataset, i)
        statistic_value = statistic_function(leave_one_out_data)
        jackknife_estimates.append(statistic_value)


    return jackknife_estimates
```

In [8]:
```
def reject_or_not(ks_statistic, critical):
    if ks_statistic > critical:
        return "Reject the null hypothesis"
    else:
        return "Fail to reject the null hypothesis"
```

In [9]:
```
# Half-Normal Distribution
bounds_half_normal = [(epsilon, 100)]  # Replace np.inf with a large valu

# Lognormal Distribution
bounds_lognormal = [(-100, 100), (epsilon, 100)]  # Replace np.inf with a

# Lévy Distribution
bounds_levy = [(-100, min(dataset_2) - epsilon), (epsilon, 100)]  # Repla

# Exponential Distribution
bounds_exponential = [(epsilon, 100)]  # Replace np.inf with a large valu

# Pareto Distribution
bounds_pareto = [(epsilon, min(dataset_2)), (epsilon, 100)]  # Replace No

# Gamma Distribution
bounds_gamma = [(epsilon, 10), (epsilon, 10)]  # Replace None and np.inf

# Gumbel Distribution
bounds_gumbel = [(-100, 100), (epsilon, 100)]  # Replace np.inf with a la

# Cauchy Distribution
bounds_cauchy = [(-100, 100), (epsilon, 100)]  # Replace np.inf with a la


# Define bounds for other distributions in a similar manner

result_half_normal = dual_annealing(nnlf_half_normal, bounds=bounds_half_
result_lognormal = dual_annealing(nnlf_lognormal, bounds=bounds_lognormal
result_levy = dual_annealing(nnlf_levy, bounds=bounds_levy, args=(dataset
result_exponential = dual_annealing(nnlf_exponential, bounds=bounds_expon
result_pareto = dual_annealing(nnlf_pareto, bounds=bounds_pareto, args=(d
result_gamma = dual_annealing(nnlf_gamma, bounds=bounds_gamma, args=(data
result_gumbel = dual_annealing(nnlf_gumbel, bounds=bounds_gumbel, args=(d
result_cauchy = dual_annealing(nnlf_cauchy, bounds=bounds_cauchy, args=(d

# Access the optimized parameters
optimized_params_half_normal = result_half_normal.x
optimized_params_lognormal = result_lognormal.x
optimized_params_levy = result_levy.x
optimized_params_exponential = result_exponential.x
optimized_params_pareto = result_pareto.x
optimized_params_gamma = result_gamma.x
optimized_params_gumbel = result_gumbel.x
optimized_params_cauchy = result_cauchy.x

# Print or use the optimized parameters as needed
print("Optimized Parameters - Half-Normal:", optimized_params_half_normal
print("Optimized Parameters - Lognormal:", optimized_params_lognormal)
print("Optimized Parameters - Lévy:", optimized_params_levy)
print("Optimized Parameters - Exponential:", optimized_params_exponential
print("Optimized Parameters - Pareto:", optimized_params_pareto)
print("Optimized Parameters - Gamma:", optimized_params_gamma)
print("Optimized Parameters - Gumbel:", optimized_params_gumbel)
print("Optimized Parameters - Cauchy:", optimized_params_cauchy)
```

```
/var/folders/rm/7h6vlsd94v9b6hq6m0qjz4680000gn/T/ipykernel_54266/64895940
8.py:34: RuntimeWarning: overflow encountered in exp
  return np.exp(-np.exp((x - alpha) / beta) + (x - alpha) / beta) / beta
Optimized Parameters - Half-Normal: [1.34965799]
Optimized Parameters - Lognormal: [-0.31187334  0.37717188]
Optimized Parameters - Lévy: [0.49298647 0.07236848]
Optimized Parameters - Exponential: [1.2468024]
Optimized Parameters - Pareto: [0.50086095 2.63467525]
Optimized Parameters - Gamma: [5.63837458 0.14224875]
Optimized Parameters - Gumbel: [1.10213206 0.93070378]
Optimized Parameters - Cauchy: [0.62091525 0.09857946]
```

In [10]:
```
x_values = np.linspace(min(dataset_2), max(dataset_2), len(dataset_2))


# Calculate values for each distribution
half_normal_values = half_normal(x_values, optimized_params_half_normal)
lognormal_values = lognormal(x_values, optimized_params_lognormal[0],opti
levy_values = levy(x_values, optimized_params_levy[0], optimized_params_l
exponential_values = exponential(x_values, optimized_params_exponential)
pareto_values = pareto(x_values, optimized_params_pareto[0], optimized_pa
gamma_values = gamma_distribution(x_values, optimized_params_gamma[0], op
gumbel_values = gumbel(x_values, optimized_params_gumbel[0], optimized_pa
cauchy_values = cauchy_distribution(x_values, optimized_params_cauchy[0],
```

In [24]:
```
# Plot the PDFs
plt.figure(figsize=(12, 8))

plt.subplot(3, 3, 1)
plt.plot(x_values, half_normal_values, label='Half Normal')
plt.hist(dataset_2, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Half Normal Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 2)
plt.plot(x_values, lognormal_values, label='Lognormal')
plt.hist(dataset_2, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Lognormal Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 3)
plt.plot(x_values, levy_values, label='Lévy')
plt.hist(dataset_2, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Lévy Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 4)
plt.plot(x_values, exponential_values, label='Exponential')
plt.hist(dataset_2, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Exponential Distribution')
plt.xlabel('x')
plt.ylabel('PDF')
```

```
plt.subplot(3, 3, 5)
plt.plot(x_values, pareto_values, label='Pareto')
plt.hist(dataset_2, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Pareto Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 6)
plt.plot(x_values, gamma_values, label='Gamma')
plt.hist(dataset_2, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Gamma Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 7)
plt.plot(x_values, gumbel_values, label='Gumbel')
plt.hist(dataset_2, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Gumbel Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.subplot(3, 3, 8)
plt.plot(x_values, cauchy_values, label='Cauchy')
plt.hist(dataset_2, bins=30, density=True, color='gray', alpha=0.7, label
plt.title('Cauchy Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

plt.text(0.5, 0.03, 'Figure 1.3.2: Dataset_2 distribution PDFs', ha='cent

# Adjust layout to prevent overlapping
plt.tight_layout()


# Save the figure
filename = os.path.join(folder_path, 'figure_1.3.2.png')
plt.savefig(filename)

plt.show()
```

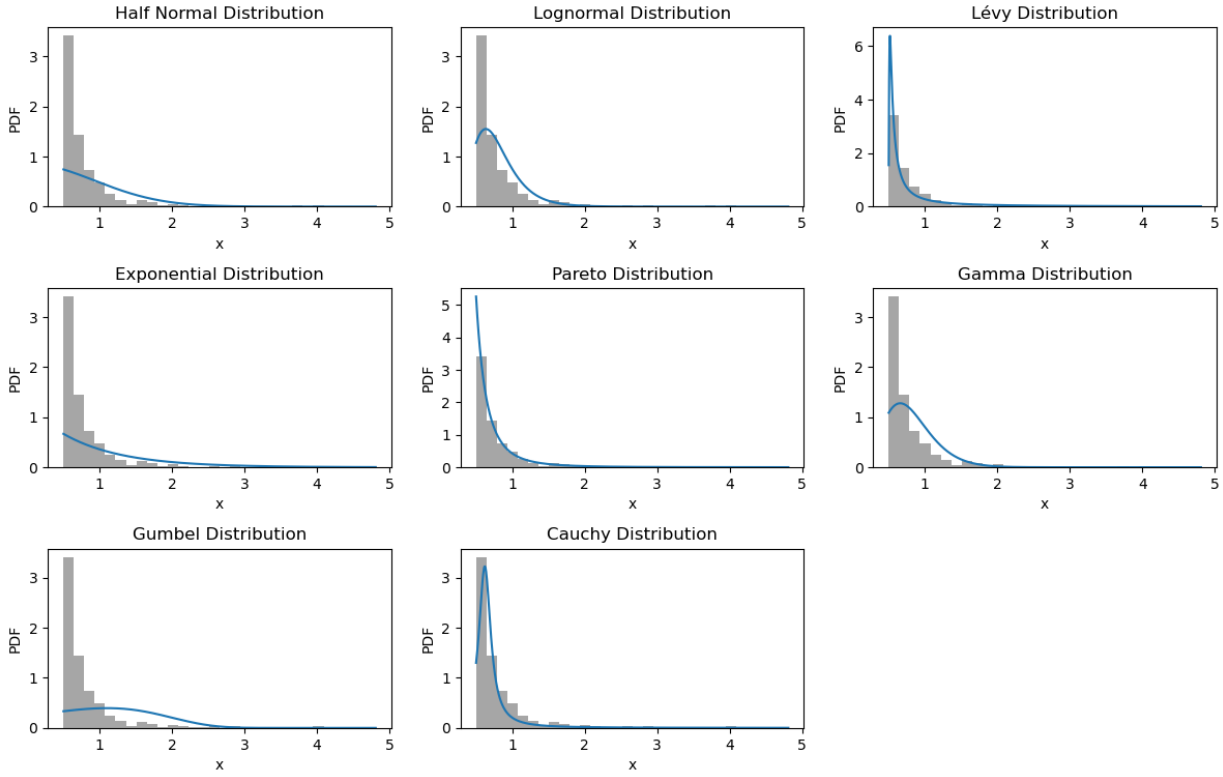


Figure 1.3.2: Dataset_2 distribution PDFs

In [36]:
```
# Create a QQ plot for each distribution against dataset_2
distributions = [lognormal_values, levy_values, pareto_values, gamma_valu
distribution_names = ['Log-Normal', 'Lévy', 'Pareto', 'Gamma', 'Cauchy']

fig, axes = plt.subplots(2, 3, figsize=(16, 8))

for i, (dist_values, dist_name) in enumerate(zip(distributions, distribut
    row, col = divmod(i, 3)
    ax = axes[row, col]
    if ax != axes[1,2]:
        sm.qqplot_2samples(dist_values, dataset_2, ax=ax, line='45')
        ax.set_title(f'QQ Plot: {dist_name} vs. Dataset_2')
        ax.set_xlabel(f'Theoretical Quantiles ({dist_name})')
        ax.set_ylabel(f'Quantiles of Dataset_2')

# Reduce the horizontal gap between subplots and eliminate vertical gap
plt.subplots_adjust(left=0.1, bottom=0.1, right=0.9, top=0.9, wspace=0.2,

plt.text(0.5, 0.03, 'Figure 1.3.5: Dataset_2 Q-Q plots', ha='center', va=

# Save the figure
filename = os.path.join(folder_path, 'figure_1.3.5.png')
plt.savefig(filename)

plt.show()
```

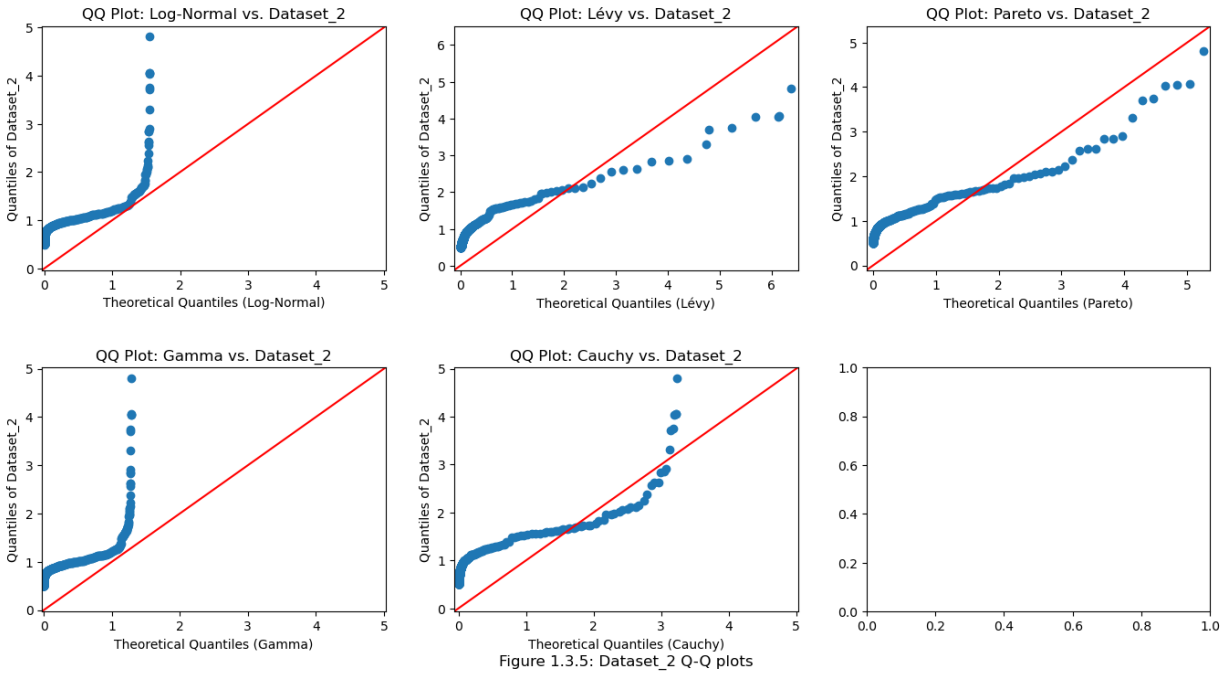


Figure 1.3.5: Dataset_2 Q-Q plots

In [37]:
```
critical_value = np.sqrt(-np.log(0.05/2)/ (2*len(dataset_2)))

print("KS-Test Critical-Value:",critical_value)
print("\n")

