Transforming Skewed Data

A Survey of Friendly Functions

Corey Wade Aug 22, 2018 · 7 min read ★

Note: The following code is written in Python and excerpted from various Jupyter Notebook. Inline comments have been eliminated to make the article more readable.

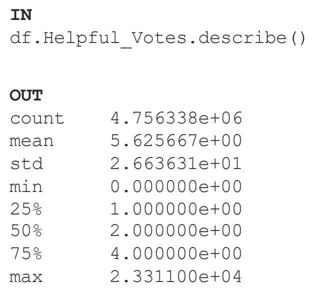
REX RILEY
SAFETY OFFICER
Keep Calm and Get your Safety On
DIRECTORATE OF AEROSPACE SAFETY • DEPUTY INSPECTOR GENERAL FOR INSPECTION AND SAFETY, USAF • NORTON AFB, CALIFORNIA • VOL. XXIII, NO. 2, 1969

Skewed data is cumbersome and common. It's often desirable to transform skewed data and to convert it into values between 0 and 1.

Standard functions used for such conversions include Normalization, the Sigmoid, Log, Cube Root and the Hyperbolic Tangent. It all depends on what one is trying to accomplish.

Here's an example of a skewed column that I generated from an 8.9 million row Amazon Books Review Dataset (Julian McAuley, UCSD). df.Helpful_Votes gives the total number of helpful votes (as opposed to unhelpful) that each book review received.

df_Helpful_Votes Original Data



```
IN
df.Helpful_Votes.describe()

OUT
count    4.756338e+06
mean     5.625667e+00
std      2.663631e+01
min      0.000000e+00
25%      1.000000e+00
50%      2.000000e+00
75%      4.000000e+00
max      2.331100e+04
```

0, 1, 2, 4, 23311. That's quite a jump!

Removing outliers is an option, but not one that I want to use here. My end goal is to build a machine learning algorithm to predict whether a given review is helpful, so reviews with the most helpful votes are indispensable.

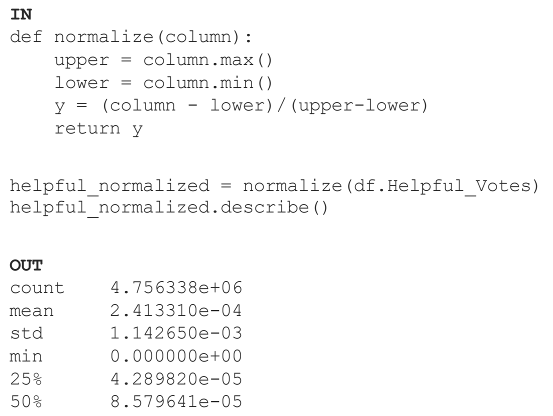
I will use the functions listed above to transform the data, explaining pros and cons along the way. As with most data science, there is no correct function. It depends on the data, and the goal of the analyst.

Normalization converts all data points to decimals between 0 and 1. If the min is 0, simply divide each point by the max.

If the min is not 0, subtract the min from each point, and then divide by the min-max difference.

The following function includes both cases.

