Advances in Data Mining

Assignment 2 - Probabilistic Counting and LogLog Algorithm

Athanasios Agrafiotis
s2029413
a.agrafiotis@umail.leidenuniv.nl

Georgios Kyziridis
s2077981
g.kyziridis@umail.leidenuniv.nl

October 16, 2017

Abstract

Generally in computer science there are too many methods and algorithms in order to measure the bitstream length or to count zeros or to count ones. Computer Scientists and Mathematicians were searching for the optimal way to estimate and predict the number of unique distinct items using some probabilistic methods. These algorithmic approaches are used for too many different cases especially when the number of different elements is great or the number of streams which have to be processed is huge for instance Yahoo! wants to count the number of unique users viewing each of its pages in a month. So in case where we cannot store the needed data in main memory these algorithms are useful for making probabilistic predictions measuring the distinct and unique elements. To estimate the number of different elements appearing an a stream, we can hash elements to integers, interpreted as binary numbers. 2 raised to the power that is the longest sequence of 0s seen in the hash value of any stream element is an estimate of the number of different elements. By using many hash functions and combining these estimates, first by taking the averages within groups, and then taking the median of the averages, we get a reliable estimate.

1 Introduction

The aim of this paper is to introduce the idea behind of some algorithms which can measure and estimate the unique elements in a bitstream. In this specific experiment we have implemented two algorithms and tested their results in order to evaluate their significant differences in respect of accuracy and RAM usage. The algorithms we have developed were "Probabilistic Counting" & "LogLog counting". The former was discovered by Flajolet and Martin in 1985 and the latter by Duran and Flajolet which was an extension of Probabilistic Counting. LogLog algorithm uses significantly less memory, at the cost of obtaining only an approximation of the cardinality.

*Bitstream is a vector consists of uniform distributed binary data
e.g. 8bit_stream = 0 1 0 0 1 1 0 1 .

2 Counting Zeros

Counting Zeros is a simplistic algorithm we have implemented which can count the trailing zeros of a binary bitstream. It is not a very significant algorithm although, it helps us to understand in a initial stage the low-level algorithms we had to develop.

The Trailing Zeros algorithm just count the last zeros of a bitstream after the last one. So with that mini-algorithm we can estimate the index of the last one in a binary-vector(bitstream). That fact is helping us to evaluate the way of thinking around hashing* and bitstreams.

*A hash_function is a function that takes input of a variable length sequence of bytes and converts it to a fixed length sequence. It is a one way function.

Trailing_Zeros.py

```
# Function for counting the last zeros from the right of a bitstream
def trailing_zeroes(num):
    bit_length = len(num)
    if 1 in num:
        i = 0
        while num[-(i+1)] == 0 and i < bit_length:
            i += 1
    else:
        print ("The bitstream does not include a ’1’")

    return i
##########################################################
```

3 Probabilistic Counting

3.1 Algorithm

The idea behind the Flajolet-Martin Algorithm (Probabilistic Counting) is that the more different elements we see in the stream, the more different hash-values we shall use. As we see more different hash-values, it becomes more likely that one of these values will be "unusual." The particular unusual property we shall exploit is that the value ends in many 0's, although many other options exist.

3.2 Implementation

In this experiment we implement the Probabilistic Counting Algorithm using 'fake data'(noise) simulating hash_function results. More precisely we generated multiple vectors consist of binary uniform distributed data (0,1) in order to test the algorithm on them. We followed the steps below:

1. Generate binary data for the input.
2. Choose the number of bits(length) in the bitstream e.g 32bit.
3. Define k = number of bits which are used for the bucket_id.
4. Initialize an empty matrix filled with zeros which called bitmap. The real dimension of that matrix must be equal to $2^{bits\_for\_bucket\_id}$ rows and $bits\_for\_bucket\_id$ - bitstream_length.
5. Check in the first bitstream where the first "1" occurs and store the specific index.
6. Assign the bitmap matrix in the same index with "1".
7. Repeat stage 5 and 6 until you scan all the different bitstreams.
8. In the end the bitmap matrix will be like this:

1	1	1	1	0	1
1	1	0	0	0	0
1	1	1	1	1	0
1	1	1	1	1	1
1	1	1	0	1	0

   The above table is an example of a bitmap matrix which shows us where were the first "1" occurred in each bitstream and from that we can conclude how many unique elements we had, by adding the ones until the first zero in each row of bitmap, then calculating the mean and then estimate the unique elements with: 2^{mean}.

