Member-only story

Detecting Power Laws in Real-world Data with Python *

Breaking down a Maximum Likelihood-based approach with example code

Shaw Talebi

Published in Towards Data Science · 10 min read · Nov 24, 2023

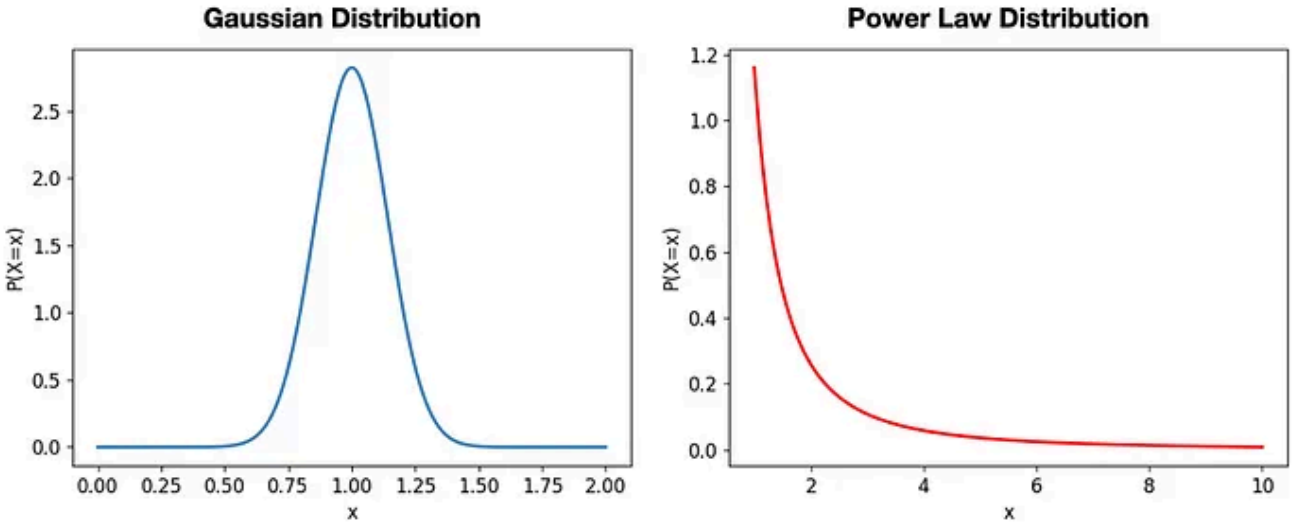
Photo by Luke Chesser on Unsplash

This is the 2nd article in a series about Power Laws and Fat Tails. In the previous post, I gave a beginner-friendly introduction to power laws and presented 3 problems with our standard statistical tools in analyzing them. While awareness can help us avoid these problems, it is not always clear what distribution some given data follow in practice. In this article, I will describe how to objectively detect Power Laws from real-world data and share a concrete example with social media data.

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

Power Laws Break STAT 101

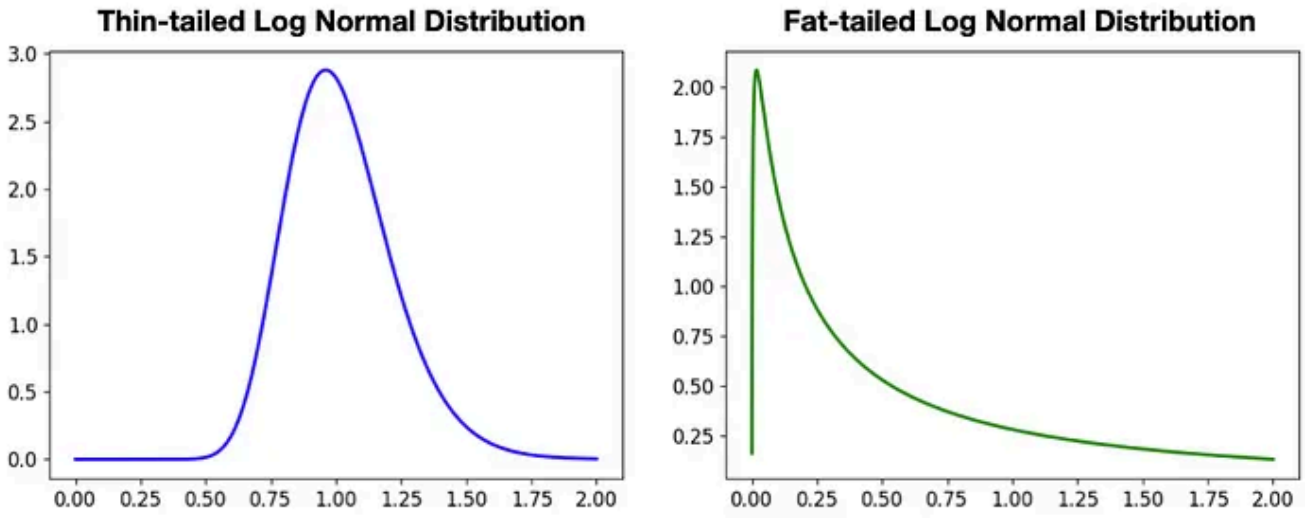
In the previous article, we focused on two types of distributions: the Gaussian distribution and the Power Law distribution. We saw that these distributions had diametrically opposite statistical properties. Namely, **Power Laws are driven by rare events, while Gaussians are not.**