# Log-Normal distribution
ks_statistic, ks_p_value = kstest(dataset_2, lambda x: lognormal_cdf(x, o
print("KS Test for Log-Normal:", ks_statistic, ks_p_value)
print("Rejection Note:", reject_or_not(ks_statistic, critical_value))
print("\n")

# Pareto distribution
ks_statistic, ks_p_value = kstest(dataset_2, lambda x: pareto_cdf(x,  opt
print("KS Test for Pareto:", ks_statistic, ks_p_value)
print("Rejection Note:", reject_or_not(ks_statistic, critical_value))
print("\n")


# Cauchy distribution
ks_statistic, ks_p_value = kstest(dataset_2, lambda x: cauchy_cdf(x,  opt
print("KS Test for Cauchy:", ks_statistic, ks_p_value)
print("Rejection Note:", reject_or_not(ks_statistic, critical_value))
print("\n")
```

```
KS-Test Critical-Value: 0.049623949371356016


KS Test for Log-Normal: 0.15713223042885965 1.2814614924474841e-16
Rejection Note: Reject the null hypothesis


KS Test for Pareto: 0.01998643980438458 0.9198535644808467
Rejection Note: Fail to reject the null hypothesis


KS Test for Cauchy: 0.21883466118085174 5.63474418926739e-32
Rejection Note: Reject the null hypothesis
```

In [14]:

```
aic_pareto = 2 * nnlf_pareto(optimized_params_pareto, dataset_2) + 2 * le
bic_pareto = 2 * nnlf_pareto(optimized_params_pareto, dataset_2) + np.log
print("Pareto AIC score:", aic_pareto)
print("Pareto BIC score:", bic_pareto)
print("\n")
```

```
Pareto AIC score: -416.38862839751573
Pareto BIC score: -407.15115043048127
```

In [15]:

```
from tqdm.notebook import tqdm as tqdm # Import tqdm for notebooks

# Bootstrapping
n_iterations = 10000
bootstrap_alpha = []
bootstrap_k = []

for _ in tqdm(range(n_iterations), desc="Bootstrapping"):
    bootstrap_sample = np.random.choice(dataset_2, size=len(dataset_2), r
    bounds_pareto = [(epsilon, min(bootstrap_sample)), (epsilon, 100)]  #
    values = dual_annealing(nnlf_pareto, bounds=bounds_pareto, args=(boot
    bootstrap_k.append(values.x[0])
    bootstrap_alpha.append(values.x[1])

# Calculate confidence intervals
confidence_intervals_alpha = np.percentile(bootstrap_alpha, [2.5, 97.5],
confidence_intervals_k = np.percentile(bootstrap_k, [2.5, 97.5], axis=0)

mean_alpha = np.mean(bootstrap_alpha)
mean_k = np.mean(bootstrap_k)
print("Bootstrapped Estimate a:", mean_alpha)
print("Bootstrapped Estimates k:", mean_k)
print("\n")

print("95% Confidence Intervals a:", confidence_intervals_alpha)
print("95% Confidence Intervals k:", confidence_intervals_k)
print("\n")
```

```
Bootstrapping:   0%|          | 0/10000 [00:00<?, ?it/s]
```

```
Bootstrapped Estimate a: 2.642826301640089
Bootstrapped Estimates k: 0.5011777668827366


95% Confidence Intervals a: [2.46067722 2.83493804]
95% Confidence Intervals k: [0.50086095 0.50205843]
```

In [16]:

```
bootstrap_standard_error_alpha = 0
for i in range(n_iterations):
    bootstrap_standard_error_alpha = bootstrap_standard_error_alpha + (bo
bootstrap_standard_error_alpha = np.sqrt(bootstrap_standard_error_alpha/(



bootstrap_standard_error_k = 0
for i in range(n_iterations):
    bootstrap_standard_error_k = bootstrap_standard_error_k + (bootstrap_
bootstrap_standard_error_k = np.sqrt(bootstrap_standard_error_k/(n_iterat

# 1.96 is z-value for 95% confidence interval
print("PARAMETER(Alpha):")
print("------------------------------------------------")

lower_CI_alpha = mean_alpha - 1.96 * bootstrap_standard_error_alpha
upper_CI_alpha = mean_alpha + 1.96 * bootstrap_standard_error_alpha
print("Percentile 95% bounds for Parameter(alpha)")
print(lower_CI_alpha)
print(upper_CI_alpha)
print("\n")

bootstrap_bias_alpha = 0
for i in range(len(bootstrap_alpha)):
    bootstrap_bias_alpha = bootstrap_bias_alpha + (mean_alpha - bootstrap

bootstrap_bias_alpha = bootstrap_bias_alpha / len(bootstrap_alpha)
p_0_alpha = np.count_nonzero(bootstrap_alpha >= mean_alpha)/len(bootstrap

print("P_0 for Parameter(a)")
print(p_0_alpha)
print("\n")
print("Bootstrap bias for Parameter(a)")
print(bootstrap_bias_alpha)
print("\n")


z_0_alpha = scipy.stats.norm.ppf(p_0_alpha)

print("Z_0 for Parameter(a)")
print(z_0_alpha)
print("\n")


bc_lower_CI_alpha = np.percentile(bootstrap_alpha, 100 * scipy.stats.norm
bc_upper_CI_alpha = np.percentile(bootstrap_alpha, 100 * scipy.stats.norm
```

```
print("Bias corrected 95% bounds for Parameter(a)")
print(bc_lower_CI_alpha)
print(bc_upper_CI_alpha)
print("\n")

estimates_alpha = jackknife(bootstrap_alpha,jackknife_mean)
jackknife_estimate_alpha = sum(jackknife(bootstrap_alpha,jackknife_mean))

standard_error_rate_alpha = 0
se_up_alpha = 0
se_down_alpha = 0

for i in range(len(bootstrap_alpha)):
    se_up_alpha = se_up_alpha + (estimates_alpha[i] - jackknife_estimate_
    se_down_alpha = se_down_alpha + (estimates_alpha[i] - jackknife_estim

standard_error_rate_alpha = (1/6) * (se_up_alpha / se_down_alpha**1.5)

print("Rate of Change of Standard Error Parameter(a) (Jackknife accelerat
print("\n")

bca_up_upper_alpha = z_0_alpha + scipy.stats.norm.ppf(0.05/2)
bca_down_upper_alpha = 1- standard_error_rate_alpha*(z_0_alpha +scipy.sta
bca_up_lower_alpha = z_0_alpha + scipy.stats.norm.ppf(1 - 0.05/2)
bca_down_lower_alpha = 1- standard_error_rate_alpha*(z_0_alpha +scipy.sta
bca_upper_CI_alpha = np.percentile(bootstrap_alpha, 100*scipy.stats.norm.
bca_lower_CI_alpha = np.percentile(bootstrap_alpha, 100*scipy.stats.norm.

print("Bias corrected accelerated 95% bounds for Parameter(a)")
print(bca_upper_CI_alpha)
print(bca_lower_CI_alpha)
print("\n")

print("PARAMETER(K):")
print("-----------------------------------------------")

lower_CI_k = mean_k - 1.96 * bootstrap_standard_error_k
upper_CI_k = mean_k + 1.96 * bootstrap_standard_error_k
print("Percentile 95% bounds for Parameter(k)")
print(lower_CI_k)
print(upper_CI_k)
print("\n")

bootstrap_bias_k = 0
for i in range(len(bootstrap_k)):
    bootstrap_bias_k = bootstrap_bias_k + (mean_k - bootstrap_k[i])

bootstrap_bias_k = bootstrap_bias_k / len(bootstrap_k)
p_0_k = np.count_nonzero(bootstrap_k >= mean_k)/len(bootstrap_k)

print("P_0 for Parameter(k)")
print(p_0_k)
print("\n")
print("Bootstrap bias for Parameter(k)")
print(bootstrap_bias_k)
print("\n")
```

```
z_0_k = scipy.stats.norm.ppf(p_0_k)

print("Z_0 for Parameter(k)")
print(z_0_k)
print("\n")


bc_lower_CI_k = np.percentile(bootstrap_k, 100 * scipy.stats.norm.cdf(2 *
bc_upper_CI_k = np.percentile(bootstrap_k, 100 * scipy.stats.norm.cdf(2 *

print("Bias corrected 95% bounds for Parameter(k)")
print(bc_lower_CI_k)
print(bc_upper_CI_k)
print("\n")

estimates_k = jackknife(bootstrap_k,jackknife_mean)
jackknife_estimate_k = sum(jackknife(bootstrap_k,jackknife_mean)) / len(b

standard_error_rate_k = 0
se_up_k = 0
se_down_k = 0

for i in range(len(bootstrap_k)):
    se_up_k = se_up_k + (estimates_k[i] - jackknife_estimate_k)**3
    se_down_k = se_down_k + (estimates_k[i] - jackknife_estimate_k)**2

standard_error_rate_k = (1/6) * (se_up_k / se_down_k**1.5)

print("Rate of Change of Standard Error Parameter(k) (Jackknife accelerat
print("\n")

bca_up_upper_k = z_0_k + scipy.stats.norm.ppf(0.05/2)
bca_down_upper_k = 1- standard_error_rate_k*(z_0_k +scipy.stats.norm.ppf(
bca_up_lower_k = z_0_k + scipy.stats.norm.ppf(1 - 0.05/2)
bca_down_lower_k = 1- standard_error_rate_k*(z_0_k +scipy.stats.norm.ppf(
bca_upper_CI_k = np.percentile(bootstrap_k, 100*scipy.stats.norm.cdf(z_0_
bca_lower_CI_k = np.percentile(bootstrap_k, 100*scipy.stats.norm.cdf(z_0_

print("Bias corrected accelerated 95% bounds for Parameter(k)")
print(bca_upper_CI_k)
print(bca_lower_CI_k)
print("\n")
```

```
PARAMETER(Alpha):
------------------------------------------------
Percentile 95% bounds for Parameter(alpha)
2.456561699146023
2.8290909041341545


P_0 for Parameter(a)
0.4928


Bootstrap bias for Parameter(a)
6.9277916736609765e-18
```

```
Z_0 for Parameter(a)
-0.018048703440788155


Bias corrected 95% bounds for Parameter(a)
2.4579149051203695
2.8313559316729795


Rate of Change of Standard Error Parameter(a) (Jackknife accelerated valu
e): -0.0002082020882556721


Bias corrected accelerated 95% bounds for Parameter(a)
2.45790614620304
2.8312709814533745


PARAMETER(K):
------------------------------------------------
Percentile 95% bounds for Parameter(k)
0.5003339562871251
0.5020215774783482


P_0 for Parameter(k)
0.3663


Bootstrap bias for Parameter(k)
-7.55284723652494e-17


Z_0 for Parameter(k)
-0.34166900533109773


Bias corrected 95% bounds for Parameter(k)
0.500860951183895
0.5016617275831691


Rate of Change of Standard Error Parameter(k) (Jackknife accelerated valu
e): -0.0013587648353302272


Bias corrected accelerated 95% bounds for Parameter(k)
0.500860951183895
0.5016617275831691
```

In [48]:
```
import matplotlib.pyplot as plt

print("Bootstrap mean a:", mean_alpha)
print("Bootstrap standard error a:", bootstrap_standard_error_alpha)