9. After the hole scanning and matrix assignment, calculate the mean and implement the DV value with

$$DV = \frac{2^k}{\phi 2^{\bar{R}}}$$

   , where ϕ=0.77351 constant.

```
## DV calculation in python
DV=b/phi * 2**(np.mean(R))
```

10. Calculate the final error accuracy with:

$$Error = \frac{|truth_{value} - predicted_{value}|}{truth_{value}}$$

11. We can now compare the actual error we have calculated with the Estimated Error $\alpha = 0.78/\sqrt{2^k}$.

Probabilistic Counting Pseudocode

```
Input: a stream S; Output: cardinality |S|
For each x in S do:
Set BITMAP[p(HASH(w))] := 1;  od;
Return P where P is position of the first 0
Split stream: e.g. , S --> (S....Sn) , for m = 8
Work out each Pj := P(Sj) separately
Let Ave := 1/m [(P1+.....Pm)];
Return DV;
########################################
```

4 LogLog Counting

4.1 Algorithm

The idea behind LogLog algorithm is to extend and optimize the probabilistic counting in respect to accuracy and RAM memory usage. The algorithm was discovered and published by Duran and Flajolet at 2003 with this paper. Its goal is that we can estimate bigger amount of distinct elements with better accuracy using fewer memory(RAM).

4.2 Implementation

In this experiment we implemented the Log log Algorithm we created a matrix of random binary number. More precisely we generated multiple vectors consist of binary uniform distributed data (0,1) in order to test the algorithm on them. We followed the steps below:

1. Generate binary data for input.
2. Choose the number of bits(length) in the bitstream e.g 32bit.
3. Define k = number of bits which are used for the bucket_id.
4. Initialize an empty list filled with zeros which called bitmap. The real dimension of that list must be equal to $2^{bits\_for\_bucket\_id}$ rows and one column which represent the highest number of zeroes, actually we estimate the number of trailing zeroes and we assign the highest number, like:

Bucket_id	Max num trailing zeros
1	9
2	5
3	6
4	10

   The above table is an example of Loglog algorithm bitmap.The 2nd column represent the highest number of tailing zeroes of each hash function. In the first column is the number of hash. We calculate the number of hash by converting the binary numbers of buckets into decimals. For example if we give 5 buckets of the bitstream to calculate the hash: Binary:00001 $\rightarrow$ Decimal:1

5. In Loglog calculate the distinct element of the bitmap(DV) like:

   $$DV = \phi 2^k 2^{\bar{R}}$$

   , where ϕ=0.79402 constant.

6. Standard Error Accuracy is : $\alpha = \frac{1.30}{\sqrt{2^k}}$, where k=number of bits for bucket_id

```
# LogLog Counting PseudoCode (The principal of basic LogLog)
Initialize M^1....M^m to 0;
let p(y) be the rank of the first 1-bit from the left in y;
FOR X = b1,b2... in M
DO set j:={b1...bk} (value of the first k bits in base 2)
set M(j) := MAX(M[j] , p(b[k+1] b[k+2]...);
RETURN E := a[m] * m * 2^((1/m) * SUM(M[j])  as the cardinality estimate.
################################################################################
```

5 Results

In this section we evaluate the differences between the two algorithms(Probabilistic and LogLog counting). The matrices below shows the results of accuracy errors in different tries with different number of bits and values.

5.1 Error Matrices

Probabilistic Counting Results

bits_bucketid	error(25bits)	error(32bits)	error(64bits)	alpha
4	0.05	0.31	0.22	0.20
5	0.07	0.42	0.10	0.14
6	0.04	0.09	0.00	0.97
7	0.10	0.03	0.08	0.07
8	0.45	0.09	0.002	0.05
9	0.0126	0.03	0.02	0.03
10	0.0047	0.03	0.02	0.02
11	0.0038	0.03	0.004	0.02
12	0.0015	0.01	0.003	0.01
13	0.0045	0.01	0.001	0.01
14	0.0017	0.01	0.007	0.01

