Member-only story

4 Ways to Quantify Fat Tails with Python

Intuition and Example Code

Shaw Talebi

Published in Towards Data Science · 11 min read · Dec 7, 2023

A fat (cat's) tail. Image from Canva.

This is the third article in a series on Power Laws and Fat Tails. In the previous post, we explored how to detect power laws from empirical data. While this technique can be handy, fat tails go beyond simply fitting data to a power law distribution. In this article, I will break down 4 ways we can quantify fat tails and share example Python code analyzing real-world data.

Note: If you are unfamiliar with terms like Power Law distribution or Fat Tail, review this article as a primer.

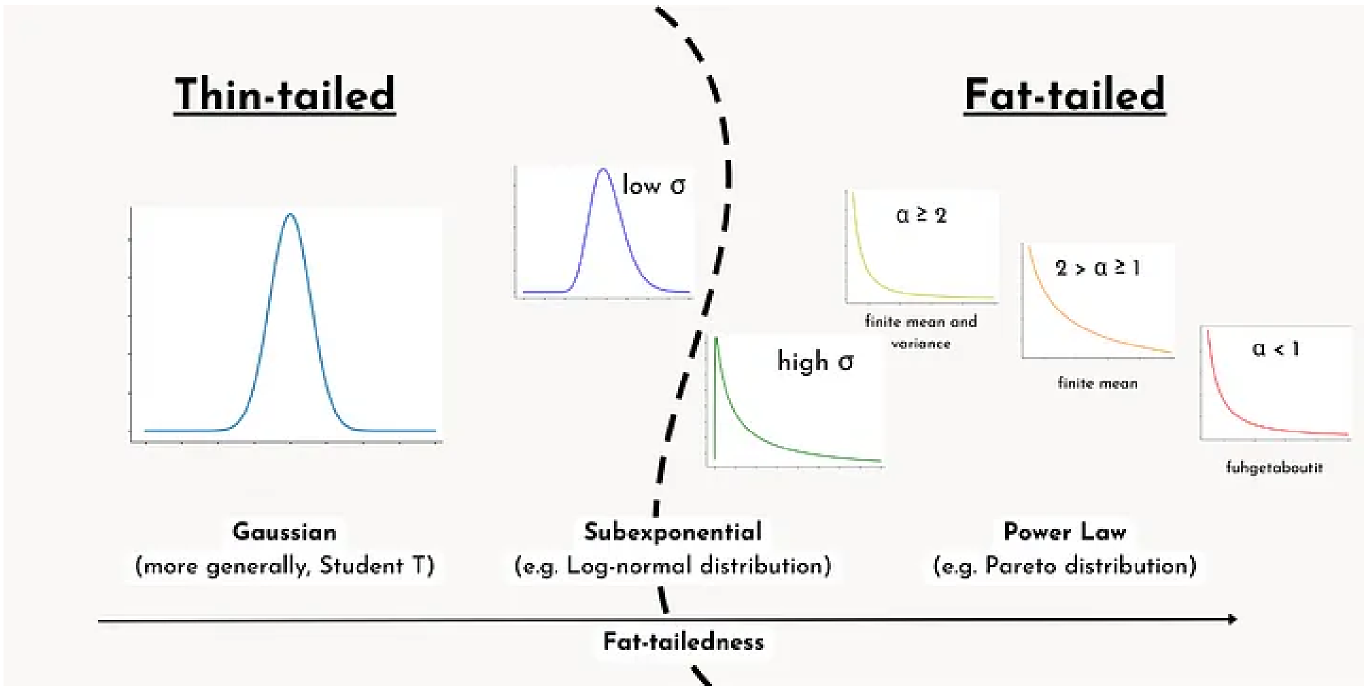
In the first article of this series, we introduced the idea of **fat tails,** which describes the **degree to which rare events drive the aggregate statistics of a distribution**. We saw an extreme example of fat tails via the Pareto distribution where, for example, 80% of sales are generated by 20% of customers (and 50% of sales are generated by just 1% of customers).

Pareto, Power Laws, and Fat Tails

What they don't teach you in statistics

towardsdatascience.com

Although Pareto (and more generally power law) distributions give us a salient example of fat tails, this is a more general notion that lives on a spectrum ranging from thin-tailed (i.e. a Gaussian) to very fat-tailed (i.e. Pareto 80–20).



The spectrum of Fat-tailedness. Image by author.