# Plot for Parameter(a)
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.hist(bootstrap_alpha, bins=60, edgecolor='black', alpha=0.75)
plt.axvline(mean_alpha, color='blue', linestyle='dashed', linewidth=2, la
plt.axvline(lower_CI_alpha, color='red', linestyle='dashed', linewidth=2,
plt.axvline(upper_CI_alpha, color='red', linestyle='dashed', linewidth=2)
plt.axvline(bc_lower_CI_alpha, color='green', linestyle='dashed', linewid
plt.axvline(bc_upper_CI_alpha, color='green', linestyle='dashed', linewid
plt.axvline(bca_lower_CI_alpha, color='cyan', linestyle='dashed', linewid
plt.axvline(bca_upper_CI_alpha, color='cyan', linestyle='dashed', linewid
plt.title('Bootstrap Distribution of Parameter(a) for Pareto')
plt.xlabel('Parameter(a) Coefficient')
plt.ylabel('Frequency')
plt.legend(loc='upper left', bbox_to_anchor=(0, 1), fontsize='small')

# Bootstrap results for Parameter(b)
print("Bootstrap mean k:", mean_k)
print("Bootstrap standard error k:", bootstrap_standard_error_k)

# Plot for Parameter(b)
plt.subplot(1, 2, 2)
plt.hist(bootstrap_k, bins=5, edgecolor='black', alpha=0.75)
plt.axvline(mean_k, color='blue', linestyle='dashed', linewidth=2, label=
plt.axvline(lower_CI_k, color='red', linestyle='dashed', linewidth=2, lab
plt.axvline(upper_CI_k, color='red', linestyle='dashed', linewidth=2)
plt.axvline(bc_lower_CI_k, color='green', linestyle='dashed', linewidth=2
plt.axvline(bc_upper_CI_k, color='green', linestyle='dashed', linewidth=2
plt.axvline(bca_lower_CI_k, color='cyan', linestyle='dashed', linewidth=2
plt.axvline(bca_upper_CI_k, color='cyan', linestyle='dashed', linewidth=2
plt.title('Bootstrap Distribution of Parameter(k) for Pareto')
plt.xlabel('Parameter(k) Coefficient')
plt.ylabel('Frequency')
plt.legend(loc='upper left', bbox_to_anchor=(0, 1), fontsize='small')

plt.text(0.5, 0.01, 'Figure 1.5.2: Dataset_2 Pareto Distribution Paramete

# Save the figure
filename = os.path.join(folder_path, 'figure_1.5.2.png')
plt.savefig(filename)

plt.show()
```

```
Bootstrap mean a: 2.642826301640089
Bootstrap standard error a: 0.09503296045615606
Bootstrap mean k: 0.5011777668827366
Bootstrap standard error k: 0.00043051561000589835
```

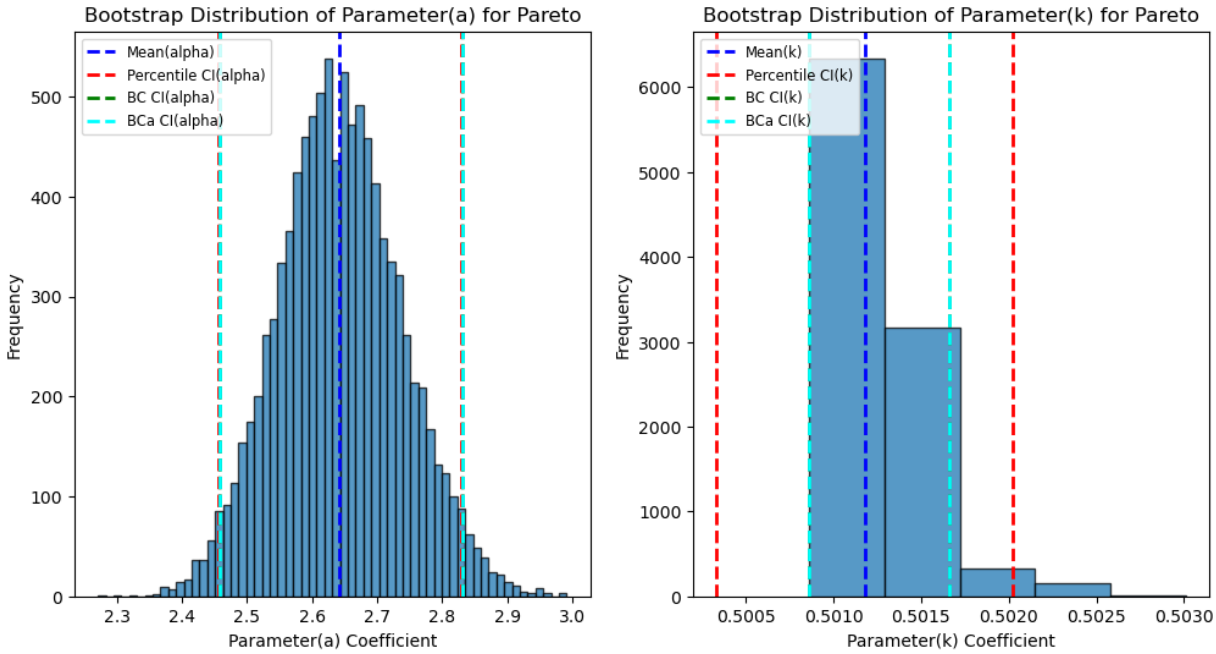


Figure 1.5.2: Dataset_2 Pareto Distribution Parameter 95% Confidence Interval

In [ ]:

Section 1: Dataset_3 model selection

In [90]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
import scipy
import csv
import pandas as pd
import seaborn as sns
import random
import statsmodels.api as sm
from scipy.integrate import quad
from scipy.special import gamma as gamma_function
from scipy.optimize import dual_annealing
from scipy.stats import kstest, yeojohnson
```

In [91]:
```
folder_path = '/Users/biligee/Documents/Classwork/semester_4/Stats/

dataset_3 = np.genfromtxt('data3.csv', delimiter=',')
```

In [131]:
```
# Create a figure with 1 row and 2 columns of subplots
plt.figure(figsize=(12, 5))

# Subplot 1
plt.subplot(1, 2, 1)
plt.hist(dataset_3, bins=30, edgecolor='black')
plt.title('Histogram dataset_3')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Subplot 2
plt.subplot(1, 2, 2)
plt.hist(dataset_3, bins=30, range=(np.percentile(dataset_3, 2.5),
plt.title('Histogram dataset_3: 95% percentile')
plt.xlabel('Value')
plt.ylabel('Frequency')

plt.text(0.5, 0.05, 'Figure 1.1.3: Dataset_3 Histograms', ha='cente

# Adjust layout to prevent overlapping
plt.tight_layout()


# Save the figure
filename = os.path.join(folder_path, 'figure_1.1.3.png')
plt.savefig(filename)
# Display the plot
plt.show()
```

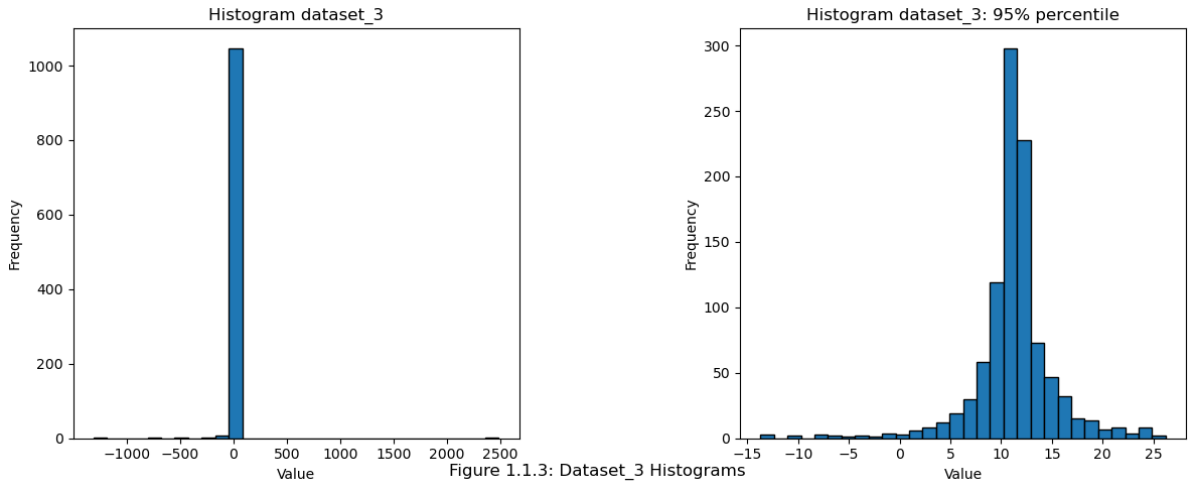
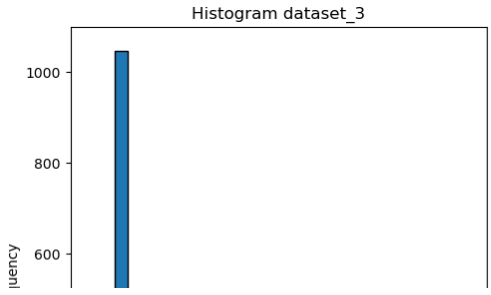
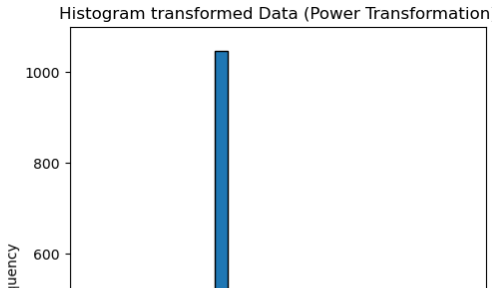


Figure 1.1.3: Dataset_3 Histograms

In [130]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import yeojohnson

# Generate example data with extreme outliers
data = dataset_3  # Assuming dataset_3 is defined

# Plot the original data
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.hist(data, bins=30, edgecolor='black')
plt.title('Histogram dataset_3')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Apply logarithmic transformation to mitigate extreme values
dataset_3, yeo_1 = yeojohnson(data)
dataset_3, yeo_2 = yeojohnson(dataset_3)

# Plot the transformed data
plt.subplot(1, 2, 2)
plt.hist(dataset_3, bins=30, edgecolor='black')
plt.title('Histogram transformed Data (Power Transformation)')
plt.xlabel('Value')
plt.ylabel('Frequency')

plt.text(0.5, 0.04, 'Figure 1.1.4: Transformed dataset_3 Histogram'

# Adjust layout to prevent overlapping
plt.tight_layout()

# Save the figure as Figure 1.1.4
filename = os.path.join(folder_path, 'figure_1.1.4.png')
plt.savefig(filename)

# Display the plot
plt.show()
```





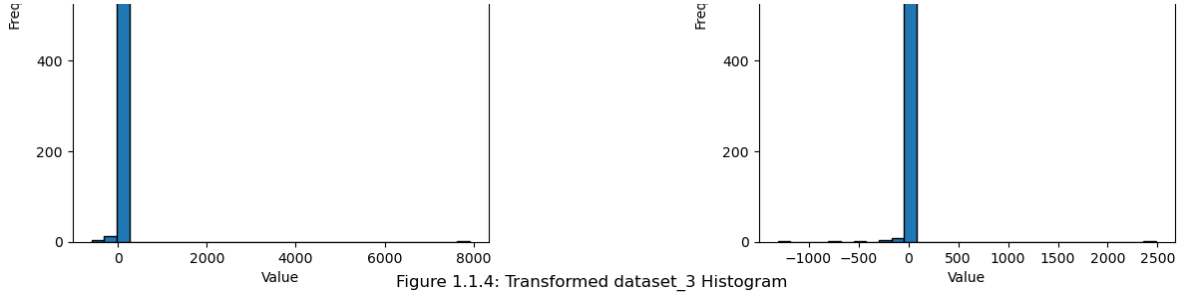


Figure 1.1.4: Transformed dataset_3 Histogram

HalfNormal Distribution: $\frac{2e^{-\frac{x^2\theta^2}{\pi}}\theta}{\pi}$

Lognormal Distribution: $\frac{e^{-\frac{(-\mu+\ln x)^2}{2\sigma^2}}}{\sqrt{2\pi}\cdot x\cdot\sigma}$

Lévy Distribution: $\frac{e^{-\frac{\sigma}{2(x-\mu)}}\left(\frac{\sigma}{x-\mu}\right)^{\frac{3}{2}}}{\sqrt{2\pi\sigma}}$

Exponential Distribution: $e^{-x\lambda}\lambda$

Pareto Distribution: $k^{\alpha}\cdot x^{-1-\alpha}\cdot\alpha$

Gamma Distribution: $\frac{e^{-\frac{x}{\beta}}x^{-1+\alpha}\beta^{-\alpha}}{\Gamma(\alpha)}$

Gumbel Distribution: $\frac{e^{-e^{\frac{x-\alpha}{\beta}}+\frac{x-\alpha}{\beta}}}{\beta}$

Cauchy Distribution: $\frac{1}{b\pi\left(1+\frac{(-a+x)^2}{b^2}\right)}$

In [94]:
```
def half_normal(x, theta):
    if not (np.all(x > 0) and np.all(theta > 0)):
        raise ValueError("Both x and theta must be greater than 0")
    return (2 * np.exp(-x**2 * theta**2 / np.pi) * theta) / np.pi