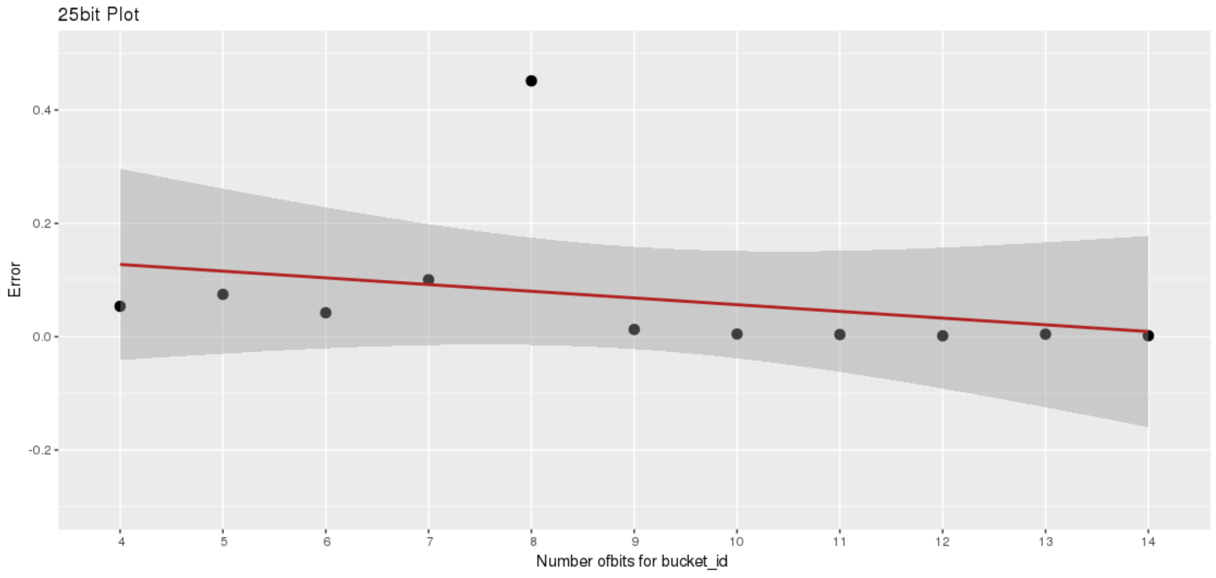
LogLog Counting Results

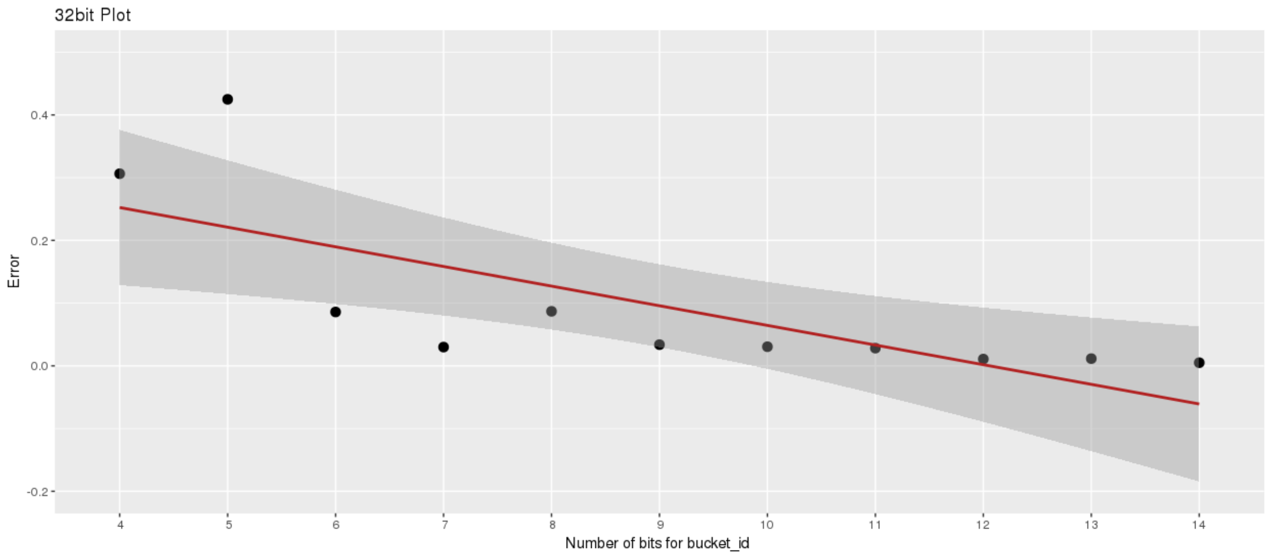
bits_bucketid	error(25bits)	error(32bits)	error(64bits)	alpha
2	0.56	0.70	0.49	0.65
3	0.48	0.28	0.35	0.46
4	0.23	0.02	0.40	0.33
5	0.34	0.31	0.18	0.23
6	0.19	0.27	0.12	0.16
7	0.14	0.05	0.21	0.11
8	0.041	0.03	0.10	0.08
9	0.022	0.06	0.01	0.06
10	0.05	0.01	0.02	0.04
11	0.022	0.03	0.02	0.03
12	0.04	0.02	0.03	0.02
13	0.022	0.007	0.02	0.01
14	0.013	0.005	0.006	0.01