Normalization Function

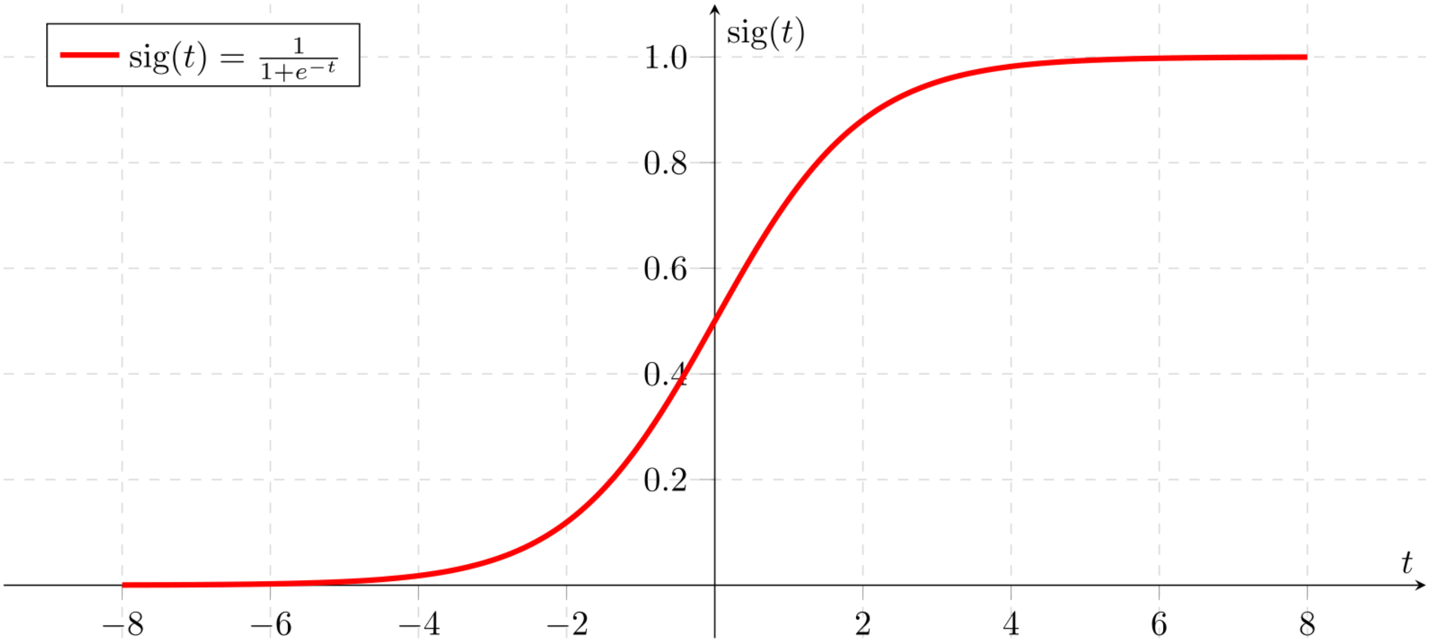


```
IN
def normalize(column):
    upper = column.max()
    lower = column.min()
    y = (column - lower)/(upper-lower)
    return y

helpful_normalized = normalize(df.Helpful_Votes)
helpful_normalized.describe()

OUT
count    4.756338e+06
mean     2.413310e-04
std      1.142650e-03
min      0.000000e+00
25%      4.289820e-05
50%      8.579641e-05
```

```
75%      1.715928e-04
max      1.000000e+00
```

After normalization, the data is just as skewed as before. If the goal is simply to convert the data to points between 0 and 1, normalization is the way to go. Otherwise, normalization should be used in conjunction with other functions.

Next, the Sigmoid function. It's worth looking at a visual if you have not seen the Sigmoid before.



It's a beautifully smooth curve that guarantees a 0 to 1 range. Let's see how it performs on df.Helpful_Votes.

Sigmoid Function

```
IN
def sigmoid(x):
    e = np.exp(1)
    y = 1/(1+e**(-x))
    return y

helpful_sigmoid = sigmoid(df.Helpful_Votes)
helpful_sigmoid.describe()

OUT
count    4.756338e+06
mean     8.237590e-01
std      1.598215e-01
min      5.000000e-01
25%      7.310586e-01
50%      8.807971e-01
75%      9.820138e-01
max      1.000000e+00
```

A definite improvement. The new data is between 0 and 1 as expected, but the min is 0.5. This makes sense when looking at the graph as there are no negative values when tallying votes.

Another point of consideration is the spread. Here, the 75th percentile is within 0.2 of the 100th percentile, comparable to other quartiles. But in the original data, the 75th percentile more than 20,000 away from the 100th percentile, not remotely close to other quartiles. In this case, the data has been distorted.

The sigmoid function can be tweaked to improve results, but for now, let's explore other options.

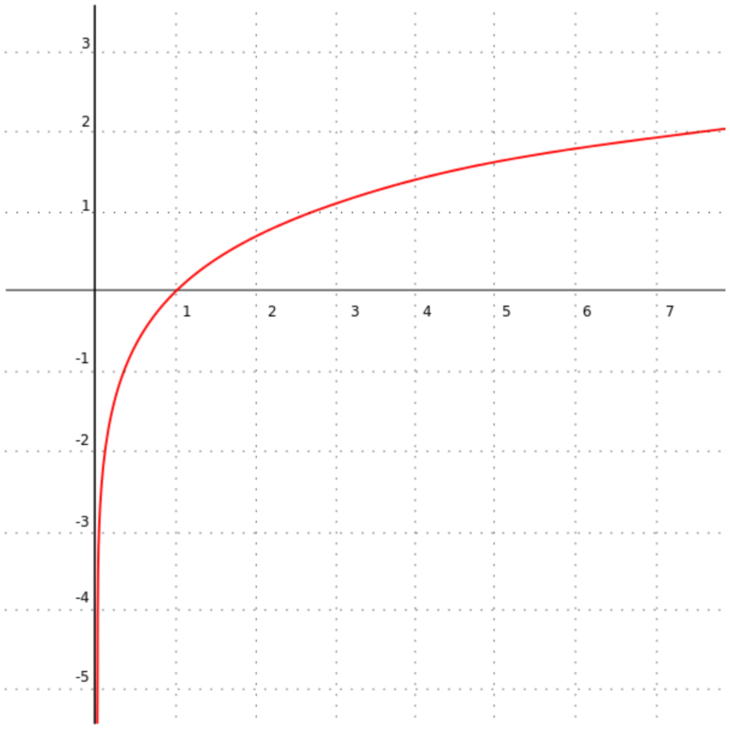
Next up, logarithms. An excellent choice for making data less skewed. When using log with Python, the default base is usually *e*.

Log Function

```
IN
helpful_log = np.log(df.Helpful_Votes)
helpful_log.describe()

OUT
RuntimeWarning: divide by zero encountered in log
```

Whoops! Division by zero! How did this happen? Perhaps a visual will clarify matters.

Log Graph

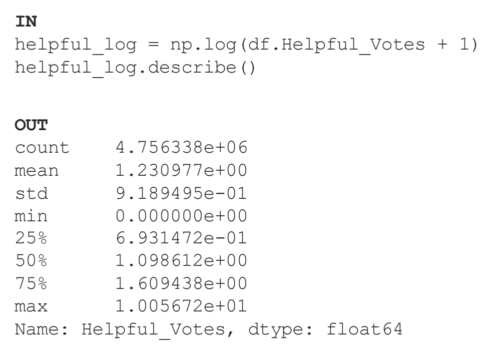


Af Adrian Neumann — Eget arbejde, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1834287

Ah, yes. The domain for log is strictly greater than 0. That's a vertical asymptote heading down the y-axis. As x approaches 0, y approaches negative infinity. In other words, 0 is excluded from the domain.

Many of my data points are 0 because many reviews received no helpful votes. For a quick fix, I can add 1 to each data point. This works well since the log of 1 is 0. Furthermore, the same spread is retained since *all* points are increased by 1.

Log Function + 1

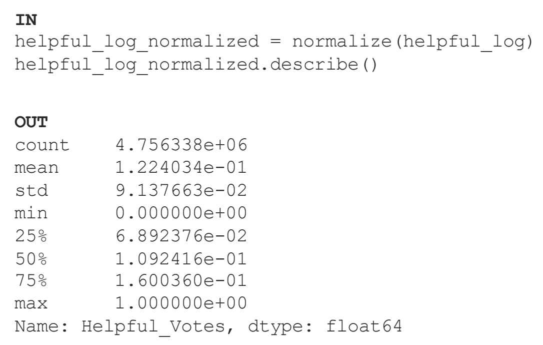


```
IN
helpful_log = np.log(df.Helpful_Votes + 1)
helpful_log.describe()

OUT
count    4.756338e+06
mean     1.230977e+00
std      9.189495e-01
min      0.000000e+00
25%      6.931472e-01
50%      1.098612e+00
75%      1.609438e+00
max      1.005672e+01
Name: Helpful_Votes, dtype: float64
```

Excellent. The new range is from 0 to 10, and the quartiles are reflective of the original data.

Time to normalize.

Log Function + 1 Normalized



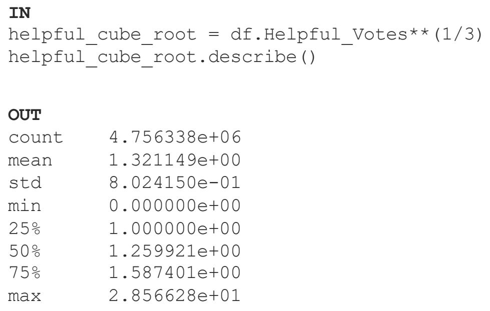
```
IN
helpful_log_normalized = normalize(helpful_log)
helpful_log_normalized.describe()

OUT
count    4.756338e+06
mean     1.224034e-01
std      9.137663e-02
min      0.000000e+00
25%      6.892376e-02
50%      1.092416e-01
75%      1.600360e-01
max      1.000000e+00
Name: Helpful_Votes, dtype: float64
```

This looks very reasonable. The log function plus normalization is an excellent way to transform skewed data if the results can still be skewed. There is, however, one major drawback in this case.

Why transform data to values between 0 and 1 in the first place? It's usually so percentages and probability can play a role. In my case, I want the median at around 50%. After normalizing here, the median is at 0.1.

When numbers are too large, one can try fractional exponents as a means of transformation. Consider the cube root.

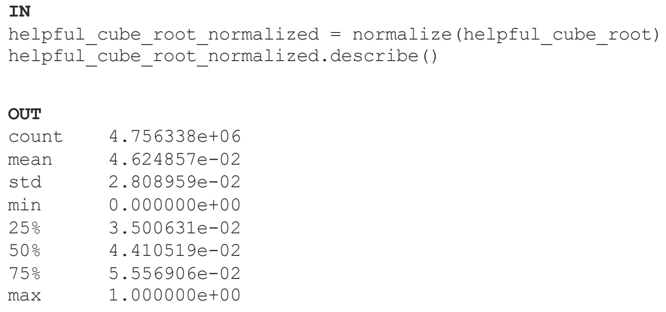
Cube Root



```
IN
helpful_cube_root = df.Helpful_Votes**(1/3)
helpful_cube_root.describe()

OUT
count    4.756338e+06
mean     1.321149e+00
std      8.024150e-01
min      0.000000e+00
25%      1.000000e+00
50%      1.259921e+00
75%      1.587401e+00
max      2.856628e+01
```

This is very similar to log, but the range here is larger, from 0 to 28.

Cube Root Normalized

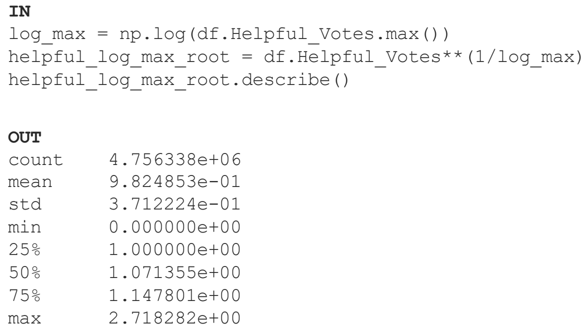


```
IN
helpful_cube_root_normalized = normalize(helpful_cube_root)
helpful_cube_root_normalized.describe()

OUT
count    4.756338e+06
mean     4.624857e-02
std      2.808959e-02
min      0.000000e+00
25%      3.500631e-02
50%      4.410519e-02
75%      5.556906e-02
max      1.000000e+00
```

As expected, the new data is more problematic after normalizing. Now the median is at 0.04, a far cry from 50%.

I like to play around with numbers, so I experimented with a few powers. Interesting results came from 1/log_max. I define log_max as the log of the maximum value. (The log of 23,311 is 10.06.)

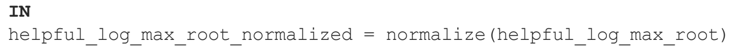
Log Max Root



```
IN
log_max = np.log(df.Helpful_Votes.max())
helpful_log_max_root = df.Helpful_Votes**(1/log_max)
helpful_log_max_root.describe()

OUT
count    4.756338e+06
mean     9.824853e-01
std      3.712224e-01
min      0.000000e+00
25%      1.000000e+00
50%      1.071355e+00
75%      1.147801e+00
max      2.718282e+00
```

A range from 0 to 2.7 is very appealing.

Log Max Root Normalized



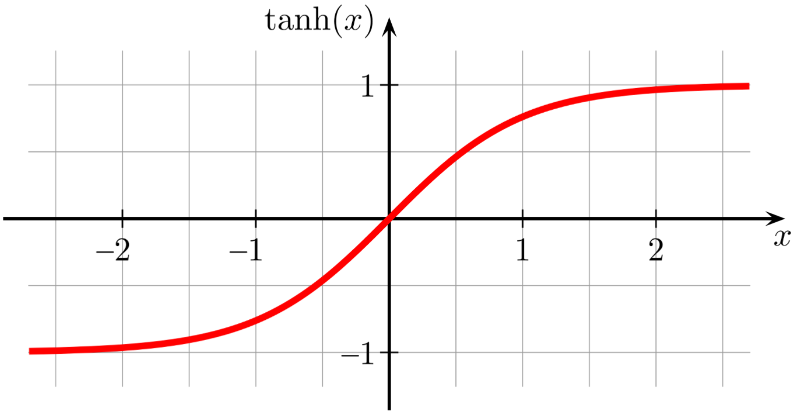
```
IN
helpful_log_max_root_normalized = normalize(helpful_log_max_root)
```

```
helpful_log_max_root_normalized.describe()
```

```
OUT
count     4.756338e+06
mean      3.614362e-01
std       1.365651e-01
min       0.000000e+00
25%       3.678794e-01
50%       3.941294e-01
75%       4.222525e-01
max       1.000000e+00
```