def lognormal(x, mu, sigma):
    if not (np.all(x > 0) and np.all(sigma > 0)):
        raise ValueError("Both x and sigma must be greater than 0")
    return (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))) / (np.sq

def levy(x, mu, sigma):
    if not (np.all(x > mu) and np.all(sigma > 0)):
        raise ValueError("x must be greater than mu, and sigma must
    return (np.exp(-sigma / (2 * (x - mu))) * (sigma / (x - mu))**(

def exponential(x, lam):
    if not np.all(lam > 0):
        raise ValueError("Lambda must be greater than 0")
    return np.exp(-x * lam) * lam

def pareto(x, k, alpha):
    if not (np.all(x >= k) and np.all(k > 0) and np.all(alpha > 0))
        raise ValueError("x must be greater than or equal to k, and
    return k**alpha * x**(-1 - alpha) * alpha

def gamma_distribution(x, alpha, beta):
    if not (np.all(x > 0) and np.all(alpha > 0) and np.all(beta > 0
        raise ValueError("x, alpha, and beta must all be greater th
    return (np.exp(-x / beta) * x**(alpha - 1) * beta**(-alpha)) /

def gumbel(x, alpha, beta):
    if not np.all(beta > 0):
        raise ValueError("Beta must be greater than 0")
    return np.exp(-np.exp((x - alpha) / beta) + (x - alpha) / beta)


def cauchy_distribution(x, a, b):
    if not np.all(b > 0):
        raise ValueError("Scale parameter (b) must be greater than
    return 1 / (np.pi * b * (1 + ((x - a) / b)**2))
```

In [95]:
```
epsilon = 1e-10

def nnlf_half_normal(params, data):
```

```
    theta = params[0]
    if not (np.all(data > 0) and np.all(theta > 0)):
        return np.inf
    half_normal_values = half_normal(data, theta)
    valid_values = np.isfinite(half_normal_values)
    log_likelihood = -np.sum(np.log(np.where(half_normal_values[val
    return log_likelihood

def nnlf_lognormal(params, data):
    mu, sigma = params
    if not (np.all(data > 0) and np.all(sigma > 0)):
        return np.inf
    lognormal_values = lognormal(data, mu, sigma)
    valid_values = np.isfinite(lognormal_values)
    log_likelihood = -np.sum(np.log(np.where(lognormal_values[valid
    return log_likelihood

def nnlf_levy(params, data):
    mu, sigma = params
    if not (np.all(data > mu) and np.all(sigma > 0)):
        return np.inf
    levy_values = levy(data, mu, sigma)
    valid_values = np.isfinite(levy_values)
    log_likelihood = -np.sum(np.log(np.where(levy_values[valid_valu
    return log_likelihood

def nnlf_exponential(params, data):
    lam = params[0]
    if not (np.all(lam > 0)):
        return np.inf
    exponential_values = exponential(data, lam)
    valid_values = np.isfinite(exponential_values)
    log_likelihood = -np.sum(np.log(np.where(exponential_values[val
    return log_likelihood

def nnlf_pareto(params, data):
    k, alpha = params
    if not (np.all(data >= k) and np.all(k > 0) and np.all(alpha >
        return np.inf
    pareto_values = pareto(data, k, alpha)
    valid_values = np.isfinite(pareto_values)
    log_likelihood = -np.sum(np.log(np.where(pareto_values[valid_va
    return log_likelihood

def nnlf_gamma(params, data):
    alpha, beta = params
    if not (np.all(data > 0) and np.all(alpha > 0) and np.all(beta
        return np.inf
    gamma_values = gamma_distribution(data, alpha, beta)
    valid_values = np.isfinite(gamma_values)
    log_likelihood = -np.sum(np.log(np.where(gamma_values[valid_val
    return log_likelihood

def nnlf_gumbel(params, data):
```

```
    alpha, beta = params
    if not np.all(beta > 0):
        return np.inf
    gumbel_values = gumbel(data, alpha, beta)
    valid_values = np.isfinite(gumbel_values)
    log_likelihood = -np.sum(np.log(np.where(gumbel_values[valid_va

    return log_likelihood

def nnlf_cauchy(params, data):
    a, b = params
    if not np.all(b > 0):
        return np.inf
    cauchy_values = cauchy_distribution(data, a, b)
    valid_values = np.isfinite(cauchy_values)
    log_likelihood = -np.sum(np.log(np.where(cauchy_values[valid_va
    return log_likelihood
```

In [96]:

```
def half_normal_cdf(x, theta):
    if not (np.all(x > 0) and np.all(theta > 0)):
        raise ValueError("Both x and theta must be greater than 0")
    integrand = lambda t: half_normal(t, theta)
    result, _ = quad(integrand, 0, x)
    return result

def lognormal_cdf(x, mu, sigma):
    if not (np.all(x > 0) and np.all(sigma > 0)):
        raise ValueError("Both x and sigma must be greater than 0")
    integrand = lambda t: lognormal(t, mu, sigma)
    result, _ = quad(integrand, 0, x)
    return result

def levy_cdf(x, mu, sigma):
    if not (np.all(x > mu) and np.all(sigma > 0)):
        raise ValueError("x must be greater than mu, and sigma must
    integrand = lambda t: levy(t, mu, sigma)
    result, _ = quad(integrand, mu+epsilon, x)
    return result

def exponential_cdf(x, lam):
    if not np.all(lam > 0):
```

```
    if not np.all(lam > 0):
        raise ValueError("Lambda must be greater than 0")
    integrand = lambda t: exponential(t, lam)
    result, _ = quad(integrand, 0, x)
    return result

def pareto_cdf(x, k, alpha):
    if not (np.all(x >= k) and np.all(k > 0) and np.all(alpha > 0))
        raise ValueError("x must be greater than or equal to k, and
    integrand = lambda t: pareto(t, k, alpha)
    result, _ = quad(integrand, k+epsilon, x)
    return result

def gamma_cdf(x, alpha, beta):
    if not (np.all(x > 0) and np.all(alpha > 0) and np.all(beta > 0
        raise ValueError("x, alpha, and beta must all be greater th
    integrand = lambda t: gamma_distribution(t, alpha, beta)
    result, _ = quad(integrand, 0, x)
    return result

def gumbel_cdf(x, alpha, beta):
    if not np.all(beta > 0):
        raise ValueError("Beta must be greater than 0")
    integrand = lambda t: gumbel(t, alpha, beta)
    result, _ = quad(integrand, -np.inf, x)
    return result

def cauchy_cdf(x, a, b):
    if not np.all(b > 0):
        raise ValueError("Scale parameter (b) must be greater than
    integrand = lambda t: cauchy_distribution(t, a, b)
    result, _ = quad(integrand, -np.inf, x)
    return result

half_normal_cdf = np.vectorize(half_normal_cdf)
lognormal_cdf = np.vectorize(lognormal_cdf)
levy_cdf = np.vectorize(levy_cdf)
exponential_cdf = np.vectorize(exponential_cdf)
pareto_cdf = np.vectorize(pareto_cdf)
gamma_cdf = np.vectorize(gamma_cdf)
gumbel_cdf = np.vectorize(gumbel_cdf)
cauchy_cdf = np.vectorize(cauchy_cdf)
```

In [97]:
```
def jackknife_mean(dataset):
    return sum(dataset)/len(dataset)

def jackknife(dataset, statistic_function):
    n = len(dataset)
    jackknife_estimates = []

    for i in range(n):
        leave_one_out_data = np.delete(dataset, i)
        statistic_value = statistic_function(leave_one_out_data)
        jackknife_estimates.append(statistic_value)


    return jackknife_estimates
```

In [98]:
```
def reject_or_not(ks_statistic, critical):
    if ks_statistic > critical:
        return "Reject the null hypothesis"
    else:
        return "Fail to reject the null hypothesis"
```

In [99]:
```
def inverse_yeojohnson(transformed_data, lambda_value):
    if lambda_value == 0:
        # Inverse for lambda = 0 (log transformation)
        inverse_transformed_data = np.exp(transformed_data) - 1
    else:
        # Inverse for other lambda values
        inverse_transformed_data = np.where(transformed_data < 0,
                                            -(np.abs(transformed_da
                                            (transformed_data * lam
    return inverse_transformed_data
```

In [100]:
```
# Half-Normal Distribution
bounds_half_normal = [(epsilon, 1000)]  # Replace np.inf with a lar

# Lognormal Distribution
bounds_lognormal = [(-1000, 1000), (epsilon, 1000)]  # Replace np.i

# Lévy Distribution
bounds_levy = [(-10000, min(dataset_3) - epsilon), (epsilon, 10000)

# Exponential Distribution
bounds_exponential = [(epsilon, 1000)]  # Replace np.inf with a lar
```

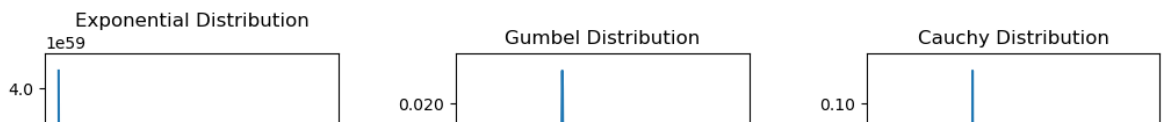
```
# Pareto Distribution
bounds_pareto = [(epsilon, min(dataset_3)), (epsilon, 1000)]  # Rep

# Gamma Distribution
bounds_gamma = [(epsilon, 1000), (epsilon, 1000)]  # Replace None a

# Gumbel Distribution
bounds_gumbel = [(-1000, 1000), (epsilon, 1000)]  # Replace np.inf

# Cauchy Distribution
bounds_cauchy = [(-1000, 1000), (epsilon, 1000)]  # Replace np.inf


# Define bounds for other distributions in a similar manner

# result_half_normal = dual_annealing(nnlf_half_normal, bounds=boun
# result_lognormal = dual_annealing(nnlf_lognormal, bounds=bounds_l
result_levy = dual_annealing(nnlf_levy, bounds=bounds_levy, args=(d
result_exponential = dual_annealing(nnlf_exponential, bounds=bounds
# result_pareto = dual_annealing(nnlf_pareto, bounds=bounds_pareto,
# result_gamma = dual_annealing(nnlf_gamma, bounds=bounds_gamma, ar
result_gumbel = dual_annealing(nnlf_gumbel, bounds=bounds_gumbel, a
result_cauchy = dual_annealing(nnlf_cauchy, bounds=bounds_cauchy, a

# Access the optimized parameters
# optimized_params_half_normal = result_half_normal.x
# optimized_params_lognormal = result_lognormal.x
optimized_params_levy = result_levy.x
optimized_params_exponential = result_exponential.x
# optimized_params_pareto = result_pareto.x
# optimized_params_gamma = result_gamma.x
optimized_params_gumbel = result_gumbel.x
optimized_params_cauchy = result_cauchy.x

# Print or use the optimized parameters as needed
# print("Optimized Parameters - Half-Normal:", optimized_params_hal
# print("Optimized Parameters - Lognormal:", optimized_params_logno
print("Optimized Parameters - Lévy:", optimized_params_levy)
print("Optimized Parameters - Exponential:", optimized_params_expon
# print("Optimized Parameters - Pareto:", optimized_params_pareto)
# print("Optimized Parameters - Gamma:", optimized_params_gamma)
print("Optimized Parameters - Gumbel:", optimized_params_gumbel)
print("Optimized Parameters - Cauchy:", optimized_params_cauchy)
```