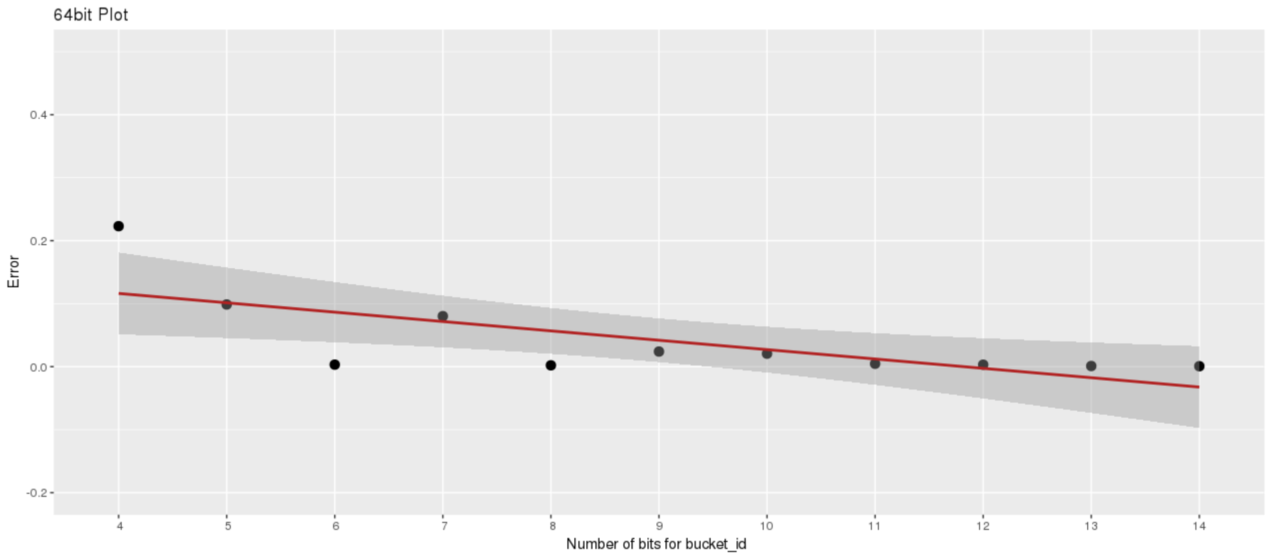
5.2 Plots

The plots below provides information about the different error values between different number of bits for the bucket_id. We have tested each algorithm on 25, 32 and 64 bit streams which were all generated randomly and uniform distributed.