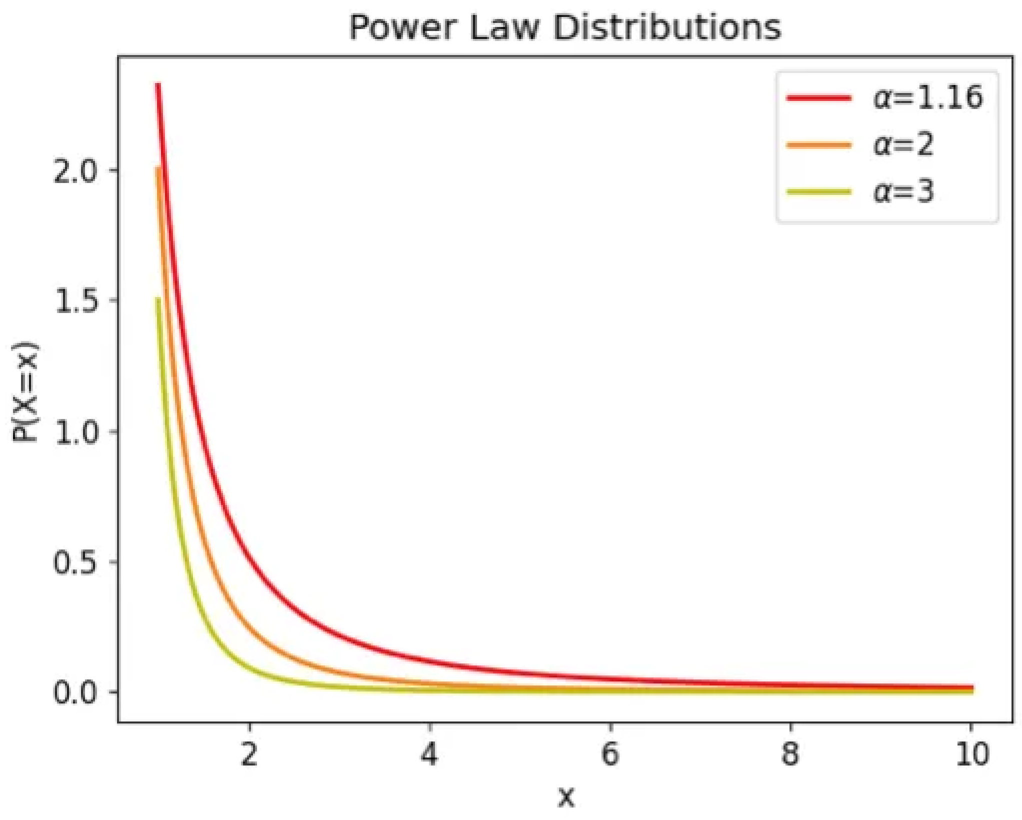
This view of *fat-tailedness* provides us with a more flexible and precise way of categorizing data than simply labeling it as a Power Law (or not). However, this begs the question: *how do we define fat-tailedness?*

4 Ways to Quantify Fat Tails

While there is no “true” measure of fat-tailedness, there are a few heuristics (i.e. rules of thumb) we can use in practice to quantify *how fat-tailed* data are. Here, we review 4 such heuristics. We start by introducing each technique conceptually and then dive into example Python code.

Heuristic 1: Power Law Tail Index

The *fattest* fat tails appear in Power Law distributions, where **the smaller a Power Law’s tail index (i.e. α), the fatter its tail**, as illustrated in the image below.



$$PDF(x) = Ax^{-(\alpha-1)}$$

Where, $x > x_{min}$

Example Power Law distributions with various α values. Image by author.

This observation that **smaller tail indexes imply fatter tails** naturally motivates us to use α to quantify fat tails. In practice, this boils down to fitting a power law distribution to a given dataset and extracting the estimated α value.

While this is a straightforward approach, it has one obvious limitation. Namely, the approach will break down when working with data that poorly fits a power law.

Heuristic 2: Kurtosis (i.e. non-Gaussianity)

The opposite of a fat tail is a **thin tail** (i.e. **rare events are so rare they are negligible**). A thin-tailed exemplar is the beloved Gaussian distribution, where the probability of an event 6 sigma away from the mean is about 1 in a billion.

This inspires another measure of fat tails by quantifying how "UN-Gaussian" the data are. We can do this via so-called non-Gaussianity measures. While we could devise many such measures, the most popular is **Kurtosis**, defined by the expression below.

$$Kurt[X] = \frac{\mu_4}{\mu_2^2}$$

$$\text{Where,} \quad \mu_k = \frac{1}{N}\sum_{i=1}^{N}(x_i - c)^k$$

$N =$ number of observations

$x_i = i^{th}$ observation

$c =$ center of moment

Definition of Kurtosis according to ref [1] and [2]. Image by author.

Kurtosis is driven by values far from the center (i.e. the tails). Thus, **the larger the kurtosis, the fatter the tail**.

This measure tends to work well when all the moments are finite [3]. One major limitation, however, is that Kurtosis is not defined for some distributions, e.g. Pareto with α =< 4, which makes it useless for many fat-tailed data.

Heuristic 3: Log-normal's σ

In past articles of this series, we discussed the Log-normal distribution, defined by the probability density function below.

Log-normal Distribution

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

The probability density function of log-normal distribution [4]. Image by author.

As we saw before, this distribution is a bit *mischievous* because it can appear Gaussian-like for low **σ**, yet Pareto-like at high **σ**. This naturally provides another way to quantify fat tails, where **the larger the σ, the fatter the tail**.