```
/var/folders/rm/7h6vlsd94v9b6hq6m0qjz4680000gn/T/ipykernel_59884/4
1822365.py:19: RuntimeWarning: overflow encountered in exp
  return np.exp(-x * lam) * lam
/var/folders/rm/7h6vlsd94v9b6hq6m0qjz4680000gn/T/ipykernel_59884/4
1822365.py:19: RuntimeWarning: overflow encountered in multiply
  return np.exp(-x * lam) * lam
/var/folders/rm/7h6vlsd94v9b6hq6m0qjz4680000gn/T/ipykernel_59884/4
1822365.py:34: RuntimeWarning: overflow encountered in exp
  return np.exp(-np.exp((x - alpha) / beta) + (x - alpha) / beta)
/ beta

Optimized Parameters - Lévy: [-1308.21872687  1314.56037608]
Optimized Parameters - Exponential: [0.10665465]
Optimized Parameters - Gumbel: [15.24836289 16.61141675]
Optimized Parameters - Cauchy: [11.32126446  1.41966338]
```

In [101]:
```
x_values = np.linspace(min(dataset_3), max(dataset_3), len(dataset_


# Calculate values for each distribution
# half_normal_values = half_normal(x_values, optimized_params_half_
# lognormal_values = lognormal(x_values, optimized_params_lognormal
# levy_values = levy(x_values, optimized_params_levy[0], optimized_
exponential_values = exponential(x_values, optimized_params_exponen
# pareto_values = pareto(x_values, optimized_params_pareto[0], opti
# gamma_values = gamma_distribution(x_values, optimized_params_gamm
gumbel_values = gumbel(x_values, optimized_params_gumbel[0], optimi
cauchy_values = cauchy_distribution(x_values, optimized_params_cauc
```

In [134]:
```

```

```
import os
import numpy as np
import matplotlib.pyplot as plt

# Plot the PDFs on a single line
plt.figure(figsize=(12, 4))  # Adjust the figure size for a single

# First subplot
plt.subplot(1, 3, 1)
plt.plot(x_values, exponential_values, label='Exponential')
plt.hist(dataset_3, bins=30, density=True, color='gray', alpha=0.7,
plt.title('Exponential Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

# Second subplot
plt.subplot(1, 3, 2)
plt.plot(x_values, gumbel_values, label='Gumbel')
plt.hist(dataset_3, bins=30, density=True, color='gray', alpha=0.7,
plt.title('Gumbel Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

# Third subplot
plt.subplot(1, 3, 3)
plt.plot(x_values, cauchy_values, label='Cauchy')
plt.hist(dataset_3, bins=30, density=True, color='gray', alpha=0.7,
plt.title('Cauchy Distribution')
plt.xlabel('x')
plt.ylabel('PDF')

# Adjust layout to prevent overlapping with extra spacing
plt.subplots_adjust(left=0.1, bottom=0.1, right=0.9, top=0.9, wspac

plt.text(0.5, 0.02, 'Figure 1.3.3: Dataset_3 distribution PDFs', ha

# Save the figure
filename = os.path.join(folder_path, 'figure_1.3.3.png')
plt.savefig(filename)

plt.show()
```



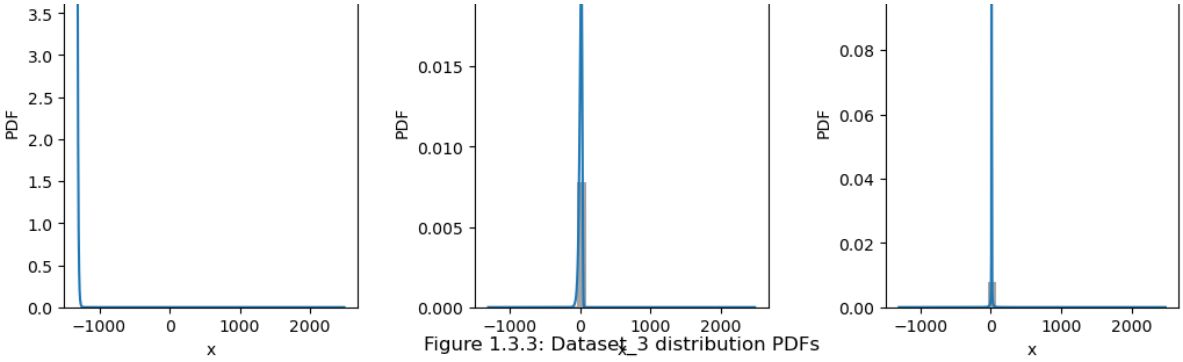


Figure 1.3.3: Dataset_3 distribution PDFs

In [140]:
```
# Create a QQ plot for each distribution against dataset_2
distributions = [ gumbel_values, cauchy_values]
distribution_names = ['Gumbel', 'Cauchy']

fig, axes = plt.subplots(1, 2, figsize=(8, 4))

for i, (dist_values, dist_name) in enumerate(zip(distributions, dis
    ax = axes[i]
    sm.qqplot_2samples(dist_values,dataset_3, ax=ax, line='45')
    ax.set_title(f'QQ Plot: {dist_name} vs. Dataset_3')
    ax.set_xlabel(f'Theoretical Quantiles ({dist_name})')
    ax.set_ylabel(f'Quantiles of Dataset_3')

plt.tight_layout()
plt.text(0.5, 0.01, 'Figure 1.3.6: Dataset_3 distribution Q-Q plots

# Save the figure
filename = os.path.join(folder_path, 'figure_1.3.6.png')
plt.savefig(filename)

plt.show()
```

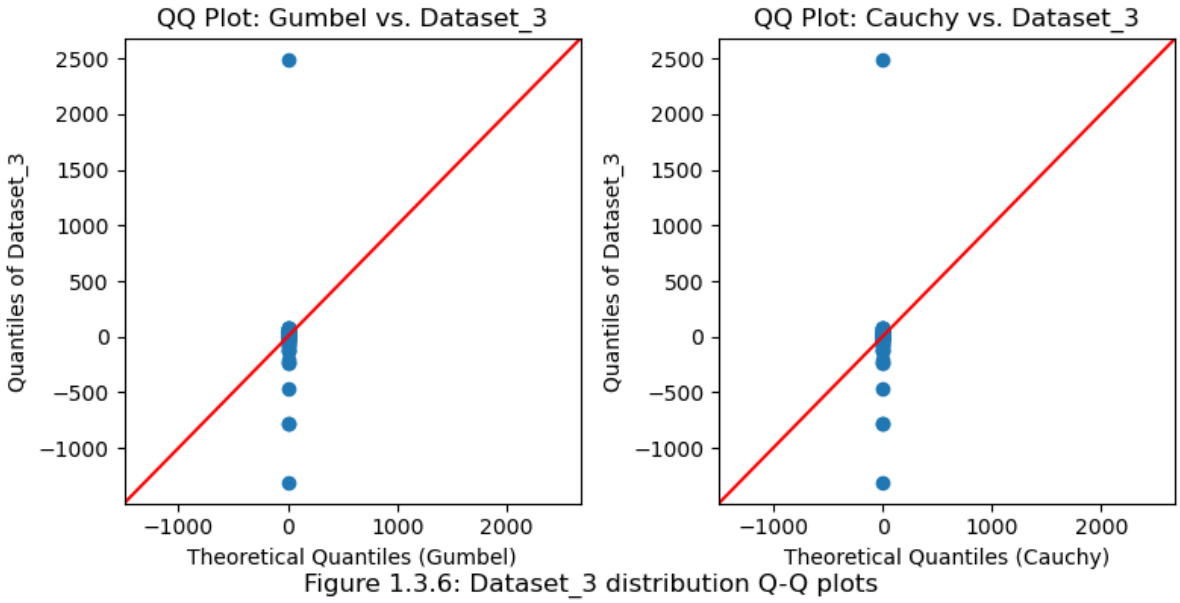


Figure 1.3.6: Dataset_3 distribution Q-Q plots

In [63]:

```
critical_value = np.sqrt(-np.log(0.05/2)/ (2*len(dataset_3)))

print("KS-Test Critical-Value:",critical_value)
print("\n")

# Pareto distribution
ks_statistic, ks_p_value = kstest(dataset_3, lambda x: exponential_
print("KS Test for Exponential:", ks_statistic, ks_p_value)
print("Rejection Note:", reject_or_not(ks_statistic, critical_value
print("\n")

# Gamma distribution
ks_statistic, ks_p_value = kstest(dataset_3, lambda x: gumbel_cdf(x
print("KS Test for Gumbel:", ks_statistic, ks_p_value)
print("Rejection Note:", reject_or_not(ks_statistic, critical_value
print("\n")


# Cauchy distribution


ks_statistic, ks_p_value = kstest(dataset_3, lambda x: cauchy_cdf(x
print("KS Test for Cauchy:", ks_statistic, ks_p_value)
print("Rejection Note:", reject_or_not(ks_statistic, critical_value
print("\n")
```

```
KS-Test Critical-Value: 0.04165485256596809


KS Test for Exponential: 3.9452213089794665e+60 0.0
Rejection Note: Reject the null hypothesis


KS Test for Gumbel: 1.0 0.0
Rejection Note: Reject the null hypothesis


KS Test for Cauchy: 0.014816877397403228 0.9710668163257185
Rejection Note: Fail to reject the null hypothesis



/var/folders/rm/7h6vlsd94v9b6hq6m0qjz4680000gn/T/ipykernel_59884/4
028956293.py:54: IntegrationWarning: The algorithm does not conver
ge.  Roundoff error is detected
  in the extrapolation table.  It is assumed that the requested to
lerance
```

```
  cannot be achieved, and that the returned result (if full_output
= 1) is
  the best which can be obtained.
  result, _ = quad(integrand, -np.inf, x)
```

In [16]:

```
aic_cauchy = 2 * nnlf_cauchy(optimized_params_cauchy, dataset_3) +
bic_cauchy = 2 * nnlf_cauchy(optimized_params_cauchy, dataset_3) +
print("Cauchy AIC score:", aic_cauchy)
print("Cauchy BIC score:", bic_cauchy)
print("\n")
```

```
Cauchy AIC score: 6188.460920906726
Cauchy BIC score: 6198.398621663409
```

In [113]:

```
from tqdm.notebook import tqdm as tqdm # Import tqdm for notebooks

# Bootstrapping
n_iterations = 10000
bootstrap_a = []
bootstrap_b = []

dataset_3 = np.genfromtxt('data3.csv', delimiter=',')


for _ in tqdm(range(n_iterations), desc="Bootstrapping"):
    bootstrap_sample = np.random.choice(dataset_3, size=len(dataset
    bounds_cauchy = [(-100, 100), (epsilon, 100)]  # Replace None a
    values = dual_annealing(nnlf_cauchy, bounds=bounds_cauchy, args
#     values = values.x
#     inverse_transformed_parameters_1 = inverse_yeojohnson(values,
#     inverse_transformed_parameters_2 = inverse_yeojohnson(inverse

    bootstrap_a.append(values.x[0])
    bootstrap_b.append(values.x[1])

# Calculate confidence intervals
confidence_intervals_a = np.percentile(bootstrap_a, [2.5, 97.5], ax
confidence_intervals_b = np.percentile(bootstrap_b, [2.5, 97.5], ax
```

```
Bootstrapping:   0%|          | 0/10000 [00:00<?, ?it/s]
```

In [114]:
```
mean_a = np.mean(bootstrap_a)
mean_b = np.mean(bootstrap_b)
print("Bootstrapped Estimate a:", mean_a)
print("Bootstrapped Estimates b:", mean_b)
print("\n")

print("95% Confidence Intervals a:", confidence_intervals_a)
print("95% Confidence Intervals b:", confidence_intervals_b)
print("\n")
```

```
Bootstrapped Estimate a: 14.97778933634648
Bootstrapped Estimates b: 2.126549413229624


95% Confidence Intervals a: [14.79914677 15.15402815]
95% Confidence Intervals b: [1.944477   2.31576855]
```

In [115]:
```
bootstrap_standard_error_a = 0
for i in range(n_iterations):
    bootstrap_standard_error_a = bootstrap_standard_error_a + (boot
bootstrap_standard_error_a = np.sqrt(bootstrap_standard_error_a/(n_



bootstrap_standard_error_b = 0
for i in range(n_iterations):
    bootstrap_standard_error_b = bootstrap_standard_error_b + (boot
bootstrap_standard_error_b = np.sqrt(bootstrap_standard_error_b/(n_