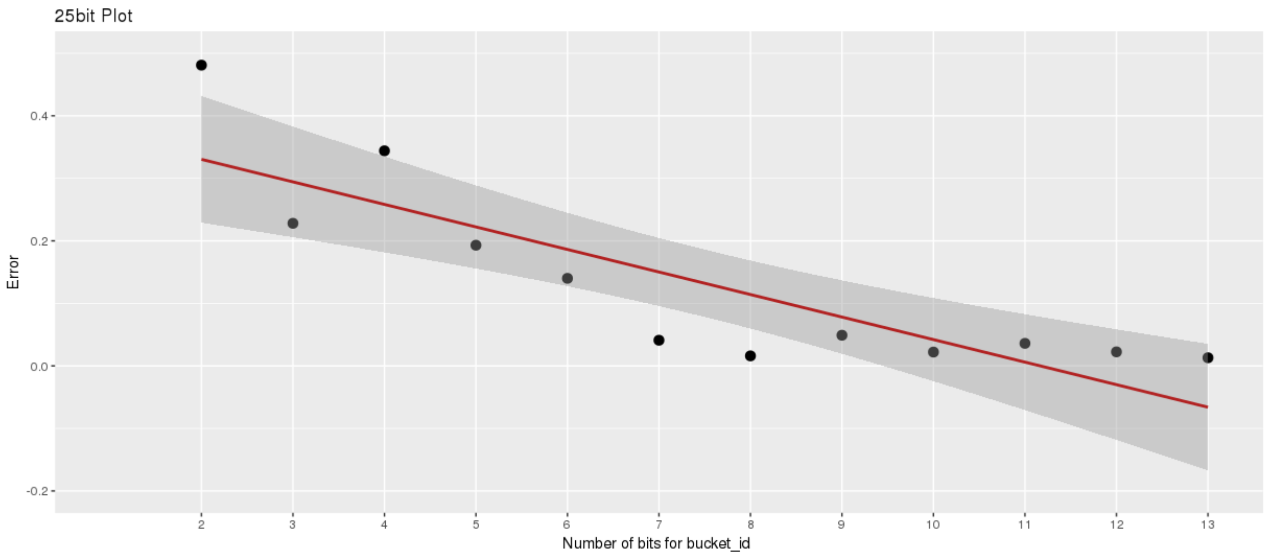
Probabilistic Counting Plots

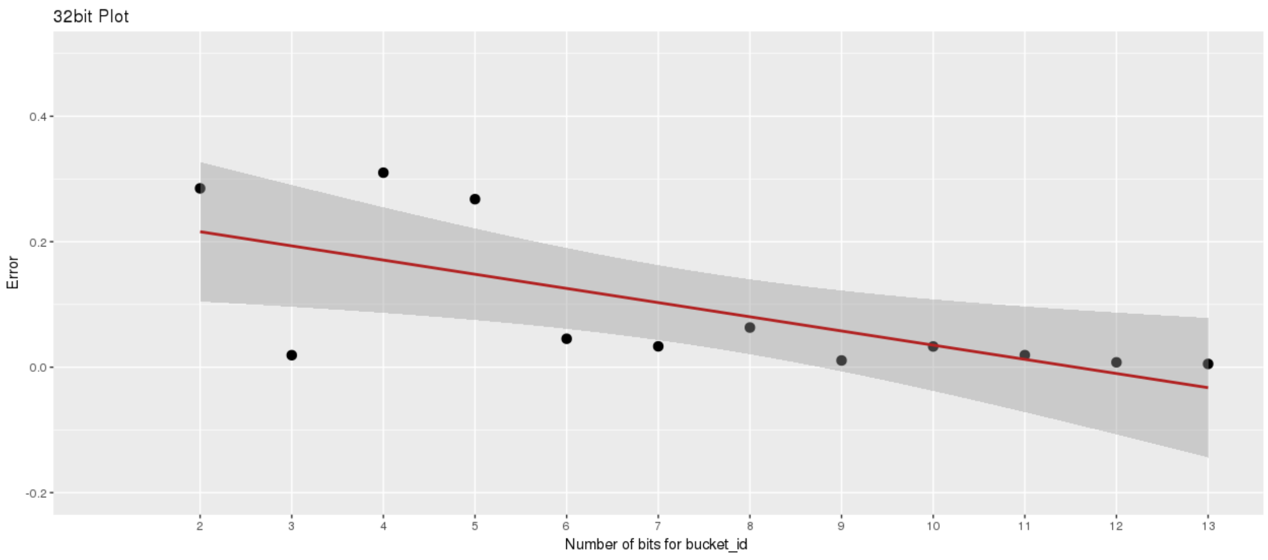


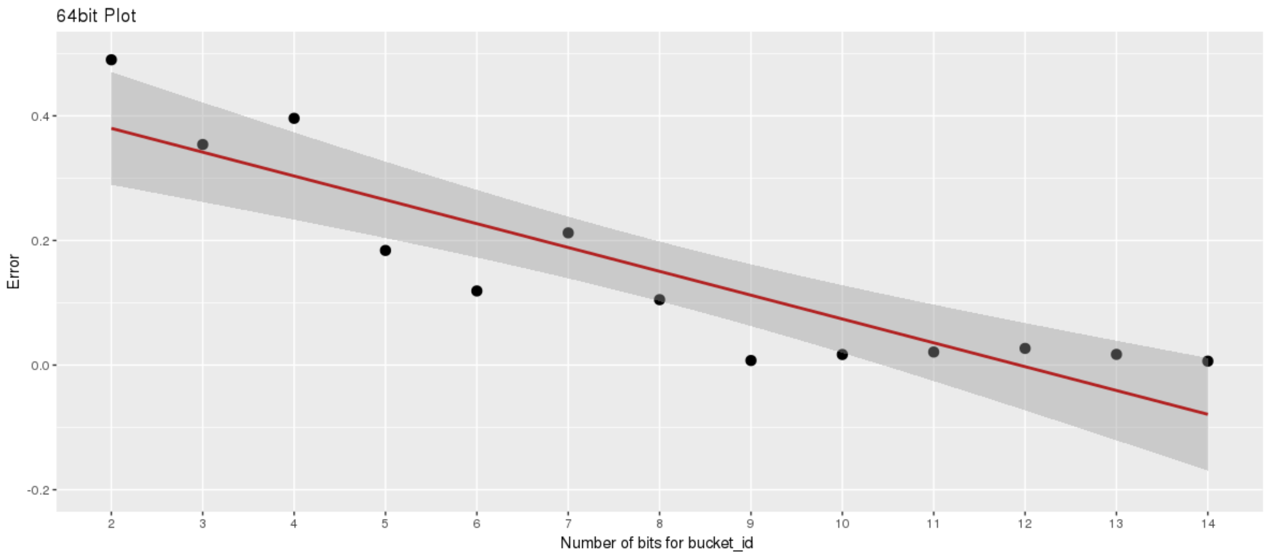




LogLog Counting Plots







6 Conclusion

It can be easily observed that the LogLog Counting gives much better output in respect to accuracy. The reason behind it is that one random occurrence of high frequency 0-prefix element can spoil everything. One way to improve it is to use many hash functions, count the maximum for each one and in the end average them out. This is an excellent idea, which will improve the estimation, but LogLog algorithm used a slightly different approach (probably because hashing is kind of expensive).

We can see at the matrices in 5.1 that the values are very close to α which is the estimated error for each algorithm. The basic advantage of LogLog counting is that it is not very "expensive" concerning to its implementation instead of Probabilistic counting because LogLog uses fewer memory(buckets) and estimates much better the unique elements. So with the fewer amount of memory, using LogLog Counting, we can make better predictions.

6.1 Application Example

The example below describes how Probabilistic algorithms are used by companies in order to count distinct elements.

Counting the number of distinct elements (the cardinality) of a set is challenge when the cardinality of the set is large.

To better understand the challenge of determining the cardinality of large sets imagine that you have a 16 character ID and you'd like to count the number of distinct IDs that you've seen in your logs. Here is an example: 4f67bfc603106cb2