We can obtain this measure in a similar way as *Heuristic 1*. Namely, we fit a log-normal distribution to our data and extract the fit's σ value. While this is a simple procedure, it (like *Heuristic 1*) breaks down when the log-normal fit does not explain the underlying data well.

Heuristic 4: Taleb's κ

The preceding heuristics (H) started with a particular distribution in mind (i.e. H1 — Power Law and H3 — Log-normal). In practice, however, our data rarely precisely follow any particular distribution.

Moreover, comparisons using these measures may be problematic when evaluating 2 variables following qualitatively different distributions. For instance, using a power law's tail index to compare Pareto-like and Gaussian-like data may have little significance since a power law will poorly fit into Gaussian-like data.

This motivates the use of non-distribution-specific measures of fat-tailedness. One such measure was proposed by Taleb in ref [3]. The proposed **metric (κ)** is defined for **unimodal data with finite mean and takes values between 0 and 1**, where **0** indicates data are **maximally thin-tailed** and **1** implies the data are **maximally fat-tailed**. It is defined according to the expression below.

Taleb's κ Metric

$$\kappa(n_0, n) = 2 - \frac{\log n - \log n_0}{\log \frac{M(n)}{M(n_0)}}$$

Definition of Taleb's κ metric [3]. Image by author.

The metric compares two samples (say, S_{n_0} and S_n) where S_n is the sum of *n* samples drawn from a particular distribution. For example, if we evaluate a Gaussian distribution and choose *n*=100, we would draw 100 samples from a Gaussian and sum them all together to create S_{100}.

M(n) in the above expression denotes the **mean absolute deviation**, defined according to the equation below. This measure of the **dispersion around the mean** tends to be more robust than the standard deviation [3][5].

Mean Absolute Deviation

$$M(n) \equiv M(S_n) = \mathbb{E}(|S_n - \mathbb{E}(S_n)|)$$

Where, $\mathbb{E}() =$ expectation value

$S_n = \Sigma_{i=1}^{n} s_i$

$s_i =$ sample from some distribution

Definition of mean absolute deviation from κ equation [3]. Image by author.

To simplify things, we can choose $n_0=1$, giving us the expression below.

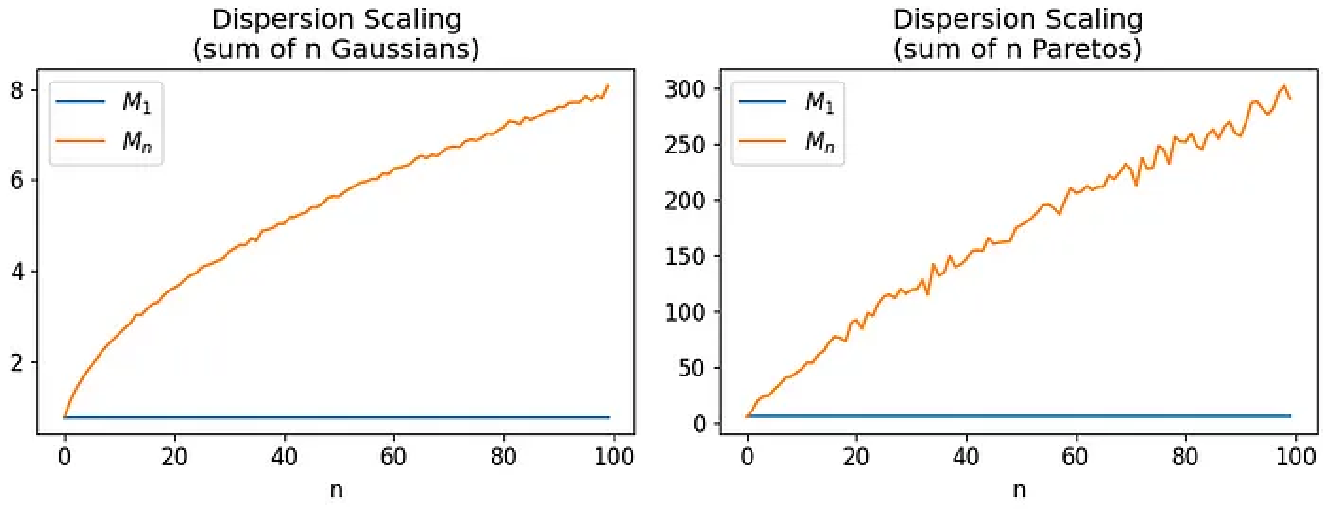
$$\kappa(1,n) = 2 - \frac{\log n}{\log \frac{M(n)}{M(1)}}$$

κ with $n_0=1$. Image by author.

The key term here is *M(n)/M(1)*, where ***M(n)*** **quantifies the dispersion around the mean for the sum of n samples** (of some distribution).