This looks good, but the data's very clustered in the 25–75% range, a cluster that likely extends much further. So although the overall spread is more desirable, it's not quite sufficient here.

It's time for our last standard function, the hyperbolic tangent. Let's start with a graph.



This looks very similar to the sigmoid. One primary difference is the range. The hyperbolic tangent has a range from -1 to 1, whereas the sigmoid has a range from 0 to 1.

In my data, since df.Helpful_Votes are all non-negative, my output will be from 0 to 1.

Hyperbolic Tangent

```
IN
helpful_hyperbolic_tangent = np.tanh(df.Helpful_Votes)
helpful_hyperbolic_tangent.describe()
```

```
OUT
count     4.756338e+06
mean      7.953343e-01
std       3.033794e-01
min       0.000000e+00
25%       7.615942e-01
50%       9.640276e-01
75%       9.993293e-01
max       1.000000e+00
```

There's no need to normalize, but there's on glaring issue.

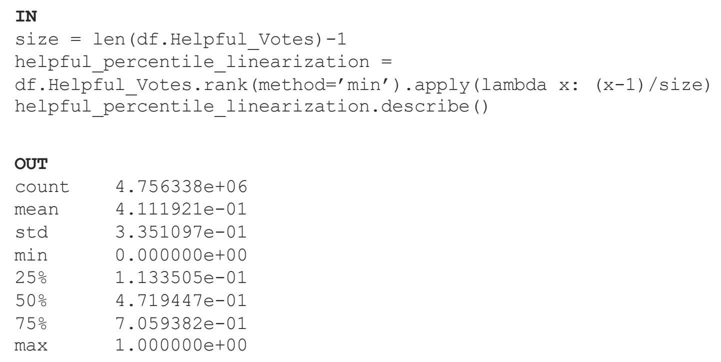
The hyperbolic tangent distorts the data more than the sigmoid. There’s a difference of 0.001 between the 75th and 100th percentiles, much closer than any other quartile. In the original data, that difference is 23,307, 1,000 times greater than the difference between other quartiles.

It’s not even close.

Percentiles provide yet another option. Each data point can be ranked according to its percentile, and pandas provides a nice built-in method, .rank, for dealing with this.

The general idea is that each point receives the value of it’s percentile. In my case, since there are many data points with a low number of helpful votes, there are different ways for choosing these percentiles. I like the “min” method, where all data points receive the percentile given to the first member of the group. The default method is “average,” where all data points with the same value take the mean percentile in that group.

Percentile Linearization



```
IN
size = len(df.Helpful_Votes)-1
helpful_percentile_linearization =
df.Helpful_Votes.rank(method=’min’).apply(lambda x: (x-1)/size)
helpful_percentile_linearization.describe()

OUT
count    4.756338e+06
mean     4.111921e-01
std      3.351097e-01
min      0.000000e+00
25%      1.133505e-01
50%      4.719447e-01
75%      7.059382e-01
max      1.000000e+00
```

Intriguing. Percentile linearization is basically the same as a ranking system. If the difference between the 1st data point and the 2nd should be the same as all data points that are one unit part, percentile linearization is the way to go.

On the downside, Percentile Linearization erases critical signs of skewness. This data appears slightly skewed because there are hundreds of thousands of reviews with 1 and 2 votes, not because there are few reviews with an insanely high number of votes.

Since I am dealing primarily with a ranking system, I combined percentile linearization with my own piecewise linearization function. Logs also played a role.

Whatever your choice or style, never limit yourself to the standard options. Mathematical functions are extremely rich, and their intersection with data science deserves more recognition and flexibility.

I would love to hear about other functions that people use for transforming skewed data. I have heard about the boxcox function, though I have yet to explore it in detail.

More details on Wade's Amazon Book Review project can be viewed on his Github page, Helpful_Reviews.

Sigmoid Data Science Mathematics Data Analysis Statistics