This is the presentation of the paper "Lossless preprocessing of floating point data to enhance compression", presented at DCAI conference in Guimarães, Portugal, in July 2023.

Suppose to have a series of floating point numbers, here represented by differently colored rectangles, and that our task is to compress this data, namely to fit the dataset in a smaller package, possibly for more efficient storage.

After compression, we want to be able to recover the original data, in order to use them for their original purpose. However, it is often the case that recovered data is not really exactly the same as the original one, but slightly “skewed”. This error is usually defined as “loss”, and compression processes that introduce these errors are called “lossy”.

For enhancing compression, namely to get a smaller compressed package size, lossy compression algorithms tend to result in better performances, where the price is the skew in the recovered data. This might be a problem, depending on the users’ specifications.

Most of the time, before compressing the data, they go through a preprocessing stage, where, as for the example, if we want them to fit a smaller triangle, they have to be transformed somehow, and then be compressed.

We say that a transformation is “lossless” when, after both preprocessing and compression, the decompressed data are exactly the same as the original, without any error being introduced.

Let’s also introduce the concept of “deduplication” which will be important later.

One possible way of compressing data is to identify common patterns in their binary representation, then building a dictionary where we associate the pattern to a smaller ID, which substitutes the pattern portion on the datum. In this way, as long as we choose wisely what is to be stored as pattern and what isn’t, we achieve compression. The best pattern, naturally, is a sequence of bit that is shared by all data in the dataset, as per the example.

Let’s also take a look at other common preprocessing techniques, by considering a small dataset of floating point data, on the left, with their binary representation on the right. Here, we use the 16 bits representation for clearness purposes, although the most common formats are 32 or 64 bits long. The different colors represent different portion of the binary representation of the number, namely sign, exponent and mantissa.

If they were all the same number, it would be very easy to deduplicate them and compress them a lot, but of course, we’d lose all useful information.

We see that these numbers all have a lot of zeros decimals, therefore one straightforward idea to make them more compressible could be to remove decimals. However, we see that the binary representation doesn’t change, therefore causing no effect on compressibility. Moreover, if there were decimals approximated out during this process, we would have incurred in losses, since part of the information in the number would be lost.

Another common technique is to remove decimals completely by using a multiplication by a power of 10. In this case, for 2 decimals, we need to multiply by 100. We see some modifications in the binary representation that might result in better patterns for compression by deduplication, but no obvious major benefit, so let’s go back to the original numbers.

In a previous paper, we introduced a technique called “Addition method”, where by summing a number “A” to all datapoint we are guaranteed to have some shared values in the mantissa and exponent of each number. Here, for example, we use A = 1800.0, and obtain shared exponent (in red) and some shared mantissa bits (in green), making the data definitely more compressible. The catch is that this process in lossy, meaning that the recovered data will be slightly skewed with respect to the original.

Since some of the previous floating point manipulations weren’t very effective in producing good patterns in the binary form, a common strategy is to cast the data to integer after having removed the decimals. This technique brings benefits in terms of compressibility, since we can immediately see some common binary portions appearing, but comes with its disadvantages. The main one is the fact that the range of feasible representable numbers is highly reduced, meaning that depending on the dataset, we might not be able to apply this technique. Moreover, a naïve floating point multiplication is not always losseless.

In this paper, we propose 4 lossless preprocessing techniques, that lead to common mantissa and exponents bits on the resulting data, which implies improved compressibility of the data. This means that, by using these algorithms, we improve compression only at the expense of processing time and power, since we do not introduce distortions like other preprocessing algorithms. We show the concepts for a single exponent region, namely between two powers of 2, but they can be easily generalized to multiple exponent regions.

The first one is called “Compact Bins”. Suppose to have the data of the dataset in this region on the real line of numbers, and that we want all of them to be guaranteed to share the first 2 mantissa bits, namely m1 and m2. The idea then is to separate the numbers into bins, shifting them closer together so to make them all belong to the region where m1= 1 and m2=1. In the paper, we show that this sequence of addition can be devised to be lossless and reversible, by storing some metadata on the operations.

The second one is called “Multiply and Shift”. Looking at the real line, we first identify the green regions where, like the previous method, we have some mantissa bits to be as we want. Given a dataset, in this case with numbers between 8 and 16, we aim at applying operations on it so that it is partially in a green area. What doesn’t fit in the area, gets pushed to next one via the combination of a lossless multiplication and a lossless addition, until all portions of the transformed dataset belong to the green areas, guaranteeing that all preprocessed number shared the wanted mantissa bits, which improves compressibility.

In order to go revert the process, we need some metadata on the values being used, and the inverse operations. We are therefore able to get back to the original dataset, with no loss.

One of the issues with this method is that we prove in the paper that only multiplication by factors greater than 2.0 are guaranteed to be lossless. Therefore, since we need to use at least 2.0 as a factor, for every jump to the next region, the range of values in the dataset that needs to fit in the green region gets wider. The next method tries to limit this effect, by substituting the multiplication for an addition.

It is called “Shift and Separate Even from Odd”, where the jump between one exponent region and the next one is achieved via an addition. In order to be lossless, we have to use different shift values depending on whether the mantissa being process is “odd”, namely with least significant mantissa bit = 1, or “even”, where that bit is 0. In order to be able to reverse losslessly these operations, we have to make the output of these operations sit in different portions of the real line, so be able to tell, for the difference, whether to use the odd or the even addendum. Like for the previous techniques, we are able to go back to the original dataset with no loss, by storing some additional info as metadata, like the addendi being used.

This however keeps showing an increase in the range of the output dataset since we need to separate the output portion of the odd and the even numbers, which increases the numbers of steps required to end the algorithm. In order to avoid this, we introduce the 4th and last method called “Shift and Save Eveness”, where we differentiate whether to use odd or even addendum during recovery by storing a bit for every value for every processing step. On one hand, this substantially increases the size of the metadata required for the algorithm to work, but reduced as lot the steps needed to terminate it as well, creating a tradeoff.

To conclude this presentation, we show the results of the introduced techniques.

The main achievement is that, for the analyzed dataset and considering a compressor based on deduplication, we were always able to find a configuration of the introduced techniques achieving improvements in compression, expressed in the form a reduced compression ratio, here represented in relative forms. Plotting the best setups for each technique, we see that we were able to achieve up to a 40% reduction in the compression ratio, meaning that we were able to compress 40% more effectively with the preprocessed dataset than the original one. We also show the results for another less successful dataset in order to show that even in that case we were able to improve its compressibility, with no real cost for the user but processing time and power.

Thanks!