# 1.96 is z-value for 95% confidence interval
print("PARAMETER(A):")
print("-------------------------------------------")

lower_CI_a = mean_a - 1.96 * bootstrap_standard_error_a
upper_CI_a = mean_a + 1.96 * bootstrap_standard_error_a
print("Percentile 95% bounds for Parameter(alpha)")
print(lower_CI_a)
print(upper_CI_a)
print("\n")

bootstrap_bias_a = 0
for i in range(len(bootstrap_a)):
    bootstrap_bias_a = bootstrap_bias_a + (mean_a - bootstrap_a[i])

bootstrap_bias_a = bootstrap_bias_a / len(bootstrap_a)
p_0_a = np.count_nonzero(bootstrap_a >= mean_a)/len(bootstrap_a)

print("P_0 for Parameter(a)")
print(p_0_a)
```

```
print("\n")
print("Bootstrap bias for Parameter(a)")
print(bootstrap_bias_a)
print("\n")


z_0_a = scipy.stats.norm.ppf(p_0_a)

print("Z_0 for Parameter(a)")
print(z_0_a)
print("\n")


bc_lower_CI_a = np.percentile(bootstrap_a, 100 * scipy.stats.norm.c
bc_upper_CI_a = np.percentile(bootstrap_a, 100 * scipy.stats.norm.c

print("Bias corrected 95% bounds for Parameter(a)")
print(bc_lower_CI_a)
print(bc_upper_CI_a)
print("\n")

estimates_a = jackknife(bootstrap_a,jackknife_mean)
jackknife_estimate_a = sum(jackknife(bootstrap_a,jackknife_mean)) /

standard_error_rate_a = 0
se_up_a = 0
se_down_a = 0

for i in range(len(bootstrap_a)):
    se_up_a = se_up_a + (estimates_a[i] - jackknife_estimate_a)**3
    se_down_a = se_down_a + (estimates_a[i] - jackknife_estimate_a)

standard_error_rate_a = (1/6) * (se_up_a / se_down_a**1.5)

print("Rate of Change of Standard Error Parameter(a) (Jackknife acc
print("\n")

bca_up_upper_a = z_0_a + scipy.stats.norm.ppf(0.05/2)
bca_down_upper_a = 1- standard_error_rate_a*(z_0_a +scipy.stats.nor
bca_up_lower_a = z_0_a + scipy.stats.norm.ppf(1 - 0.05/2)
bca_down_lower_a = 1- standard_error_rate_a*(z_0_a +scipy.stats.nor
bca_upper_CI_a = np.percentile(bootstrap_a, 100*scipy.stats.norm.cd
bca_lower_CI_a = np.percentile(bootstrap_a, 100*scipy.stats.norm.cd

print("Bias corrected accelerated 95% bounds for Parameter(a)")
print(bca_upper_CI_a)
print(bca_lower_CI_a)
print("\n")

print("PARAMETER(B):")
print("-------------------------------------------")

lower_CI_b = mean_b - 1.96 * bootstrap_standard_error_b
upper_CI_b = mean_b + 1.96 * bootstrap_standard_error_b
```

```
print("Percentile 95% bounds for Parameter(B)")
print(lower_CI_b)
print(upper_CI_b)
print("\n")

bootstrap_bias_b = 0
for i in range(len(bootstrap_b)):
    bootstrap_bias_b = bootstrap_bias_b + (mean_b - bootstrap_b[i])

bootstrap_bias_b = bootstrap_bias_b / len(bootstrap_b)
p_0_b = np.count_nonzero(bootstrap_b >= mean_b)/len(bootstrap_b)

print("P_0 for Parameter(b)")
print(p_0_b)
print("\n")
print("Bootstrap bias for Parameter(b)")
print(bootstrap_bias_b)
print("\n")


z_0_b = scipy.stats.norm.ppf(p_0_b)

print("Z_0 for Parameter(b)")
print(z_0_b)
print("\n")


bc_lower_CI_b = np.percentile(bootstrap_b, 100 * scipy.stats.norm.c
bc_upper_CI_b = np.percentile(bootstrap_b, 100 * scipy.stats.norm.c

print("Bias corrected 95% bounds for Parameter(b)")
print(bc_lower_CI_b)
print(bc_upper_CI_b)
print("\n")

estimates_b = jackknife(bootstrap_b,jackknife_mean)
jackknife_estimate_b = sum(jackknife(bootstrap_b,jackknife_mean)) /

standard_error_rate_b = 0
se_up_b = 0
se_down_b = 0

for i in range(len(bootstrap_b)):
    se_up_b = se_up_b + (estimates_b[i] - jackknife_estimate_b)**3
    se_down_b = se_down_b + (estimates_b[i] - jackknife_estimate_b)

standard_error_rate_b = (1/6) * (se_up_b / se_down_b**1.5)

print("Rate of Change of Standard Error Parameter(b) (Jackknife acc
print("\n")

bca_up_upper_b = z_0_b + scipy.stats.norm.ppf(0.05/2)
bca_down_upper_b = 1- standard_error_rate_b*(z_0_b +scipy.stats.nor
bca_up_lower_b = z_0_b + scipy.stats.norm.ppf(1 - 0.05/2)
```

```
bca_down_lower_b = 1- standard_error_rate_b*(z_0_b +scipy.stats.nor
bca_upper_CI_b = np.percentile(bootstrap_b, 100*scipy.stats.norm.cd
bca_lower_CI_b = np.percentile(bootstrap_b, 100*scipy.stats.norm.cd

print("Bias corrected accelerated 95% bounds for Parameter(b)")
print(bca_upper_CI_b)
print(bca_lower_CI_b)
print("\n")
```

```
PARAMETER(A):
_____________________________________________
Percentile 95% bounds for Parameter(alpha)
14.802172502374905
15.153406170318053


P_0 for Parameter(a)
0.4998


Bootstrap bias for Parameter(a)
1.0579981335467892e-15


Z_0 for Parameter(a)
-0.0005013256759256267
```

```
Bias corrected 95% bounds for Parameter(a)
14.798990400814215
15.153968358756535


Rate of Change of Standard Error Parameter(a) (Jackknife accelerat
ed value): 2.297686994269811e-06


Bias corrected accelerated 95% bounds for Parameter(a)
14.79899177671727
15.153968884305412



PARAMETER(B):
_____________________________________________
Percentile 95% bounds for Parameter(B)
1.9389855923218136
2.3141132341374346


P_0 for Parameter(b)
0.491


Bootstrap bias for Parameter(b)
4.85944617878431e-16


Z_0 for Parameter(b)
-0.0225615639022472


Bias corrected 95% bounds for Parameter(b)
1.9414446944135781
2.3108269349518302


Rate of Change of Standard Error Parameter(b) (Jackknife accelerat
ed value): -0.00024626445028361427


Bias corrected accelerated 95% bounds for Parameter(b)
1.9412124233154406
2.310795974758799
```

In [141]:
```
import matplotlib.pyplot as plt

# Bootstrap results for Parameter(a)
print("Bootstrap mean a:", mean_a)
```

```
print("Bootstrap mean a:", mean_a)
print("Bootstrap standard error a:", bootstrap_standard_error_a)

# Plot for Parameter(a)
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.hist(bootstrap_a, bins=60, edgecolor='black', alpha=0.75)
plt.axvline(mean_a, color='blue', linestyle='dashed', linewidth=2,
plt.axvline(lower_CI_a, color='red', linestyle='dashed', linewidth=
plt.axvline(upper_CI_a, color='red', linestyle='dashed', linewidth=
plt.axvline(bc_lower_CI_a, color='green', linestyle='dashed', linew
plt.axvline(bc_upper_CI_a, color='green', linestyle='dashed', linew
plt.axvline(bca_lower_CI_a, color='cyan', linestyle='dashed', linew
plt.axvline(bca_upper_CI_a, color='cyan', linestyle='dashed', linew
plt.title('Bootstrap Distribution of Parameter(a) for Cauchy')
plt.xlabel('Parameter(a) Coefficient')
plt.ylabel('Frequency')
plt.legend(loc='upper left', bbox_to_anchor=(0, 1), fontsize='small

# Bootstrap results for Parameter(b)
print("Bootstrap mean b:", mean_b)
print("Bootstrap standard error b:", bootstrap_standard_error_b)

# Plot for Parameter(b)
plt.subplot(1, 2, 2)
plt.hist(bootstrap_b, bins=60, edgecolor='black', alpha=0.75)
plt.axvline(mean_b, color='blue', linestyle='dashed', linewidth=2,
plt.axvline(lower_CI_b, color='red', linestyle='dashed', linewidth=
plt.axvline(upper_CI_b, color='red', linestyle='dashed', linewidth=
plt.axvline(bc_lower_CI_b, color='green', linestyle='dashed', linew
plt.axvline(bc_upper_CI_b, color='green', linestyle='dashed', linew
plt.axvline(bca_lower_CI_b, color='cyan', linestyle='dashed', linew
plt.axvline(bca_upper_CI_b, color='cyan', linestyle='dashed', linew
plt.title('Bootstrap Distribution of Parameter(b) for Cauchy')
plt.xlabel('Parameter(b) Coefficient')
plt.ylabel('Frequency')
plt.legend(loc='upper left', bbox_to_anchor=(0, 1), fontsize='small