These 16 characters represent 128 bits. 65K IDs would require 1 megabyte of space. Clearspring receives over 3 billion events per day, and each event has an ID. Those IDs require 384,000,000,000 bits or 45 gigabytes of storage. And that is just the space that the ID field requires! To get the cardinality of IDs in daily events we could take a simplistic approach. The most straightforward idea is to use an in memory hash set that contains the unique list of IDs seen in the input files. Even if we assume that only 1 in 3 records are unique the hash set would still take 119 gigs of RAM. You would need a machine with several hundred gigs of memory to count distinct elements this way and that is only to count a single day's worth of unique IDs. The problem only gets more difficult if we want to count weeks or months of data.

One common approach to this problem is the use of bitmaps. Bitmaps can be used to quickly and accurately get the cardinality of a given input. The basic idea with a bitmap is mapping the input dataset to a bit field using a hash function where each input element uniquely maps to one of the bits in the field. This produces zero collisions, and reduces the space required to count each unique element to 1 bit. While bitmaps drastically reduce the space requirements from the naive set implementation described above they are still problematic when the cardinality is very high and/or you have a very large number of different sets to count. For example, if we want to count to one billion using a bitmap you will need one billion bits, or roughly 120 megabytes for each counter. Sparse bitmaps can be compressed in order to gain space efficiency, but that is not always helpful.

*The above example was taken from this link