For **thin-tailed** distributions, *M(30)* will be relatively close to *M(1)* since the data generally sit close to the mean. Thus, *M(30)/M(1) ~ 1.*

For **fat-tailed** data, however, *M(30)* will be much larger than *M(1)*. Thus, *M(30)/M(1) >> 1*. This is illustrated below, where the left plot shows how dispersion scales for a sum of Gaussians, and the right plot shows how it scales for a Pareto.



Scaling of M(n) and M(1) for Gaussian (left) and Pareto 80–20 (right) distributions. Notice the y-axis labels. Note: the scale of Gaussian dispersion increases due to the summing of n distributions. Image by author.

Thus, for fat-tailed data, the denominator in the κ equation will be bigger than the numerator, making the second term on the RHS smaller and, ultimately, κ larger.

If this was all more math than you bargained for, here's the takeaway:
Big κ = fat-tailed, small κ = thin-tailed.

Example Code: Quantifying the Fat-tailedness of (Real-world) Social Media Data

With the conceptual stuff out of the way, let's see what using these heuristics looks like in practice. Here, we will use each approach described above to analyze the same data from the previous article of this series.

The data are from my social media accounts, which include monthly followers gained on **Medium**, earnings per **YouTube** video, and daily impressions on **LinkedIn**. The data and code are freely available at the GitHub repo.

YouTube-Blog/power-laws/3-quantifying-fat-tails at main · ShawhinT/YouTube-Blog

Codes to complement YouTube videos and blog posts on Medium. - YouTube-Blog/power-laws/3-quantifying-fat-tails at main...

github.com

We start by importing some helpful libraries.

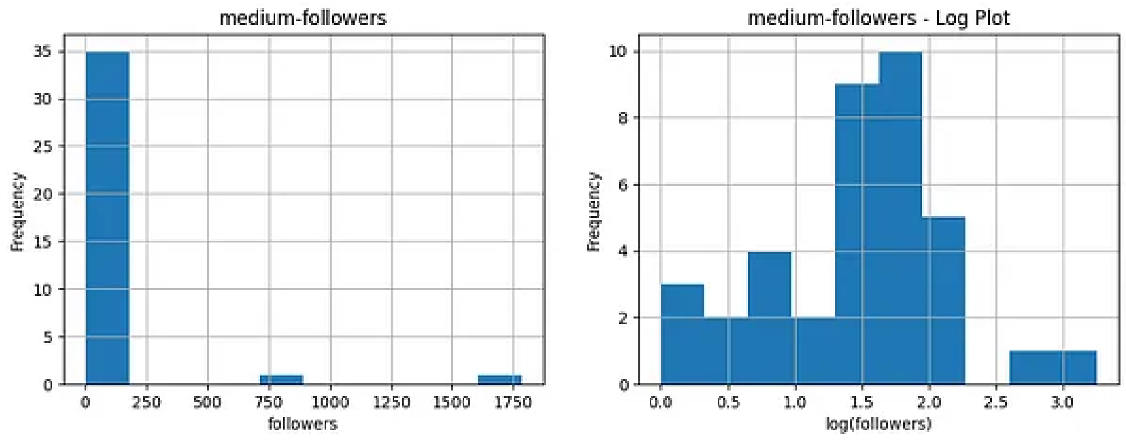
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import powerlaw
from scipy.stats import kurtosis
```

Next, we will load each dataset and store them in a dictionary.

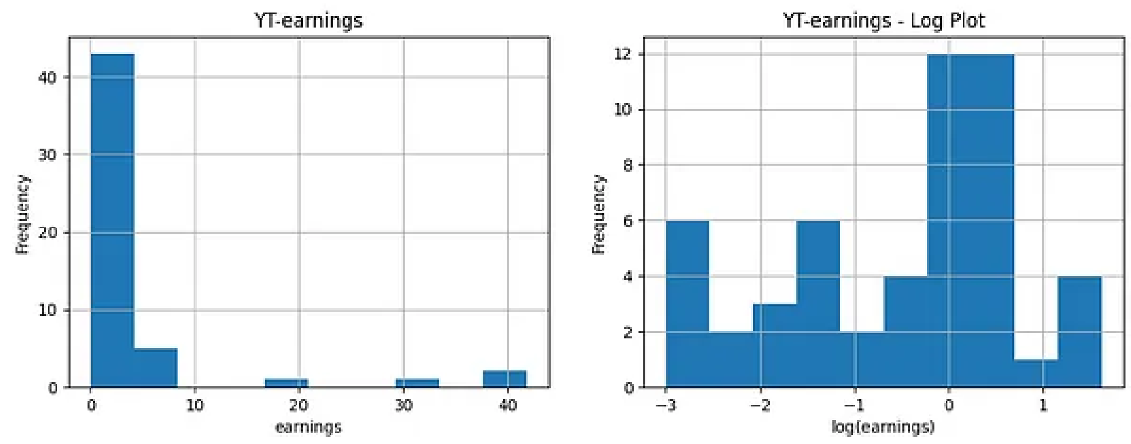
```
filename_list = ['medium-followers', 'YT-earnings', 'LI-impressions']