plt.text(0.5, 0.01, 'Figure 1.5.3: Dataset_3 Cauchy Distribution Pa

# Save the figure
filename = os.path.join(folder_path, 'figure_1.5.3.png')
plt.savefig(filename)

plt.show()
```

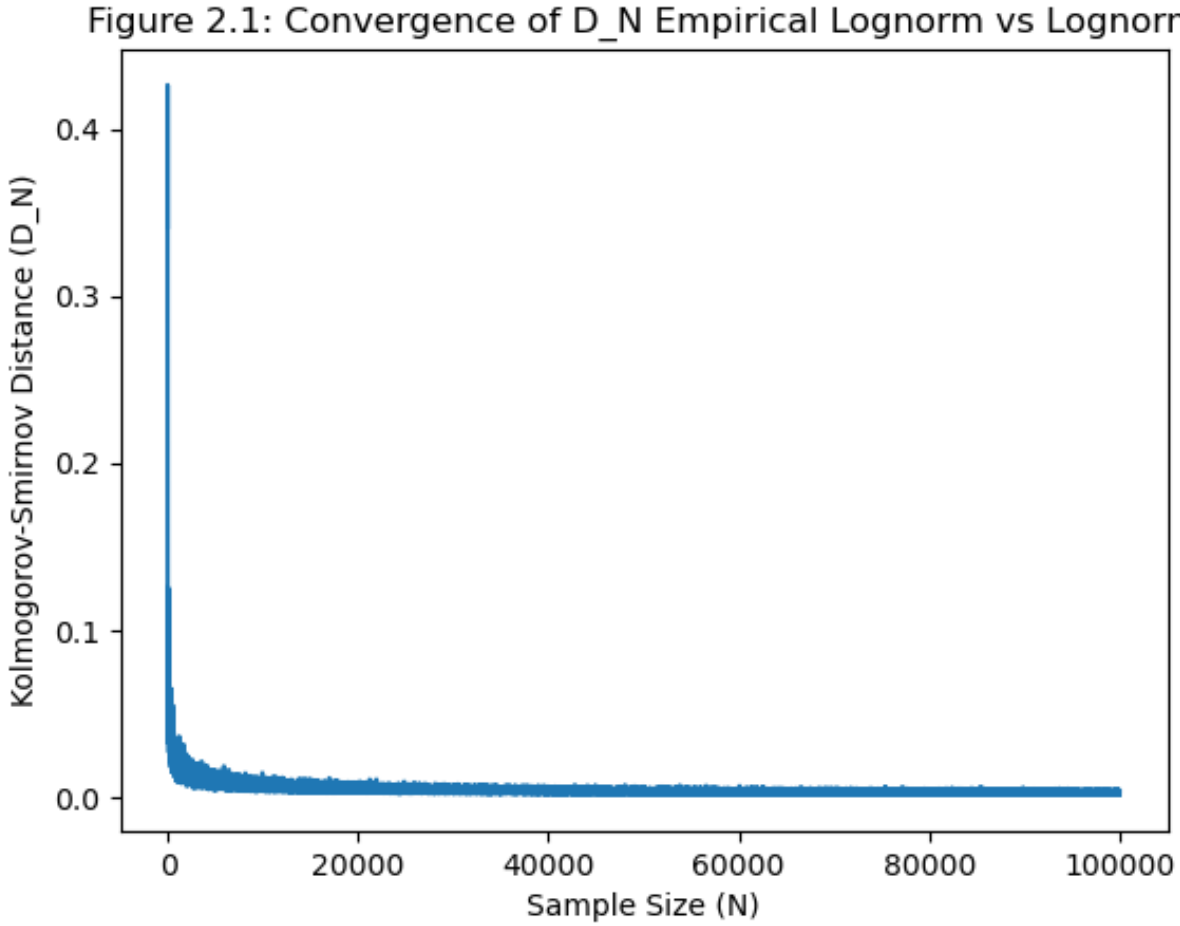
Section 2: Glivenko Cantelli Theorem

In [383…
```
import math
import numpy as np
import matplotlib.pyplot as plt
import scipy
import csv
import pandas as pd
import seaborn as sns
import random
import statsmodels.api as sm
```

In [5]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import pareto, cauchy, lognorm
from tqdm.notebook import tqdm as tqdm # Import tqdm for notebooks


# Function to generate hypothetical true distribution CDF
def true_distribution_cdf(x, true_distribution, bootstrap_sample):
    if true_distribution == 'pareto':
        return pareto.cdf(x, pareto_alpha_true, scale=pareto_x_m_true)
    elif true_distribution == 'cauchy':
        return cauchy.cdf(x, loc=np.median(bootstrap_sample), scale=np.pe
    elif true_distribution == 'lognorm':
        lognorm_mean = np.exp(np.mean(np.log(bootstrap_sample)))
        lognorm_std = np.std(np.log(bootstrap_sample))
        return lognorm.cdf(x, lognorm_std, loc=0, scale=np.exp(np.mean(np
    else:
        raise ValueError("Invalid true distribution specified.")

# Function to calculate empirical distribution function (EDF)
def empirical_distribution(data):
    sorted_data = np.sort(data)
    n = len(data)
    edf = np.arange(1, n + 1) / n
    return sorted_data, edf

# Function to compute D_N
def compute_d_n(data, true_distribution, alpha_true=None, x_m_true=None):
    sorted_data, edf_empirical = empirical_distribution(data)

    # Calculate the true distribution function (CDF)
    true_cdf_values = true_distribution_cdf(sorted_data, true_distributio

    # Compute the difference D_N
    d_n = np.max(np.abs(edf_empirical - true_cdf_values))

    return d_n

# Perform Bootstrap resampling with decreasing sample sizes
def bootstrap_resampling(original_data, true_distribution, num_resamples)
    results = []
    for N in tqdm(range(len(original_data), 0, -1), desc="Calculating D_N
        resampled_data = np.random.choice(original_data, size=N, replace=
        distance = compute_d_n(resampled_data, true_distribution)
        results.append((N, distance))
```

```
    return results

# Generate a large dataset from the true distribution (replace with your
large_sample_size = 100000
large_data = lognorm.rvs(size=large_sample_size, s=1)

# Perform Bootstrap resampling with decreasing sample sizes
bootstrap_results = bootstrap_resampling(large_data, true_distribution='l

# Extract results for plotting
sample_sizes, d_n_values = zip(*bootstrap_results)

# Plotting Fig 2.1
plt.figure()
plt.plot(sample_sizes, d_n_values)
plt.xlabel('Sample Size (N)')
plt.ylabel('Kolmogorov-Smirnov Distance (D_N)')
plt.title('Figure 2.1: Convergence of D_N Empirical Lognorm vs Lognorm')
plt.savefig('Figure_2.1.png')
plt.show()

# Perform Bootstrap resampling with decreasing sample sizes
bootstrap_results_pareto = bootstrap_resampling(large_data, true_distribu

# Extract results for plotting
sample_sizes_pareto, d_n_values_pareto = zip(*bootstrap_results_pareto)

# Plotting Fig 2.2
plt.figure()
plt.plot(sample_sizes_pareto, d_n_values_pareto)
plt.xlabel('Sample Size (N)')
plt.ylabel('Kolmogorov-Smirnov Distance (D_N)')
plt.title('Figure 2.2: Convergence of D_N Empirical Lognorm vs Pareto ')
plt.savefig('Figure_2.2.png')
plt.show()
```

```
Calculating D_N:   0%|          | 0/100000 [00:00<?, ?it/s]
```




```
Calculating D_N:   0%|          | 0/100000 [00:00<?, ?it/s]
```

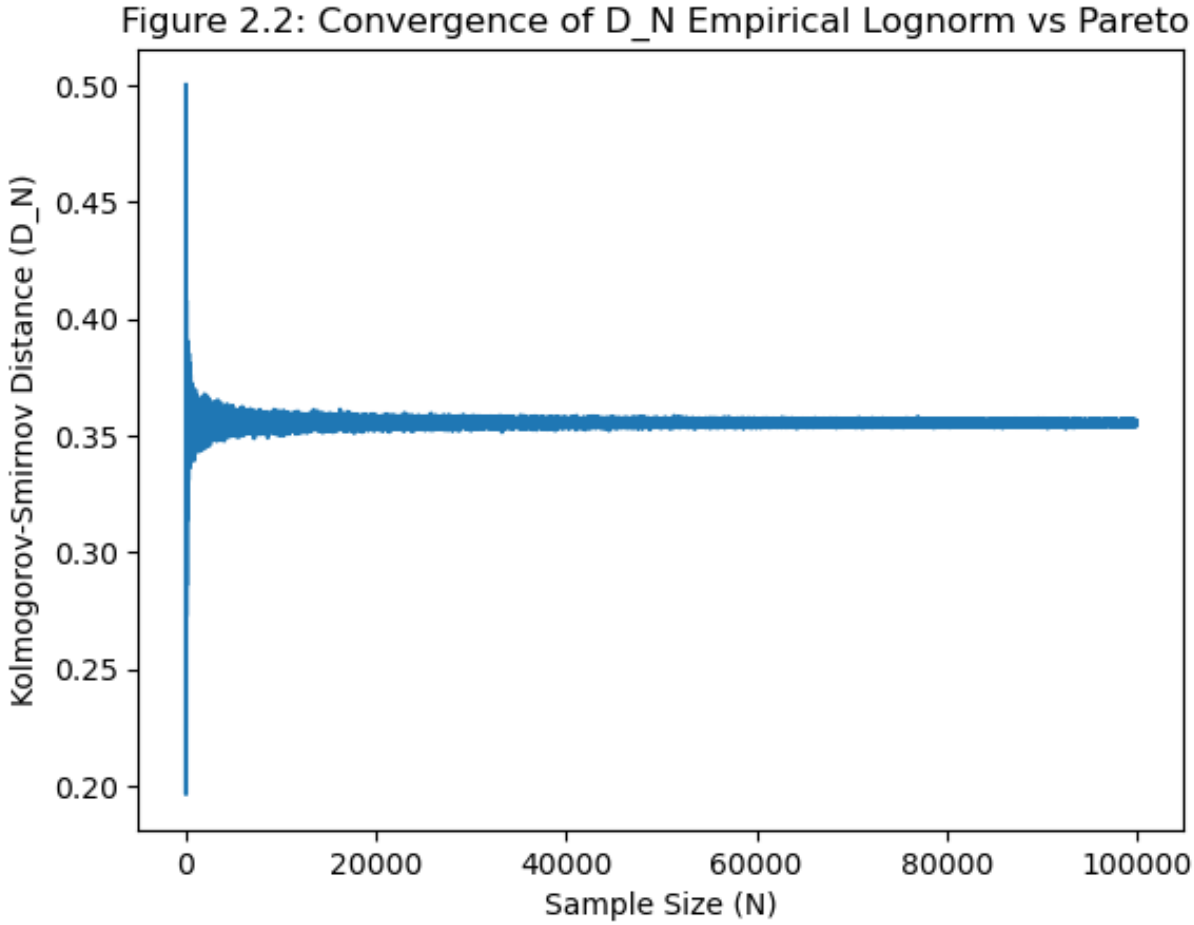


Figure 2.2: Convergence of D_N Empirical Lognorm vs Pareto
Kolmogorov-Smirnov Distance (D_N)
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0
20000
40000
60000
80000
100000
Sample Size (N)

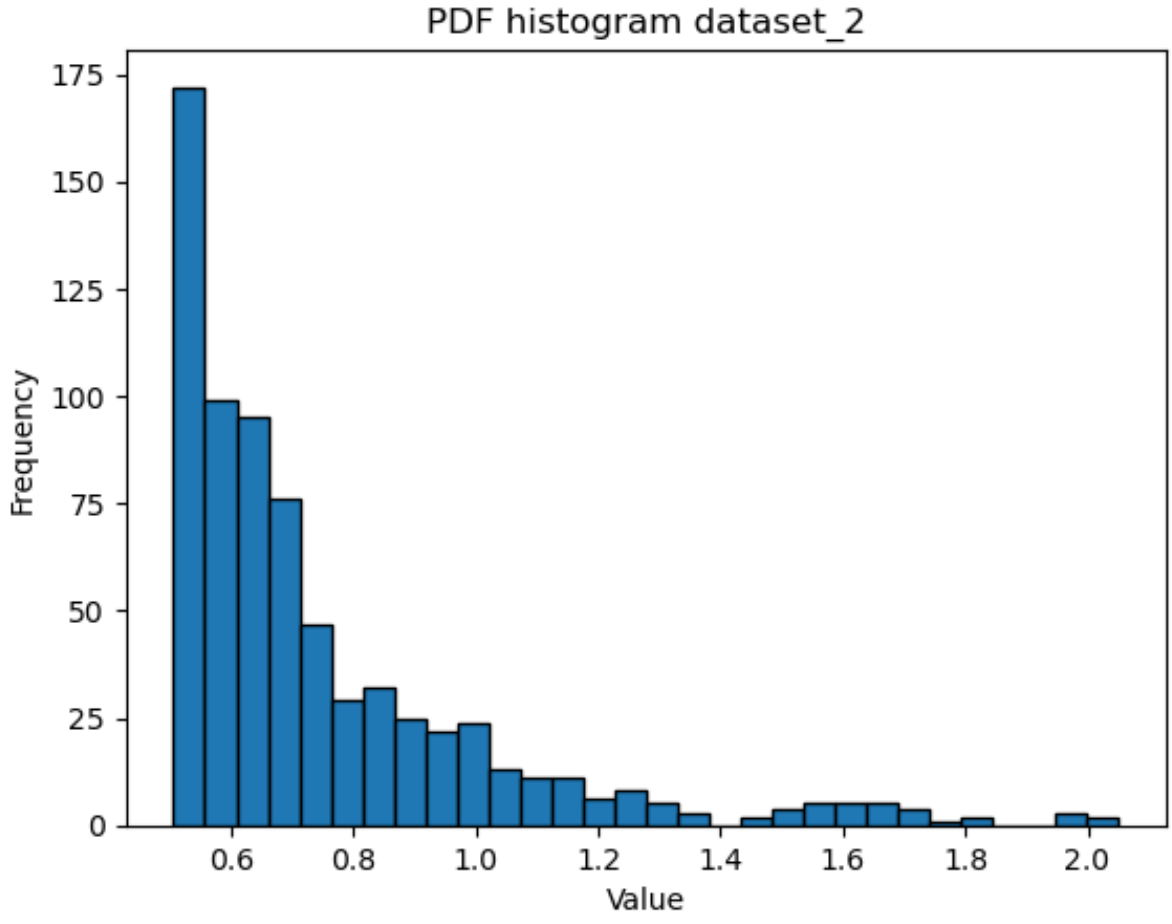
In [ ]:

