The Log-Sum-Exp Trick

Normalizing vectors of log probabilities is a common task in statistical modeling, but it can result in under- or overflow when exponentiating large values. I discuss the log-sum-exp trick for resolving this issue.

PUBLISHED

09 February 2020

In statistical modeling and machine learning, we often work in a logarithmic scale. There are many good reasons for this. For example, when x and y are both small numbers, multiplying x times y may underflow. However, we can work in a logarithmic scale to convert multiplication to addition because

$$\log(xy) = \log x + \log y. \tag{1}$$

Furthermore, if we are dealing with functions such as $f(x)$ and $g(x)$, then by the linearity of differentiation, we have

$$\frac{\partial}{\partial x} \log[f(x)g(x)] = \frac{\partial}{\partial x} \log f(x) + \frac{\partial}{\partial x} \log g(x). \tag{2}$$

Often, differentiating each function separately is easier than applying the product rule.

These are just two reasons that working with quantities such as log likelihoods and log probabilities is often preferred. And working in log scale is so much more numerically stable that many standard libraries compute probability density functions (PDFs) by simply exponentiating log PDFs. See my previous post on SciPy's multivariate normal PDF for an example.

However, sometimes we need to back out the possibly large log-scale values. For example, if we need to normalize an N-vector of log probabilities, we might naively compute

$$\frac{\exp(x_m)}{\sum_{n=1}^{N}\exp(x_n)} = 1. \tag{3}$$

Since each x_n is a log probability which may be very large, and either negative or positive, then exponentiating might result in under- or overflow respectively. (If a value x is in $(0, 1)$, then $\log(x)$ must be negative. However, we often want to interpret quantities as probabilities even if they are outside of this range.) Think about how the log likelihood can have arbitrary scale depending on the likelihood function and number of data points.

With these ideas in mind, consider the log-sum-exp operation,

$$\mathrm{LSE}(x_1, \ldots, x_N) = \log\left(\sum_{n=1}^{N}\exp(x_n)\right). \tag{4}$$

The log-sum-exp operation can help prevent these numerical issues, but it may not be immediately obvious why it works. First, let's rewrite (3) as

$$\begin{aligned}
\exp(x_m) &= \sum_{n=1}^{N}\exp(x_n) \\
x_m &= \log\sum_{n=1}^{N}\exp(x_n) \\
0 &= x_m - \log\sum_{n=1}^{N}\exp(x_n) \qquad (5) \\
1 &= \exp\left(x_m - \underbrace{\log\sum_{n=1}^{N}\exp(x_n)}_{\mathrm{LSE}(x_1,\ldots,x_N)}\right).
\end{aligned}$$

In words, we can perform the normalization in (3) using the log-sum-exp operation in (4). But does this really help? We're still exponentiating x_n. The final step to seeing the utility of the operation is to consider this derivation:

$$
\begin{aligned}
y &= \log \sum_{n=1}^{N} \exp(x_n) \\
e^y &= \sum_{n=1}^{N} \exp(x_n) \\
e^y &= e^c \sum_{n=1}^{N} \exp(x_n - c) \\
y &= c + \log \sum_{n=1}^{N} \exp(x_n - c).
\end{aligned} \tag{6}
$$

In other words, the log-sum-exp operator in (4) is nice because we can shift the values in the exponent by an arbitrary constant c while still computing the same final value. If we set $c = \max x_1, \ldots, x_N$, we ensure that the largest positive exponentiated term is $\exp(0) = 1$.

Examples in code

Let's look at this trick in code. First, note that it does not take a large value of x_n to cause an overflow:

```
>>> x = np.array([1000, 1000, 1000])
>>> np.exp(x)
array([inf, inf, inf])
```

Your results may vary depending on your machine's precision. Clearly, normalizing `x` as in (3) is impossible with its current values. If you ran this input and code in a larger pipeline, eventually some function would crash on the `inf` values or convert them to `nan` values. However, let's implement a `logsumexp` function—NB: you should probably use SciPy's implementation if working in Python—,

```
def logsumexp(x):
    c = x.max()
    return c + np.log(np.sum(np.exp(x - c)))
```

and then apply the normalization trick in (5),

```
>>> logsumexp(x)
```

```
1001.0986122886682
>>> np.exp(x - logsumexp(x))
array([0.33333333, 0.33333333, 0.33333333])
```

This trick can also help with underflow, when machine precision rounds a small value to zero:

```
>>> x = np.array([-1000, -1000, -1000])
>>> np.exp(x)
array([0., 0., 0.])
>>> np.exp(x - logsumexp(x))
array([0.33333333, 0.33333333, 0.33333333])
```

Again, our normalization in (3), naively computed, would crash, probably due to a division by zero error. This trick still works with a big range in values:

```
>>> x = np.array([-1000, -1000, 1000])
>>> np.exp(x - logsumexp(x))
array([0., 0., 1.])
```

While the probability of the first and second components is not truly zero, this is a reasonable approximation of what those log probabilities represent. Furthermore, we have achieved numerical stability. We can reliably compute (3) without introducing `inf` or `nan` values or dividing by zero.