df_dict = {}

for filename in filename_list:
    df = pd.read_csv('data/'+filename+'.csv')
    df = df.set_index(df.columns[0]) # set index
    df_dict[filename] = df
```

At this point, looking at the data is always a good idea. We can do that by plotting histograms and printing the top 5 records for each dataset.

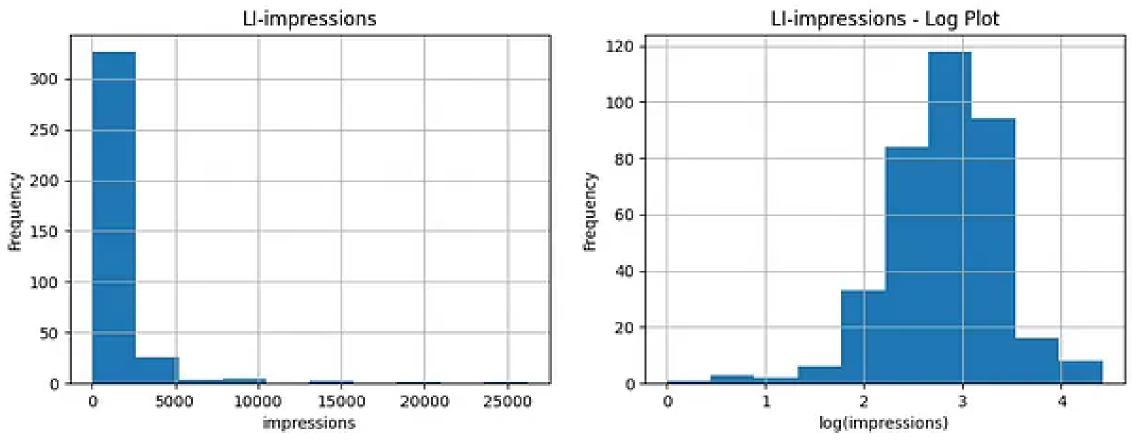
```
for filename in filename_list:
    df = df_dict[filename]

    # plot histograms (function bleow is defined in notebook on GitHub)
    plot_histograms(df.iloc[:,0][df.iloc[:,0]>0], filename, filename.split('-')[
    plt.savefig("images/"+filename+"_histograms.png")

    # print top 5 records
    print("Top 5 Records by Percentage")
    print((df.iloc[:,0]/df.iloc[:,0].sum()).sort_values(ascending=False)[:5])
    print("")
```



Histograms for monthly Medium followers. Image by author.

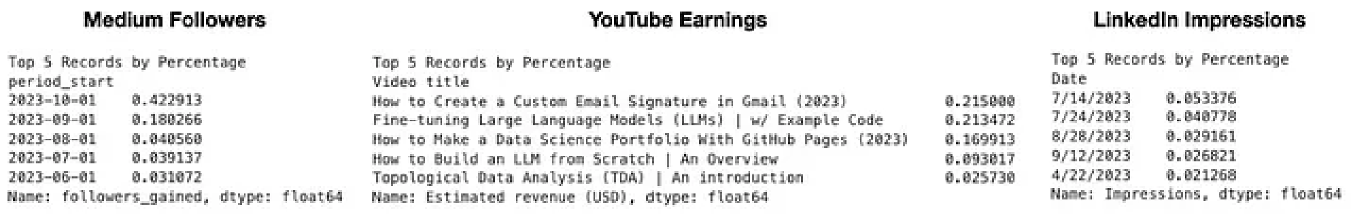


Histograms for YouTube video earnings. Note: if you notice a difference from the previous article, it's because I found a rogue record in the data (that's why looking is a good idea 😅). Image by author.



Histograms for daily LinkedIn impressions. Image by author.

Based on the histograms above, each dataset appears fat-tailed to some extent. Let's see the top 5 records by percentage to get another look at this.



Medium Followers

```
Top 5 Records by Percentage
period_start
2023-10-01    0.422913
2023-09-01    0.180266
2023-08-01    0.040560
2023-07-01    0.039137
2023-06-01    0.031072
Name: followers_gained, dtype: float64
```

YouTube Earnings

```
Top 5 Records by Percentage
Video title
How to Create a Custom Email Signature in Gmail (2023)              0.215000
Fine-tuning Large Language Models (LLMs) | w/ Example Code          0.213472
How to Make a Data Science Portfolio With GitHub Pages (2023)       0.169913
How to Build an LLM from Scratch | An Overview                      0.093017
Topological Data Analysis (TDA) | An introduction                   0.025730
Name: Estimated revenue (USD), dtype: float64
```

LinkedIn Impressions