Section 3: PBMC method

In [17]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
import scipy
import csv
import pandas as pd
import seaborn as sns
import random
import statsmodels.api as sm
```

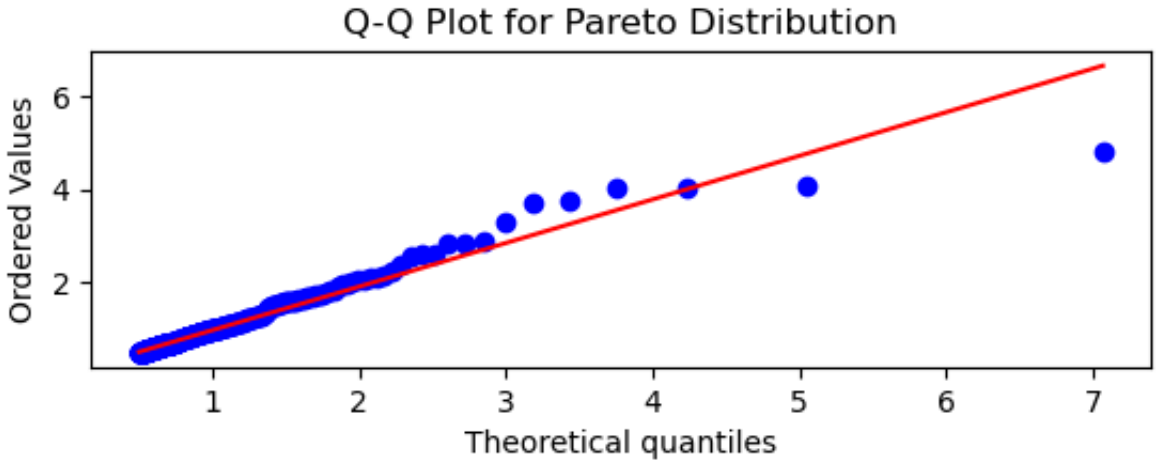
In [18]:
```
dataset_2 = np.genfromtxt('data2.csv', delimiter=',')
```

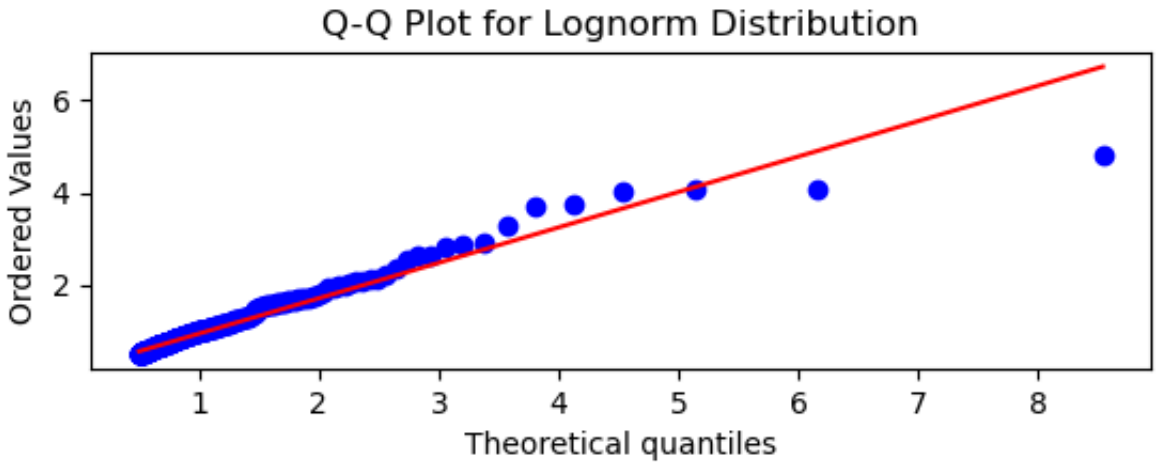
In [19]:
```
plt.hist(dataset_2,bins = 30,range = (np.percentile(dataset_2,2.5),
plt.title('PDF histogram dataset_2')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```



Model A is Log-Normal distribution and Model B is Pareto Distribution

In [20]:
```
# 4. Exponential Distribution
plt.subplot(2,1,1)
params_pareto = scipy.stats.pareto.fit(dataset_2)
scipy.stats.probplot(dataset_2, plot=plt, dist=scipy.stats.pareto,
plt.title("Q-Q Plot for Pareto Distribution")

# 4. Exponential Distribution
plt.subplot(2,1,2)
params_lognorm = scipy.stats.lognorm.fit(dataset_2)
scipy.stats.probplot(dataset_2, plot=plt, dist=scipy.stats.lognorm,
plt.title("Q-Q Plot for Lognorm Distribution")
plt.subplots_adjust(hspace=0.5, wspace= 0.5)
```





In [21]:
```
print("OBSERVED VALUES:")
print("--------------------")
print("\n")
print("Model-A fitted params:", params_lognorm)
print("Model-B fitted params:", params_pareto)
print("\n")
gof_a, _ = scipy.stats.kstest(dataset_2, scipy.stats.lognorm.cdf, a
gof_b, _ = scipy.stats.kstest(dataset_2, scipy.stats.pareto.cdf, ar
print("KS Test GOF for Model_A:", gof_a)
print("KS Test GOF for Model_B:", gof_b)
print("\n")
delta_gof = gof_a - gof_b
print("KS Test GOF Difference:", delta_gof)
```

```
OBSERVED VALUES:
--------------------


Model-A fitted params: (1.2940487780100147, 0.4964765599896324, 0.
14316397612742932)
Model-B fitted params: (2.6428441849944346, -0.002142523079998293,
0.5030034742638931)


KS Test GOF for Model_A: 0.04173500583705403
KS Test GOF for Model_B: 0.019748586122813716


KS Test GOF Difference: 0.021986419714240313
```

In [22]:
```
num_bootstrap_samples = 1000

# Array to store bootstrap sample statistics
bootstrap_params_a = []
bootstrap_params_b = []


for _ in range(num_bootstrap_samples):
    # Generate a bootstrap sample by sampling without replacement
    bootstrap_sample = np.random.choice(dataset_2, size=len(dataset

    # Calculate the statistic of interest for the bootstrap sample
    params_a = scipy.stats.lognorm.fit(bootstrap_sample)
    params_b = scipy.stats.pareto.fit(bootstrap_sample)

    # Store the statistic for each bootstrap sample
    bootstrap_params_a.append(params_a)

    # Store the statistic for each bootstrap sample
    bootstrap_params_b.append(params_b)
```

In [23]:
```
bootstrap_samples_a = []
bootstrap_samples_b = []
for i in range(num_bootstrap_samples):
    # For normal distribution
    generated_data_a = scipy.stats.lognorm.rvs(*bootstrap_params_a[
    bootstrap_samples_a.append(generated_data_a)

    # For exponential distribution
    generated_data_b = scipy.stats.pareto.rvs(*bootstrap_params_b[i
    bootstrap_samples_b.append(generated_data_b)
```

In [24]:
```
bootstrap_samples_params_a = []
bootstrap_samples_params_b = []

"A True"
for i in range(num_bootstrap_samples):
    generated_params_a = scipy.stats.lognorm.fit(bootstrap_samples_
    bootstrap_samples_params_a.append(generated_params_a)

    generated_params_b = scipy.stats.pareto.fit(bootstrap_samples_a
    bootstrap_samples_params_b.append(generated_params_b)

bootstrap_samples_statistics_a = []
bootstrap_samples_statistics_b = []
bootstrap_delta_gof_a = []

for i in range(num_bootstrap_samples):
    ks_samples_statistic_a, _ = scipy.stats.kstest(bootstrap_sample
    bootstrap_samples_statistics_a.append(ks_samples_statistic_a)

    ks_samples_statistic_b, _ = scipy.stats.kstest(bootstrap_sample
    bootstrap_samples_statistics_b.append(ks_samples_statistic_b)

    gof = ks_samples_statistic_a - ks_samples_statistic_b
    bootstrap_delta_gof_a.append(gof)
```

In [25]:
```
bootstrap_samples_params_a = []
bootstrap_samples_params_b = []

"B True"

for i in range(num_bootstrap_samples):
    generated_params_a = scipy.stats.lognorm.fit(bootstrap_samples_
    bootstrap_samples_params_a.append(generated_params_a)

    generated_params_b = scipy.stats.pareto.fit(bootstrap_samples_b
    bootstrap_samples_params_b.append(generated_params_b)

bootstrap_samples_statistics_a = []
bootstrap_samples_statistics_b = []
bootstrap_delta_gof_b = []

for i in range(num_bootstrap_samples):
    ks_samples_statistic_a, _ = scipy.stats.kstest(bootstrap_sample
    bootstrap_samples_statistics_a.append(ks_samples_statistic_a)

    ks_samples_statistic_b, _ = scipy.stats.kstest(bootstrap_sample
    bootstrap_samples_statistics_b.append(ks_samples_statistic_b)

    gof = ks_samples_statistic_a - ks_samples_statistic_b
    bootstrap_delta_gof_b.append(gof)
```

In [26]:
```
import numpy as np
import os
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from tabulate import tabulate  # Ensure that you have the 'tabulate
folder_path = '/Users/biligee/Documents/Classwork/semester_4/Stats/

# Assuming you have the data for the histograms or probability dist
# Replace these with your actual data
hist_a, edges_a = np.histogram(bootstrap_delta_gof_a, bins=30, dens
hist_b, edges_b = np.histogram(bootstrap_delta_gof_b, bins=30, dens

# Find the intersection point where histograms have the same height
max_height = min(np.max(hist_a), np.max(hist_b))
index_optimal_a = np.argmax(hist_a >= max_height)
index_optimal_b = np.argmax(hist_b >= max_height)

# Set the optimal criterion as the average of the x-values at the i
optimal_criterion = (edges_a[index_optimal_a] + edges_b[index_optim

# Plotting the histograms
plt.figure(figsize=(10, 6))  # Adjust the figure size as needed
```

```
plt.hist(bootstrap_delta_gof_a, bins=30, alpha=0.5, label='A true(L
plt.hist(bootstrap_delta_gof_b, bins=30, alpha=0.5, label='B true(P

# Mark the optimal criterion with a vertical line
plt.axvline(optimal_criterion, color='green', linestyle='dashed', l
plt.axvline(delta_gof, color='red', linestyle='dashed', linewidth=2

# Adding labels and title
plt.xlabel('Difference in GOF')
plt.ylabel('Probability Density')
plt.title('Difference Distributions and Optimal Criterion')


plt.text(0.5, 0.01, 'Figure 3.1: GOF difference distribution', ha='
plt.legend()

# Save the figure
filename = os.path.join(folder_path, '3.1.png')
plt.savefig(filename)
plt.show()
```

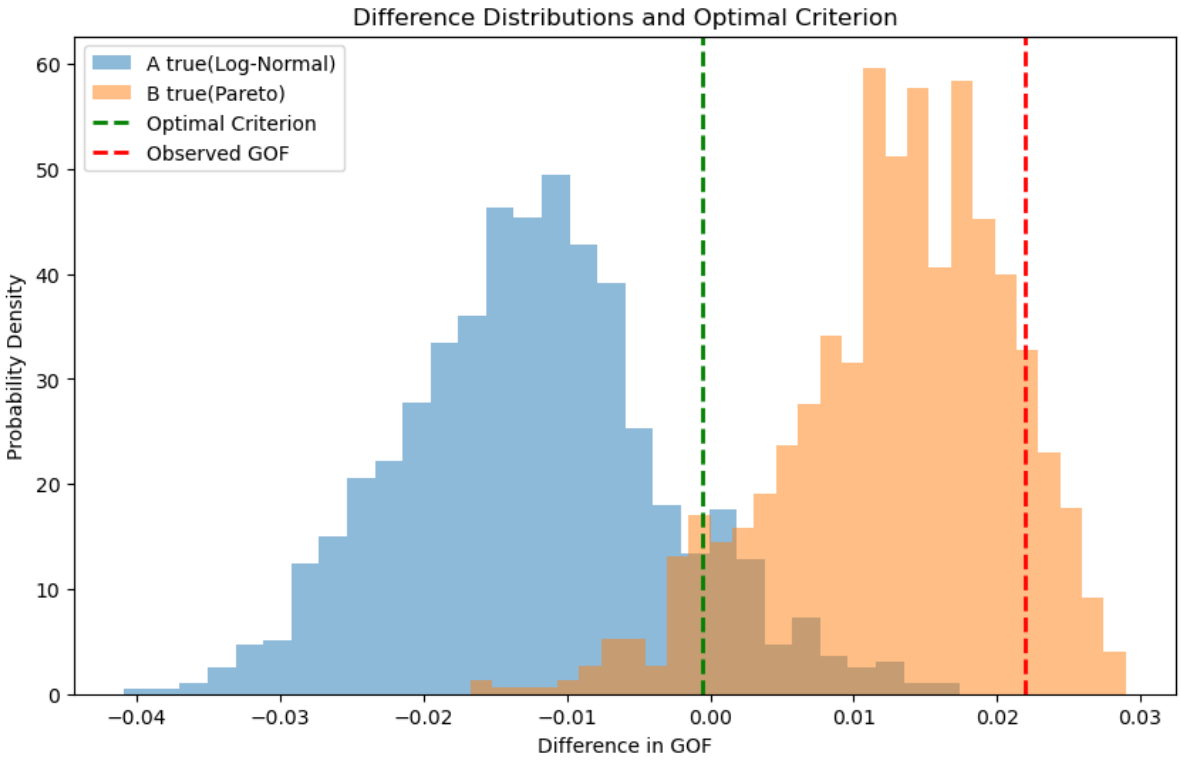


Figure 3.1: GOF difference distribution

In [ ]: