A Compression Generation Tool

A Technical Report
presented to the faculty of the
School of Engineering and Applied Science
University of Virginia

by

Nathaniel Saxe

with

Nathan Brunelle
Yonathan Fisseha

May 11, 2020

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Signed: ____

Approved: _______________________________________ Date ________________________
Nathan Brunelle, Department of Computer Science

A Compression Generation Tool

Nathaniel Saxe

Motivation

When developing compression-aware algorithms, it is useful to know the properties of compression itself to see if they can be exploited. In particular, we were attempting to create a compression-aware string metric algorithm, and wanted to know which conventional string metric could most leverage compression.

The effect of two strings' compressibility on various string metrics was measured. In theory, if the distribution of a metric score when evaluated on pairs of highly-compressible strings was more bimodal or otherwise non-normal than it was for less-compressible strings, then the metric might be able to better take advantage of compression. For example, if there is a set of low-scoring strings mostly separated from a set of high-scoring strings, then the strings and their respective compressions in each set could have some trait in common which can be used to discriminate the sets from each other.

Process

The desired functionality was to input a string length l and a compression ratio r, and receive a string of exact length l which, when compressed using LZ-78, would have approximate length l/r. The compression ratio of a string is an emergent property of the string, and not one which can be easily adjusted, so the first step was finding some tunable parameter which could be used when generating strings and correlated to the compression ratio of the result. Explaining the choice of parameter requires an overview of LZ-78.

An LZ-78 compression is a list of tokens.

For example the LZ-78 compression of the string “bananasavanna” is

(0, ‘b’), (0, ‘a’), (0, ‘n’), (2, ‘n’), (2, ‘s’), (2, ‘v’), (4, ‘n’), (2, ‘’)

Each token *t* in the compression contains an integer representing the index of a previous token which *t* extends, and a character which *t* extends the previous token by. Token 0 is assumed to be the empty string ‘’. Going token by token, this compression would be decoded as

“b”, “a”, “n”, “an”, “as”, “av”, “ann”, “a”

yielding the original string when concatenated together.

Typically, tokens are larger near the end of the compression, because they can build off previous tokens. If a token builds off an immediately preceding token, it will generally expand to more letters in the string than if it had built off a token near the beginning of the compression, slightly increasing the compression ratio of the string. This difference between a token’s position within the compression and the index of the token it extends will be referred to as its *index diff*, and can be used as a proxy for compression ratio.

The first attempt at generating a string with a desired compression ratio was to generate a random compression with a constant index diff for every token, using logic like this:

```
def same_index_diff_compression(num_tokens,index_diff):
    result = []
    for i in range(num_tokens):
        index_to_use = i - index_diff
        result.append((index_to_use, letters[random.randrange(0,len(letters))]))
    return result
```

Here, “letters” is an arbitrary alphabet of symbols which can be included in strings. After generating the compression, it can be decompressed to yield a string.

There are a couple problems with this method:

- The indices in each token are always the same for a given value of index_diff. This means the only difference between strings generated is that the characters are swapped

around randomly; the string is "structurally" nearly identical. This bias can be corrected for by slightly randomly perturbing the index diff for each token.

- The calculated index may be outside the legal range for a pointer to a previous token, leading to a malformed compression. This can be fixed by clamping the value to the legal range (0,i).

The resulting logic after fixing these holes looked like this:

```
def same_index_diff_compression(num_tokens,index_diff,perturb_amount):
    result = []
    for i in range(num_tokens):
        index_to_use = i - index_diff + \
        random.randrange(-perturb_amount,perturb_amount+1)
        index_to_use = max(0,min(i,index_to_use))
        result.append((index_to_use, letters[random.randrange(0,len(letters))]))
    return result
```

This could be used to generate compressions of roughly similar compression ratio, but the relationship between the input index diff parameter and the resulting compression ratio was still uncertain. Plotting index diff vs compression ratio for 500-token compressions revealed this relationship (Fig. 1).