```
Top 5 Records by Percentage
Date
7/14/2023     0.053376
7/24/2023     0.040778
8/28/2023     0.029161
9/12/2023     0.026821
4/22/2023     0.021268
Name: Impressions, dtype: float64
```

Top 5 records by percentage for each dataset. Image by author.

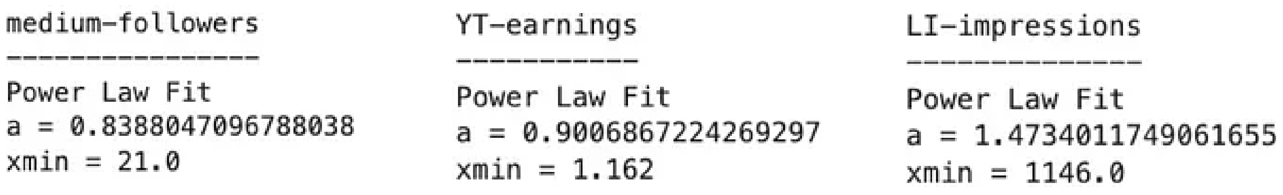
From this view, Medium followers appear the most fat-tailed, with 60% of followers coming from just 2 months. YouTube earnings are also strongly fat-tailed, where about 60% of revenue comes from just 4 videos. LinkedIn impressions seem the least fat-tailed.

While we may get a qualitative sense of the fat-tailedness just by *looking at the data*, let’s make this more quantitative via our 4 heuristics.

Heuristic 1: Power Law Tail Index

To obtain an *α* for each dataset, we can use the *powerlaw* library as we did in the previous article. This is done in the code block below, where we perform the fit and print the parameter estimates for each dataset in a for loop.

```
for filename in filename_list:
    df = df_dict[filename]

    # perform Power Law fit
    results = powerlaw.Fit(df.iloc[:,0])

    # print results
    print("")
    print(filename)
    print("-"*len(filename))
    print("Power Law Fit")
    print("a = " + str(results.power_law.alpha-1))
    print("xmin = " + str(results.power_law.xmin))
    print("")
```



```
medium-followers          YT-earnings               LI-impressions
----------------          -----------               --------------
Power Law Fit             Power Law Fit             Power Law Fit
a = 0.8388047096788038    a = 0.9006867224269297    a = 1.4734011749061655
xmin = 21.0               xmin = 1.162              xmin = 1146.0
```

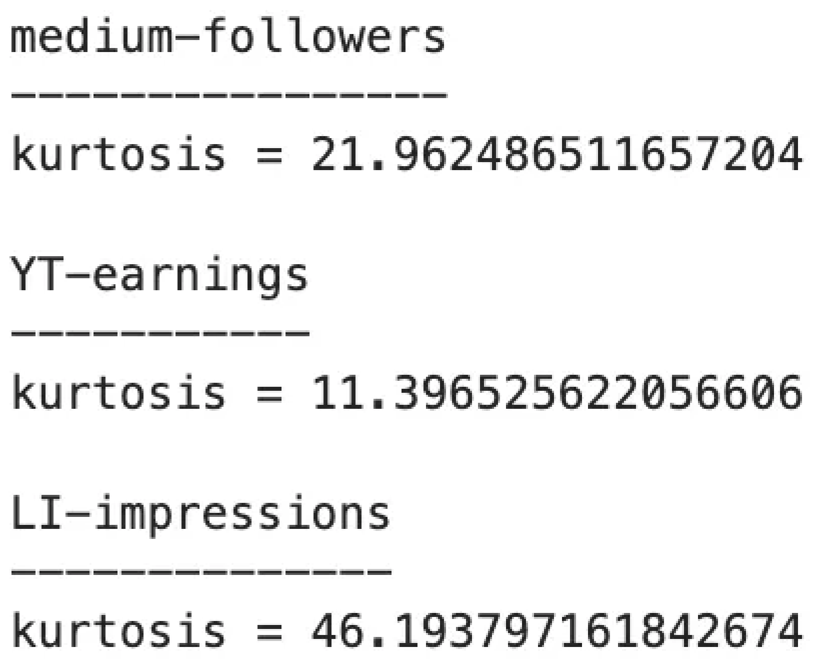
Power Law fit results. Image by author.

The results above match our qualitative assessment that Medium followers are the most fat-tailed, followed by YouTube earnings and LinkedIn impressions (remember, a smaller α means a fatter tail).

Heuristic 2: Kurtosis

An easy way to compute Kurtosis is using an off-the-shelf implementation. Here, I use Scipy and print the results in a similar way as before.

```
for filename in filename_list:
    df = df_dict[filename]

    # print results
    print(filename)
    print("-"*len(filename))
    print("kurtosis = " + str(kurtosis(df.iloc[:,0], fisher=True)))
    print("")
```