Example Gaussian and Power Law distributions, respectively. Image by author

This rare-event-driven property raised 3 problems with many of our favorite statistical tools (e.g. mean, standard deviation, regression, etc.) in analyzing Power Laws. The takeaway was that if data are Gaussian-like, one can use common approaches like regression and computing expectation values without worry. However, if data are more **Power Law-like, these techniques can give incorrect and misleading results.**

We also saw a third (more mischievous) distribution that could resemble both a Gaussian and a Power Law (despite their opposite properties) called a **Log Normal distribution.**



The (mischievous) Log Normal distribution appears both Guassian-like and Power Law-like. Image by author.

This ambiguity presents challenges for practitioners in deciding the *best* way to analyze a given dataset. To help overcome these challenges, it can be advantageous to determine whether data fit a Power Law, Log Normal, or some other type of distribution.

Log-Log Approach

A popular way of fitting a Power Law to real-world data is what I'll call the "Log-Log approach" [1]. The idea comes from **taking the logarithm of the Power Law's probability density function (PDF)**, as derived below.

Power Law PDF

$$p(z) = F(z) * z^{-(\alpha+1)}$$

Where, $p(z)$ = Power Law's probability density function

$F(z)$ is a slowly varying function

$z > z_{min}$

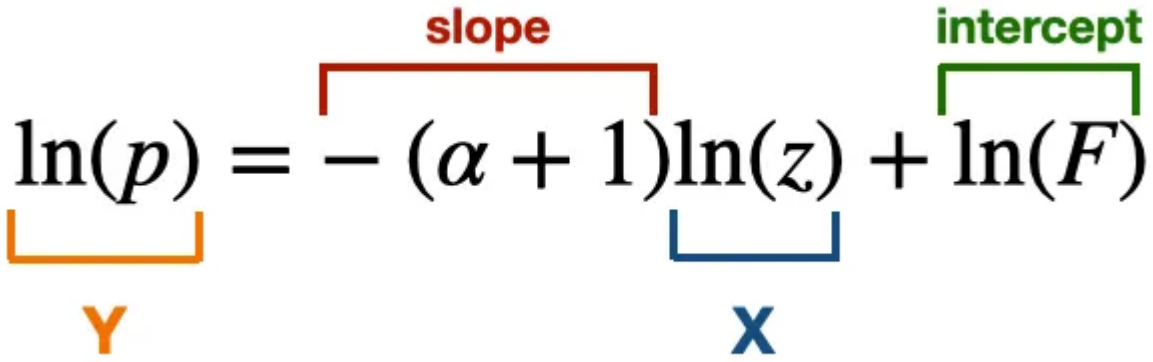
Taking log of both sides

$$\ln(p) = \ln(F * z^{-}(\alpha + 1))$$

$$= \ln(F) - (\alpha + 1)\ln(z)$$

Taking the log of Power Law probability distribution function [2]. Image by author.

The above derivation translates the Power Law's PDF definition into a linear equation, as shown in the figure below.



Highlight the linear form of the log(PDF). Image by author.

This implies that the **histogram of data following a power law will follow a straight line.** In practice, what this looks like is generating a histogram for some data and plotting it on a log-log plot [1]. One might go even further and perform a linear regression to estimate the distribution's α value (here, α = -m+1).

However, there are significant limitations to this approach. These are described in reference [1] and summarized below.

- Slope (hence α) estimations are subject to systematic errors
- Regression errors can be hard to estimate
- Fit can look good even if the distribution does not follow a Power Law
- Fits may not obey basic conditions for probability distributions e.g. normalization

Maximum Likelihood Approach

While the Log-Log approach is simple to implement, its limitations make it less than optimal. Instead, we can turn to a more mathematically sound

approach via **Maximum Likelihood**, a widely used statistical **method for inferring the *best* parameters for a model given some data.**

Maximum Likelihood consists of 2 key steps. **Step 1:** obtain a likelihood function. **Step 2:** maximize the likelihood with respect to your model parameters.

Step 1: Write Likelihood Function

Likelihood is a special type of probability. Put simply, it **quantifies the probability of our data given a particular model.** We can express it as the joint probability over all our observed data [3]. In the case of a Pareto distribution, we can write this as follows.

Likelihood Function definition,

$$L(\alpha, x_{min}) = \Pi_{i=1}^{n}\ p(x_i \mid \alpha, x_{min})$$

For, Pareto Distribution i.e. $p(x) = \alpha x_{min}^{\alpha} x^{-(\alpha+1)}$

$$\implies L(\alpha, x_{min}) = \Pi_{i=1}^{n} \alpha x_{min}^{\alpha} x_i^{-(\alpha+1)}$$

$$= \alpha^n x_{min}^{n\alpha} \Pi_{i=1}^{n} x_i^{-(\alpha+1)}$$

Likelihood function for Pareto distribution (i.e. a special type of Power Law) [4]. **Note**: when working with Likelihood functions, observations (i.e. x_i) are fixed while model parameters are what vary. Image by author.

To make maximizing the likelihood a little easier, it is customary to work with the log-likelihood (they are maximized by the same value of α).

Log-likelihood function,

$$l(\alpha, x_{min}) \equiv \ln(L(\alpha, x_{min})) = \ln(\alpha^n x_{min}^{n\alpha} \Pi_{i=1}^{n} x_i^{-(\alpha+1)})$$

$$= \ln \alpha^n + \ln x_{min}^{n\alpha} + \Sigma_{i=1}^{n} \ln(x_i^{-(\alpha+1)})$$

$$= n \ln \alpha + n\alpha \ln x_{min} - (\alpha + 1)\Sigma_{i=1}^{n} \ln(x_i)$$

Log-likelihood derivation [4]. Image by author.

Step 2: Maximize Likelihood

With a (log) likelihood function in hand, we can now frame the task of determining the best choice of parameters as an optimization problem. To find the optimal α value based on our data, this boils down to setting the derivative of $l(\alpha)$ with respect to α equal to zero and then solving for α. A derivation of this is given below.

Maximize Log-likelihood

$$\frac{\partial l}{\partial \alpha} = \frac{n}{\alpha} + n \ln x_{min} - \Sigma_{i=1}^{n} \ln x_i$$

$$\frac{\partial l}{\partial \alpha} = 0 \implies \hat{\alpha} = \Sigma_{i=1}^{n} \ln(x_i) - n \ln x_{min}$$

$$\hat{\alpha} = \frac{n}{\Sigma_{i=1}^{n} \ln(x_i / x_{min})}$$

Max likelihood estimator

Derivation of max likelihood estimator for α [4]. Image by author.

This provides us with the so-called **Maximum Likelihood estimator** for α. With this, we can plug in observed values of x to estimate a Pareto distribution's α value.

With the theoretical foundation set, let's see what this looks like when applied to real-world data (from my social media accounts).

Example code: Fitting Power Laws to Social Media Data

One domain in which fat-tailed data are prevalent is social media. For instance, a small percentage of creators get the bulk of the attention, a minority of Medium blogs get the majority of reads, and so on.

Here we will use the *powerlaw* Python library to determine whether data from my various social media channels (i.e. Medium, YouTube, LinkedIn)

truly follow a Power Law distribution. The data and code for these examples are available on the GitHub repository.

Artificial Data

Before applying the Maximum Likelihood-based approach to messy data from the real world, let's see what happens when we apply this technique to artificial data (*truly*) generated from Pareto and Log Normal distributions, respectively. This will help ground our expectations before using the approach on data in which we do not know the "true" underlying distribution class.

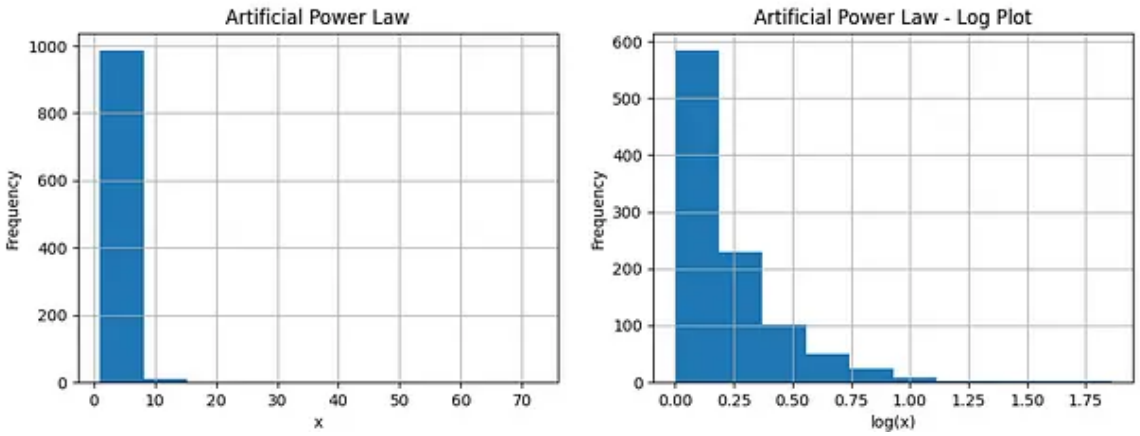
First, we import some helpful libraries.

```
import numpy as np
import matplotlib.pyplot as plt
import powerlaw
import pandas as pd

np.random.seed(0)
```

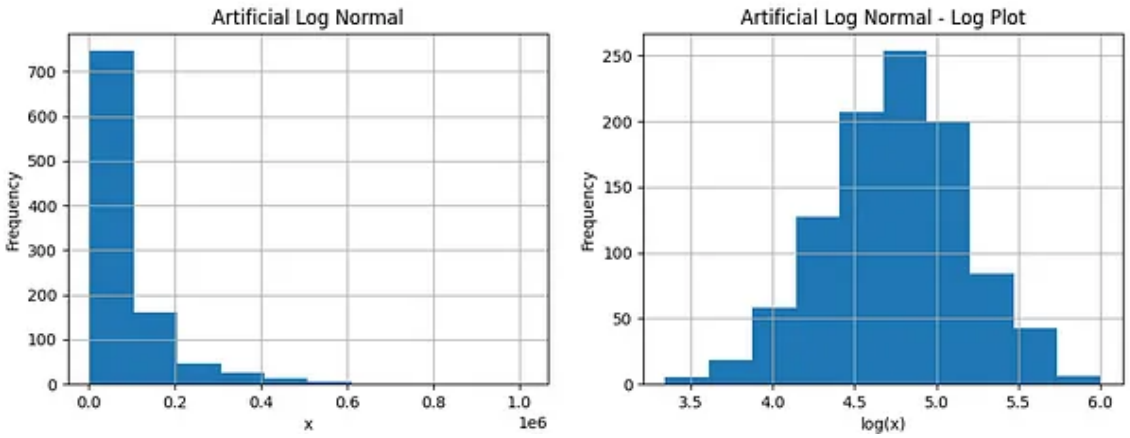
Next, let's generate data from Pareto and Log Normal distributions.

```
# power law data
a = 2
x_min = 1
n = 1_000
x = np.linspace(0, n, n+1)
s_pareto = (np.random.pareto(a, len(x)) + 1) * x_min

# log normal data
m = 10
s = 1
s_lognormal = np.random.lognormal(m, s, len(x)) * s * np.sqrt(2*np.pi)
```

To get a sense of what these data look like, it's helpful to plot histograms. Here, I plot a histogram of each sample's raw values and the log of the raw values. This latter distribution makes it easier to distinguish between Power Law and Log Normal data visually.



Histograms of data from Power Law distribution. Image by author.



Histograms of data from Log Normal distribution. Image by author.

As we can see from the above histograms, the distributions of raw values look qualitatively similar for both distributions. However, we can see a **stark difference in the log distributions**. Namely, the log Power Law distribution is highly skewed and not mean-centered, while the log of the Log Normal distribution is reminiscent of a Gaussian distribution.

Now, we can use the *powerlaw* library to fit a Power Law to each sample and estimate values for α and x_min. Here's what that looks like for our Power Law sample.

```
# fit power to power law data
results = powerlaw.Fit(s_pareto)

# printing results
print("alpha = " + str(results.power_law.alpha)) # note: powerlaw lib's alpha de
print("x_min = " + str(results.power_law.xmin))
print('p = ' + str(compute_power_law_p_val(results)))

# Calculating best minimal value for power law fit
# alpha = 2.9331912195958676
# x_min = 1.2703447024073973
# p = 0.999
```

The fit does a decent job at estimating the true parameter values (i.e. a=3, x_min=1), as seen by the alpha and x_min values printed above. The value p above quantifies the quality of the fit. A higher p means a better fit *(more on this value in section 4.1 of ref [1])*.

We can do a similar thing for the Log Normal distribution.

```
# fit power to log normal data
results = powerlaw.Fit(s_lognormal)
print("alpha = " + str(results.power_law.alpha)) # note: powerlaw lib's alpha de
print("x_min = " + str(results.power_law.xmin))
print('p = ' + str(compute_power_law_p_val(results)))

# Calculating best minimal value for power law fit
# alpha = 2.5508694755027337
# x_min = 76574.4701482522
# p = 0.999
```

We can see that the Log Normal sample also fits a Power Law distribution well (p=0.999). Notice, however, that the x_min value is far in the tail. While this may be helpful for some use cases, it doesn't tell us much about the distribution that best fits all the data in the sample.

To overcome this, we can manually set the x_min value to the sample minimum and redo the fit.

```
# fixing xmin so that fit must include all data
results = powerlaw.Fit(s_lognormal, xmin=np.min(s_lognormal))
print("alpha = " + str(results.power_law.alpha))
print("x_min = " + str(results.power_law.xmin))
```

```
# alpha = 1.3087955873576855
# x_min = 2201.318351239509
```

The .Fit() method also automatically generates estimates for a Log Normal distribution.

```
print("mu = " + str(results.lognormal.mu))
print("sigma = " + str(results.lognormal.sigma))

# mu = 10.933481999687547
# sigma = 0.9834599169175509
```

The estimated Log Normal parameter values are close to the actual values (mu=10, sigma=1), so the fit did a good job once again!

However, by fixing x_min, we lost our quality metric p (*for whatever reason, the method doesn't generate values for it when x_min is provided*). So this begs the question, *which distribution parameters should I go with? The Power Law or Log Normal?*

To answer this question, we can compare the Power Law fit to other candidate distributions via **Log-likelihood ratios (R)**. A positive R implies the Power Law is a better fit, while a negative R implies the alternative distribution is better. Additionally, each comparison gives us a significance value (p). This is demonstrated in the code block below.

```
distribution_list = ['lognormal', 'exponential', 'truncated_power_law', \
      'stretched_exponential', 'lognormal_positive']
```

```
for distribution in distribution_list:
    R, p = results.distribution_compare('power_law', distribution)
    print("power law vs " + distribution +
          ": R = " + str(np.round(R,3)) +
          ", p = " + str(np.round(p,3)))

# power law vs lognormal: R = -776.987, p = 0.0
# power law vs exponential: R = -737.24, p = 0.0
# power law vs truncated_power_law: R = -419.958, p = 0.0
# power law vs stretched_exponential: R = -737.289, p = 0.0
# power law vs lognormal_positive: R = -776.987, p = 0.0
```

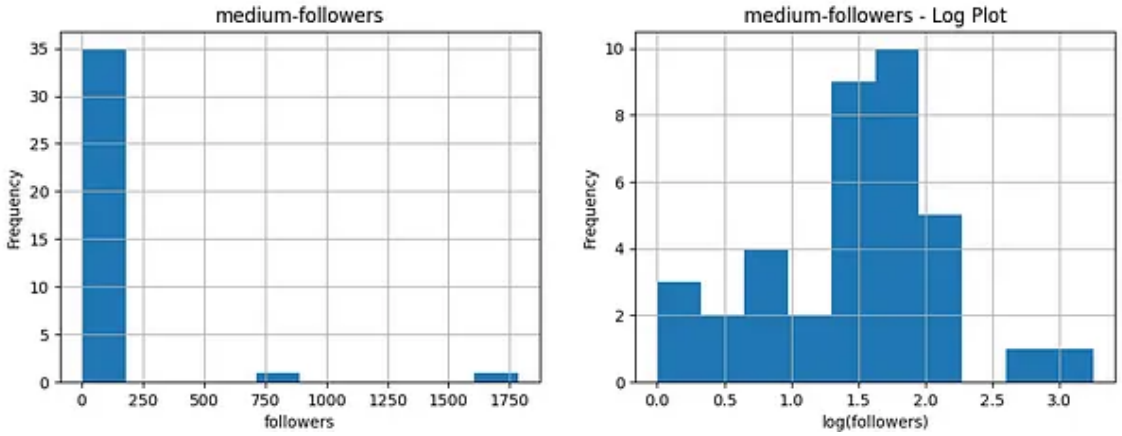
As shown above, every alternative distribution is preferred over the Power Law when including all the data in the Log Normal sample. Additionally, based on the likelihood ratios, the lognormal and lognormal_positive fits work best.

Real-world Data

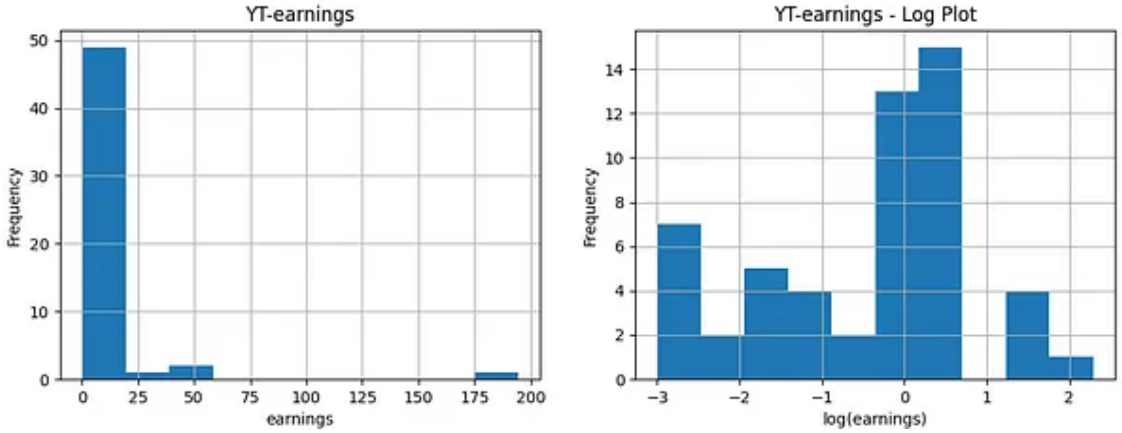
Now that we've applied the *powerlaw* library to data where we know the ground truth let's try it on data for which the underlying distribution is unknown.

We will follow a similar procedure as we did above but with data from the real world. Here, we will analyze the following data. Monthly followers gained on my **Medium** profile, earnings across all my **YouTube** videos, and daily impressions on my **LinkedIn** posts for the past year.

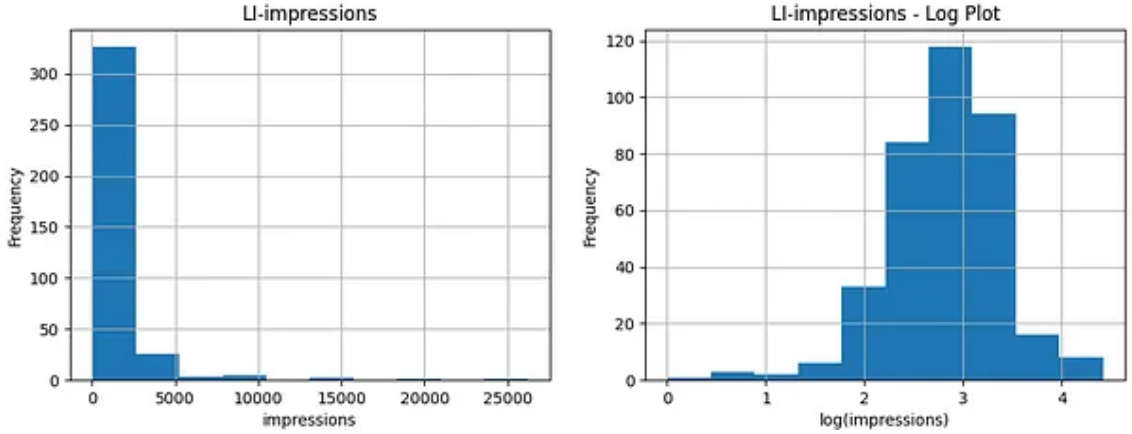
We'll start by plotting histograms.



Medium followers gained histograms. Image by author.



YouTube video earnings histograms. Image by author.



LinkedIn daily impressions histograms. Image by author.

Two things jump out to me from these plots. **One**, all three look more like the Log Normal histograms than the Power Law histograms we saw before. **Two**, the Medium and YouTube distributions are sparse, meaning they may have insufficient data for drawing strong conclusions.

Next, we'll apply the Power Law fit to all three distributions while setting x_min as the smallest value in each sample. The results of this are printed below.

Medium Followers	YouTube Earnings	LinkedIn Impressions
Power Law Fit a = 1.2942788495873196 xmin = 1.0	Power Law Fit: 97% a = 1.790855210187927 xmin = 1.162	Power Law Fit a = 1.1550153110047172 xmin = 1.0
Log Normal Fit mu = 3.273520645763708 sigma = 1.7482207134146353	Log Normal Fit mu = -1864.7463763738424 sigma = 48.61486236978004	Log Normal Fit mu = 6.450975671490735 sigma = 1.322621515289094

Power Law and Log Normal parameter estimates for empirical data. Image by author.

To determine which distribution is best, we can again do head-to-head comparisons of the Power Law fit to some alternatives. These results are given below.

Medium Followers

```
Comparing Power Law to Other Fits
power law vs lognormal: R = -12.709, p = 0.007
power law vs exponential: R = 3.905, p = 0.767
power law vs truncated_power_law: R = -10.097, p = 0.0
power law vs stretched_exponential: R = -12.676, p = 0.003
power law vs lognormal_positive: R = -12.709, p = 0.007
```

YouTube Earnings

```
Comparing Power Law to Other Fits
power law vs lognormal: R = -0.603, p = 0.344
power law vs exponential: R = 24.969, p = 0.004
power law vs truncated_power_law: R = -0.09, p = 0.671
power law vs stretched_exponential: R = 0.252, p = 0.559
power law vs lognormal_positive: R = 2.074, p = 0.181
```

LinkedIn Impressions

```
Comparing Power Law to Other Fits
power law vs lognormal: R = -425.472, p = 0.0
power law vs exponential: R = -405.675, p = 0.0
power law vs truncated_power_law: R = -210.094, p = 0.0
power law vs stretched_exponential: R = -422.689, p = 0.0
power law vs lognormal_positive: R = -425.472, p = 0.0
```

Fit comparisons of Power Law and alternative distributions. Image by author.

Using the rule of thumb significance cutoff of $p<0.1$ we can draw the following conclusions. Medium followers and LinkedIn impressions best fit a Log Normal distribution, while a Power Law best represents YouTube earnings.

Of course, since the Medium followers and YouTube earrings data here is limited ($N<100$), we should take any conclusions from those data with a grain of salt.

Conclusion

Many standard statistical tools break down when applied to data following a Power Law distribution. Accordingly, detecting Power Laws in empirical data can help practitioners avoid incorrect analyses and misleading conclusions.

However, Power Laws are an extreme case of the more general phenomenon of **fat tails**. In the next article of this series, we will take this work one step further and quantify fat-tailedness for any given dataset via 4 handy heuristics.

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

4 Ways to Quantify Fat Tails with Python

Intuition and Example Code

towardsdatascience.com

Resources

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

Socials: YouTube 📽️ | LinkedIn | Twitter

Support: Buy me a coffee ☕

The Data Entrepreneurs

A community for entrepreneurs in the data space. 👉 Join the Discord!

medium.com

[1] arXiv:0706.1062 [physics.data-an]

[2] arXiv:2001.10488 [stat.OT]

[3] https://en.wikipedia.org/wiki/Likelihood_function

[4] https://en.wikipedia.org/wiki/Pareto_distribution

Data Science | Statistics | Data Analysis | Python | Hands On Tutorials

Written by Shaw Talebi

Edit profile

6.8K Followers · Writer for Towards Data Science

Data Scientist | PhD, Physics | Editor for The Data Entrepreneurs

More from Shaw Talebi and Towards Data Science

Shaw Talebi in Towards Data Science

QLoRA—How to Fine-Tune an LLM on a Single GPU

An introduction with Python example code (ft. Mistral-7b)

· 16 min read · Feb 22, 2024

726 2

Patrick Brus in Towards Data Science

How to Write Clean Code in Python

Top takeaways from the book Clean Code

· 21 min read · 6 days ago

885 11

Hamza Gharbi in Towards Data Science

Building a Chat App with LangChain, LLMs, and Streamlit f...

Build and deploy a chat application for complex database interaction with LangChai...

16 min read · Feb 9, 2024

1K 9

Shaw Talebi in Towards Data Science

How to Build an AI Assistant with OpenAI + Python

Step-by-step guide on using the Assistants API & Fine-tuning

· 13 min read · Feb 8, 2024

478 4

See all from Shaw Talebi

See all from Towards Data Science

Recommended from Medium

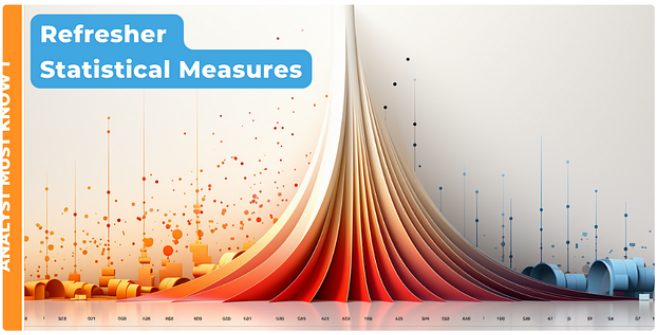
Louis Chan in Towards Data Science

SHAP: Explain Any Machine Learning Model in Python

Your Comprehensive Guide to SHAP, TreeSHAP, and DeepSHAP

✦ · 13 min read · Jan 11, 2023

805 3



Prof. Frenzel

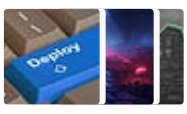
Statistical Measures Every Analyst Must Know—Part1

Measures of Central Tendency, Variability, Quartiles, Z-Scores, and as always:...

11 min read · Feb 4, 2024

283 1

Lists



Predictive Modeling w/ Python

20 stories · 955 saves

Practical Guides to Machine Learning

10 stories · 1125 saves

Coding & Development

11 stories · 473 saves

ChatGPT prompts

44 stories · 1187 saves

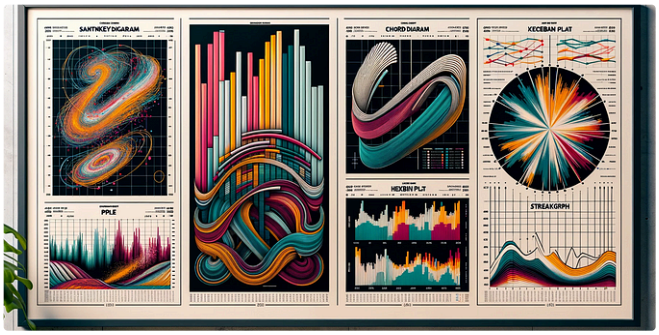
Cris Velasquez

Further Implementation of Dynamic Risk Management...

15 Time-varying Techniques for Proactive Risk Analytics [Part 2/2]

13 min read · Sep 20, 2023

350 3



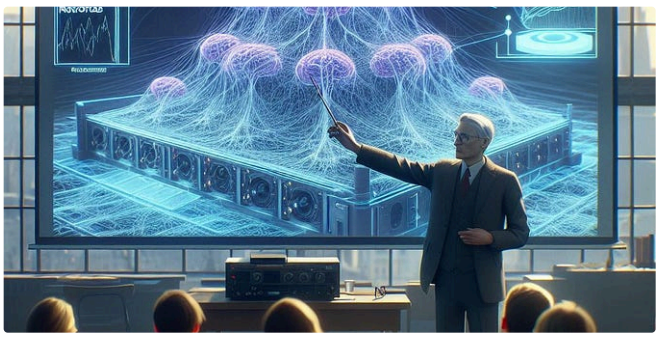
Dr. Ashish Bamania in Level Up Coding

5 Extremely Useful Plots For Data Scientists That You Never Knew...

"5. Theme River"

6 min read · Jan 2, 2024

2.8K 19

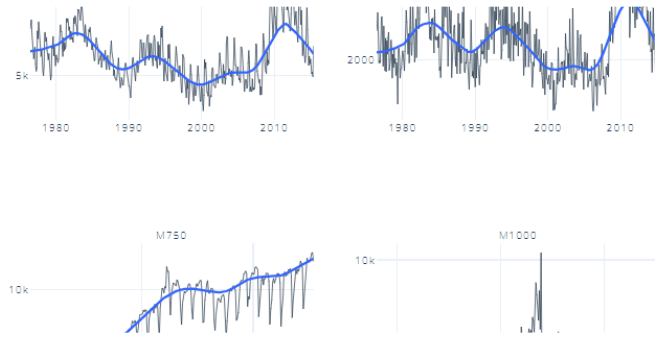


Austin Star... in Artificial Intelligence in Plain Engli...

Reinforcement Learning is Dead. Long Live the Transformer!

Large Language Models are more powerful than you imagine

8 min read · Jan 13, 2024



sunku sowmya Sree

PyTimeTK—Time Series Analysis Package

What is PyTimeTK

8 min read · Dec 11, 2023