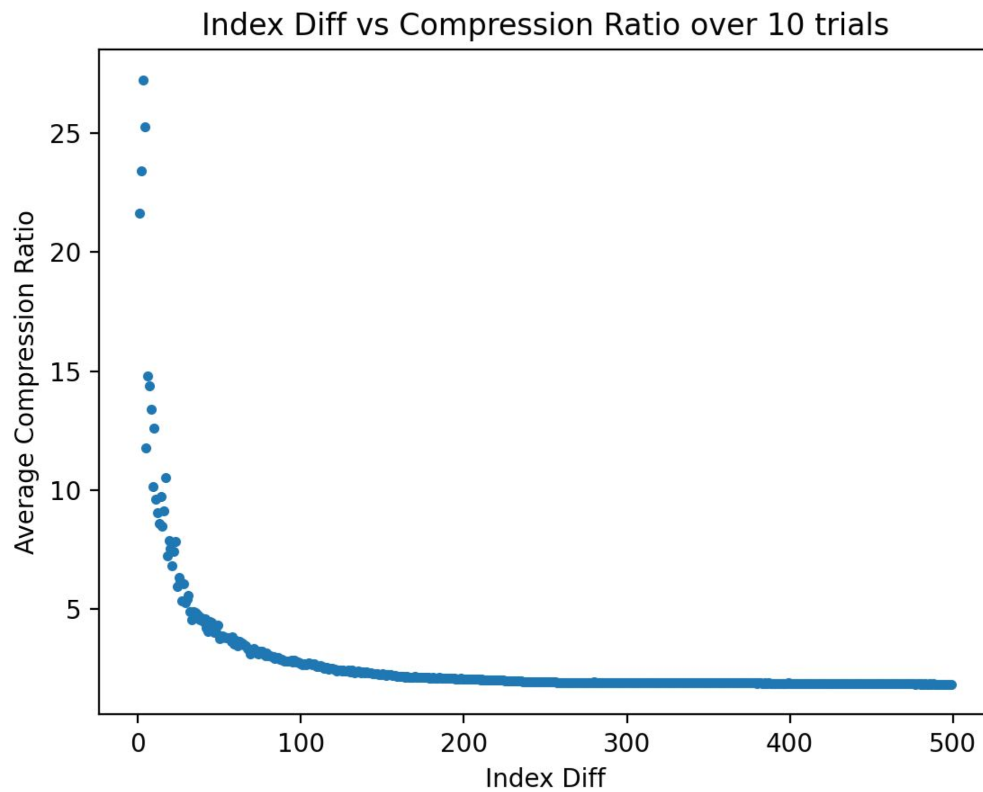


Figure 1. Index Diff vs Mean Compression Ratio over 10 trials.

In order to treat compression ratio as an input rather than an output, the relationship would need to be modeled and then inverted to yield a function mapping a target compression ratio to a desired index diff. Luckily, the relationship seemed to be both basically smooth and strictly decreasing (and therefore invertible).

The index diff axis was normalized relative to the total number of tokens so that the relationship could be meaningfully compared even with different numbers of tokens (Fig. 2)

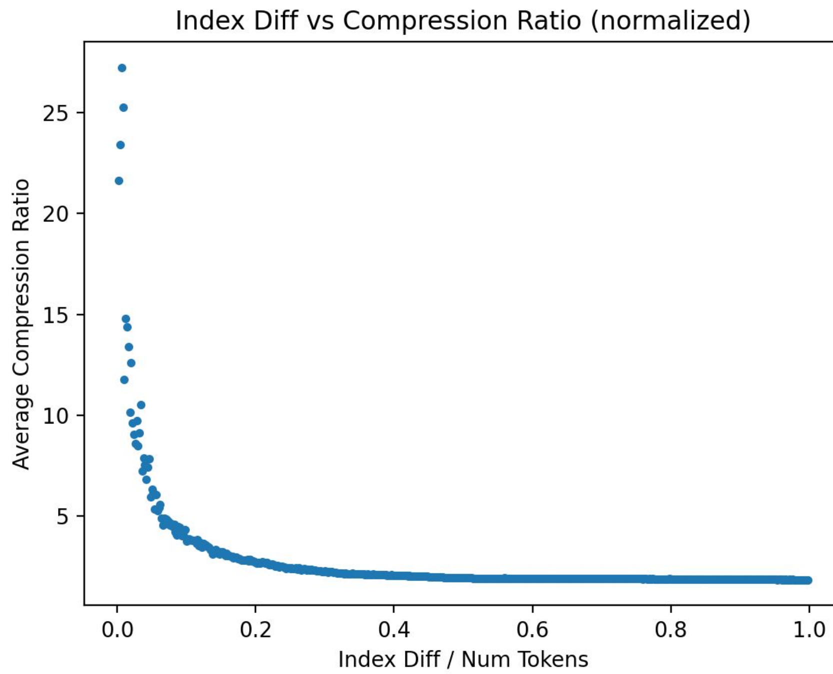


Figure 2. Index Diff vs Compression Ratio (normalized to number of tokens)

After some experimentation with this graph, the compression ratio was determined to be roughly proportional to the inverse square root of the index diff, that is $CR \approx (k * diff)^{-\frac{1}{2}}$ for some k. However, there were unexpected knees in the graph (Fig 3).

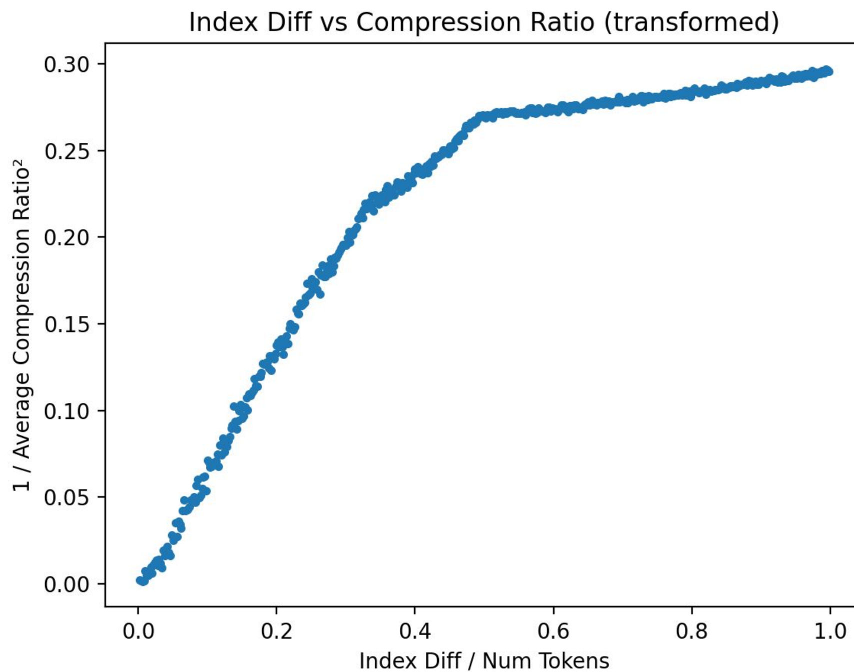


Figure 3. Index Diff vs Compression Ratio (transformed).

The reason for the appearance of the knees is not yet understood. Their exact placement and slope seems to depend on both the number of tokens and the number of characters in the alphabet used. Regardless of their origin, accounting for the knees was not difficult. Instead of assuming a linear relationship, the model now assumed a piecewise linear relationship with index diff: $CR \approx (pwlf(diff))^{-\frac{1}{2}}$ for some continuous piecewise linear function $pwlf$.

Performing a basic linear regression is a trivial statistical technique. Performing a regression over a multiple-segment piecewise linear function is substantially less so, because the model parameters include the start and end intervals for each segment in addition to the slope

and offset for the segment. A simple solution would be to eyeball the breakpoints manually and perform a linear regression on each of several predefined line segments. But this would require a human to tweak parameters for every regression, leading to imprecision.

Instead, a specialized python library was employed for determining optimal breakpoints for a given number of line segments using an evolutionary algorithm called differential evolution[1]. Although the running time of this algorithm grows quickly with the number of line segments, 3 or 4 segments sufficed for the purposes of this model (Fig 4).

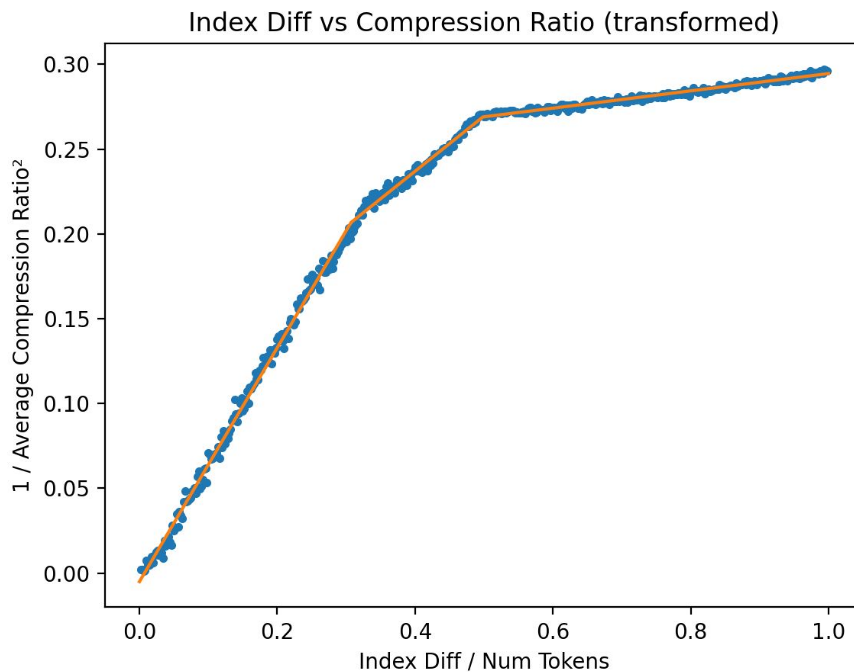


Figure 4. Piecewise linear fit of transformed data

This model accurately translates an index diff to a compression ratio. However, we care about the inverse problem of translating a target compression ratio to an index diff. We can derive the formula for the inverse using simple algebra:

$$CR \approx (pwlf(diff))^{-\frac{1}{2}}$$

$$CR^{-2} \approx pwlf(diff)$$

$$diff \approx pwlf^{-1}(CR^{-2})$$

$pwlf^{-1}$ is the inverse of the piecewise linear function fit to the data. The following logic was written to return the inverse of a given piecewise linear function:

```
def piecewiseInverse(fit):
    for slope in fit.slopes:
        if(slope <= 0):
            print("fit is not strictly increasing, so I can't invert it")
            return
    newBreakpoints = fit.predict(fit.fit_breaks)
    newBreakpointYs = fit.fit_breaks
    newSlopes = np.reciprocal(fit.slopes)
    def invertedFit(x):
        return piecewiseEvaluate(x, newBreakpoints, newBreakpointYs, newSlopes)
    return invertedFit
```

Not every piecewise linear function has an inverse. The function must be strictly increasing everywhere to be invertible, hence the first check. Since the original compression ratio data was strictly decreasing, the transformed version ends up strictly increasing, so this check has never failed thus far. After transforming the piecewise data back, the model closely approximates the original index diff data (Fig. 5).

Finally, the model for translating a compression ratio to an index diff was derivable for a given number of tokens:

```
def deriveCRToDiff(num_tokens):
    data = generateData(num_tokens) #generate same data as in Fig. 3
    piecewiseFit = findPiecewiseFit(data) #fit the data using pwlf library
    invertedFit = piecewiseInverse(piecewiseFit) #invert the piecewise model
    #resulting function should take in a compression ratio, transform it,
    #and call invertedFit on it to yield an index diff
    def result(compression_ratio): #resulting function
        return invertedFit(1 / (compression_ratio * compression_ratio)) * num_tokens
    return result
```

This was used to construct a function similar, but not identical, to the ultimate goal:

```
#random compression by num tokens
def random_compression_tokens(compression_ratio, num_tokens=500):
    CRToDiff = deriveCRToDiff(num_tokens)
    avgIndexDiff = CRToDiff(compression_ratio)
    return same_index_diff_compression(num_tokens, avgIndexDiff)
```

One hole with this scheme is that the CRToDiff function is real valued and returns a desired average index diff, while index diff itself is an integer. The same_index_diff_compression function was modified to handle this:

```
def same_index_diff_compression(num_tokens,index_diff,perturb_amount):
    result = []
    for i in range(num_tokens):
        chanceToLower = index_diff % 1
        index_diff_to_use = int(index_diff)
        if(random.random() > chanceToLower):
            index_diff_to_use += 1
        index_to_use = i - index_diff_to_use + \
        random.randrange(-perturb_amount,perturb_amount+1)
        index_to_use = max(0,min(i,index_to_use))
        result.append((index_to_use, letters[random.randrange(0,len(letters))]))
    return result
```

This way, if the target index diff is some integer k plus some fraction f, each token will choose an index diff of k with probability f, or else $k+1$ with probability $1-f$ (before random perturbation). The average index diff used over the whole compression therefore approaches the target value, in a kind of linear interpolation.