```
medium-followers
----------------
kurtosis = 21.962486511657204

YT-earnings
-----------
kurtosis = 11.39652562205660б

LI-impressions
--------------
kurtosis = 46.193797161842674
```

Kurtosis values for each dataset. Image by author.

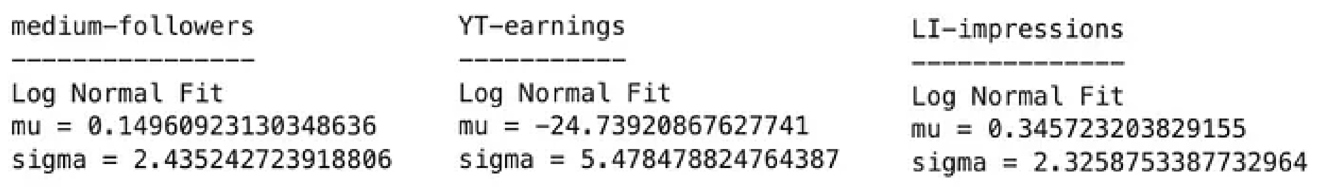
Kurtosis tells us a different story than *Heuristic 1*. The ranking of fat-tailedness according to this measure is as follows: LinkedIn > Medium > YouTube.

However, these results should be taken with a grain of salt. As we saw with the power law fits above, all 3 datasets fit a power law with $\alpha < 4$, meaning the Kurtosis is infinite. So, while the computation returns a value, it's probably wise to be suspicious of these numbers.

Heuristic 3: Log-normal's σ

We can again use the powerlaw library to obtain σ estimates similar to what we did for *Heuristic 1*. Here's what that looks like.

```
for filename in filename_list:
    df = df_dict[filename]

    # perform Power Law fit
    results = powerlaw.Fit(df.iloc[:,0])

    # print results
    print("")
    print(filename)
    print("-"*len(filename))
    print("Log Normal Fit")
    print("mu = " + str(results.lognormal.mu))
    print("sigma = " + str(results.lognormal.sigma))
    print("")
```



```
medium-followers
----------------
Log Normal Fit
mu = 0.14960923130348636
sigma = 2.435242723918806
```

```
YT-earnings
-----------
Log Normal Fit
mu = -24.73920867627741
sigma = 5.478478824764387
```

```
LI-impressions
--------------
Log Normal Fit
mu = 0.345723203829155
sigma = 2.3258753387732964
```

Log-normal fit results. Image by author.

Looking at the σ values above, we see all fits imply the data are fat-tailed, where Medium followers and LinkedIn impressions have similar σ estimates. YouTube earnings, on the other hand, have a significantly larger σ value, implying a (much) fatter tail.

One cause for suspicion, however, is that the fit estimates a negative μ, which may suggest a Log-normal fit may not explain the data well.

Heuristic 4: Taleb's κ

Since I couldn't find an off-the-shelf Python implementation for computing κ (I didn't look very hard), this computation requires a few extra steps. Namely, we need to define 3 helper functions, as shown below.

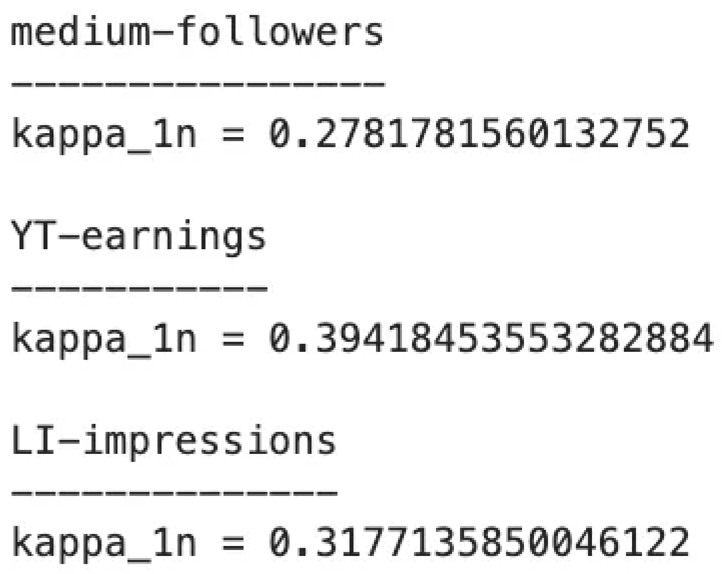
```
def mean_abs_deviation(S):
    """
        Computation of mean absolute deviation of an input sample S
    """
    M = np.mean(np.abs(S - np.mean(S)))

    return M

def generate_n_sample(X,n):
    """
        Function to generate n random samples of size len(X) from an array X
    """
    # initialize sample
    S_n=0

    for i in range(n):
        # ramdomly sample len(X) observations from X and add it to the sample
        S_n = S_n + X[np.random.randint(len(X), size=int(np.round(len(X))))]

    return S_n

def kappa(X,n):
    """
        Taleb's kappa metric from n0=1 as described here: https://arxiv.org/abs/

        Note: K_1n = kappa(1,n) = 2 - ((log(n)-log(1))/log(M_n/M_1)), where M_n
    """
    S_1 = X
    S_n = generate_n_sample(X,n)

    M_1 = mean_abs_deviation(S_1)
    M_n = mean_abs_deviation(S_n)

    K_1n = 2 - (np.log(n)/np.log(M_n/M_1))

    return K_1n
```

The first function, *mean_abs_deviation(),* computes the mean absolute deviation as defined earlier.

Next, we need a way to generate and sum *n* samples from our empirical data. Here, I take a naive approach and randomly sample an input array (X) *n* times and sum the samples together.

Finally, I bring together *mean_abs_deviation(S)* and *generate_n_sample(X,n)* to implement the κ calculation defined before and compute it for each dataset.

```
n = 100 # number of samples to include in kappa calculation

for filename in filename_list:
    df = df_dict[filename]

    # print results
    print(filename)
    print("-"*len(filename))
    print("kappa_1n = " + str(kappa(df.iloc[:,0].to_numpy(), n)))
    print("")
```



```
medium-followers
----------------
kappa_1n = 0.2781781560132752

YT-earnings
-----------
kappa_1n = 0.3941845355328284

LI-impressions
--------------
kappa_1n = 0.3177135850046122
```

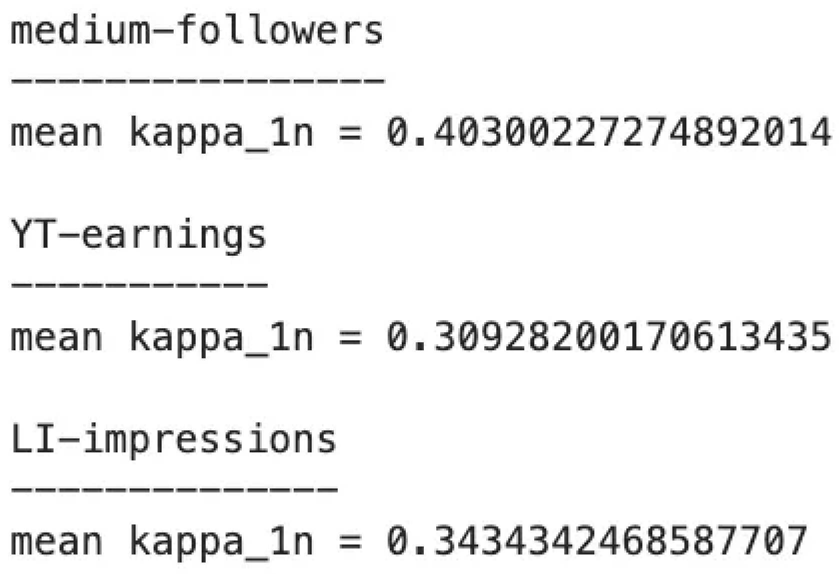
κ(1,100) values for each dataset. Image by author.

The results above give us yet another story. However, given the implicit randomness of this calculation (recall *generate_n_sample()* definition) and the fact we're dealing with fat tails, point estimates (i.e. just running the computation once) cannot be trusted.

Accordingly, I run the same calculation 1000x and print the mean *κ(1,100)* for each dataset.

```python
num_runs = 1_000
kappa_dict = {}

for filename in filename_list:
    df = df_dict[filename]

    kappa_list = []
    for i in range(num_runs):
        kappa_list.append(kappa(df.iloc[:,0].to_numpy(), n))


    kappa_dict[filename] = np.array(kappa_list)

    print(filename)
    print("-"*len(filename))
```

```
        print("mean kappa_1n = " + str(np.mean(kappa_dict[filename])))
        print("")
```



```
medium-followers
----------------
mean kappa_1n = 0.4030227274892014

YT-earnings
-----------
mean kappa_1n = 0.30928200170613435

LI-impressions
--------------
mean kappa_1n = 0.343434246858707
```

Mean κ(1,100) values from 1000 runs for each dataset. Image by author.

These more stable results indicate Medium followers are the most fat-tailed, followed by LinkedIn Impressions and YouTube earnings.

Note: One can compare these values to Table III in ref [3] to better understand each κ value. Namely, these values are comparable to a Pareto distribution with α between 2 and 3.

Although each heuristic told a slightly different story, all signs point toward Medium followers gained being the most fat-tailed of the 3 datasets.

Conclusion

While binary labeling data as fat-tailed (or not) may be tempting, fat-tailedness lives on a spectrum. Here, we broke down 4 heuristics for quantifying *how fat-tailed* data are.

Although each approach has its limitations, they provide practitioners with quantitative ways of comparing the fat-tailedness of empirical data.

👉 **More on Power Laws & Fat Tails:** Introduction | Power Law Fits

Resources

Connect: My website | Book a call | Ask me anything

Socials: YouTube 🎥 | LinkedIn | Twitter

Support: Buy me a coffee ☕️

[1] Scipy Kurtosis

[2] Scipy Moment

[3] arXiv:1802.05495 [stat.ME]

[4] https://en.wikipedia.org/wiki/Log-normal_distribution

[5] Pham-Gia, T., & Hung, T. (2001). The mean and median absolute deviations. *Mathematical and Computer Modelling*, *34*(7–8), 921–936. https://doi.org/10.1016/S0895-7177(01)00109-1

Data Science Statistics Python Fat Tails Power Law