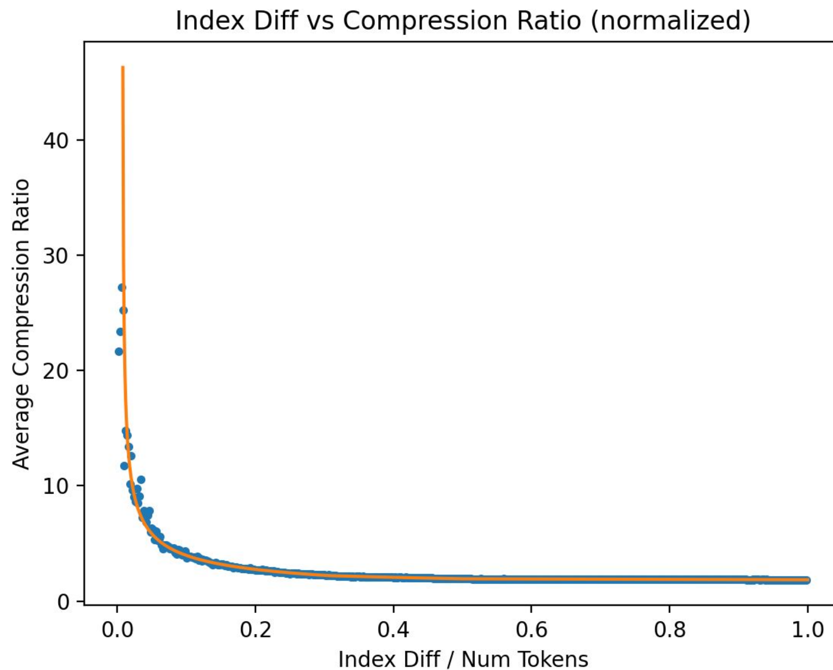


Figure 5. Final Index Diff vs Compression Ratio model.

There was one more obstacle to overcome before the desired functionality was satisfied. The current method took in a desired number of tokens in the compression, not a desired number of characters in the resulting decompressed string. Holding string length constant is important for measurements involving string metrics, because otherwise variance in the sizes of strings could bias the metric scores. For example, the Hamming Distance of two strings, usually defined as the number of dissimilar bits between them, is heavily dependent on the length of the strings in question. So there needed to be some way to take these token-based compressions and impose a strict bound on the length of the resulting string.

The solution was fairly simple: for a given string length l and compression ratio r,

- Figure out how many tokens "should be" in the compression (l/r tokens)
- Generate a compression with slightly more tokens than that (one which will definitely expand to more than l characters when decompressed)

- Decompress it and take the first l characters of the resulting string
- Hope that chopping off the last few characters did not affect the compression ratio too much

In code:

```
def truncated_compression(compression_ratio, num_tokens, string_size):
    comp = random_compression_tokens(compression_ratio, num_tokens)
    decomp = decompress(comp)
    if(len(decomp)>string_size):
        comp = compress(decomp[:string_size])
    return comp

def random_compression(compression_ratio, string_size=2000):
    num_tokens = int(round(string_size / compression_ratio)) + 25
    return truncated_compression(compression_ratio, num_tokens, string_size)
```

This random_compression function was the one used to generate compressions and strings for the rest of the project.

Speed

At first, this compression generation method was very slow. By far the biggest reason for this is the deriveCRToDiff function, which derives a model translating compression ratio to index diff for a given number of tokens. It was slow because every time it was called it needed to generate hundreds of compressions each hundreds of tokens long, measure their compression ratios, and fit a linear piecewise function to the data. That's no small task in python.

However, once derived, the models were not expected to change between any two runs of the program. The data had small enough variance that the parameters of the piecewise functions hardly changed for a given value of num_tokens no matter how many times the model was re-derived. Therefore, some basic memoization was applied where, upon deriving a new model, the function would store it into a dictionary lasting as long as the program execution, as well as

writing the fit parameters to a file in a format where they could be retrieved and repackaged into a function later.

After a single run in which a bunch of models were generated fresh, this drastically improved runtime. Everything else in the generation was comparatively breezy.

However, generating a new model for each individual value of num_tokens would have still incurred a painfully large one-time cost. To work around this, the logic for choosing a "slightly larger" number of tokens for random_compression was reworked. Instead of simply adding a constant amount to the number of tokens needed, it picked the nearest multiple of some constant (I chose 100) above the "right number" of tokens. This way only 1 in every 100 possible models would actually end up being derived and used.

Accuracy

After random_compression was finalized, tests were run to determine the skew between the theoretical and actual compression ratios of the strings generated. First, the skew was plotted for strings of length 2000, which we used in the string metric experiments (Fig. 6).

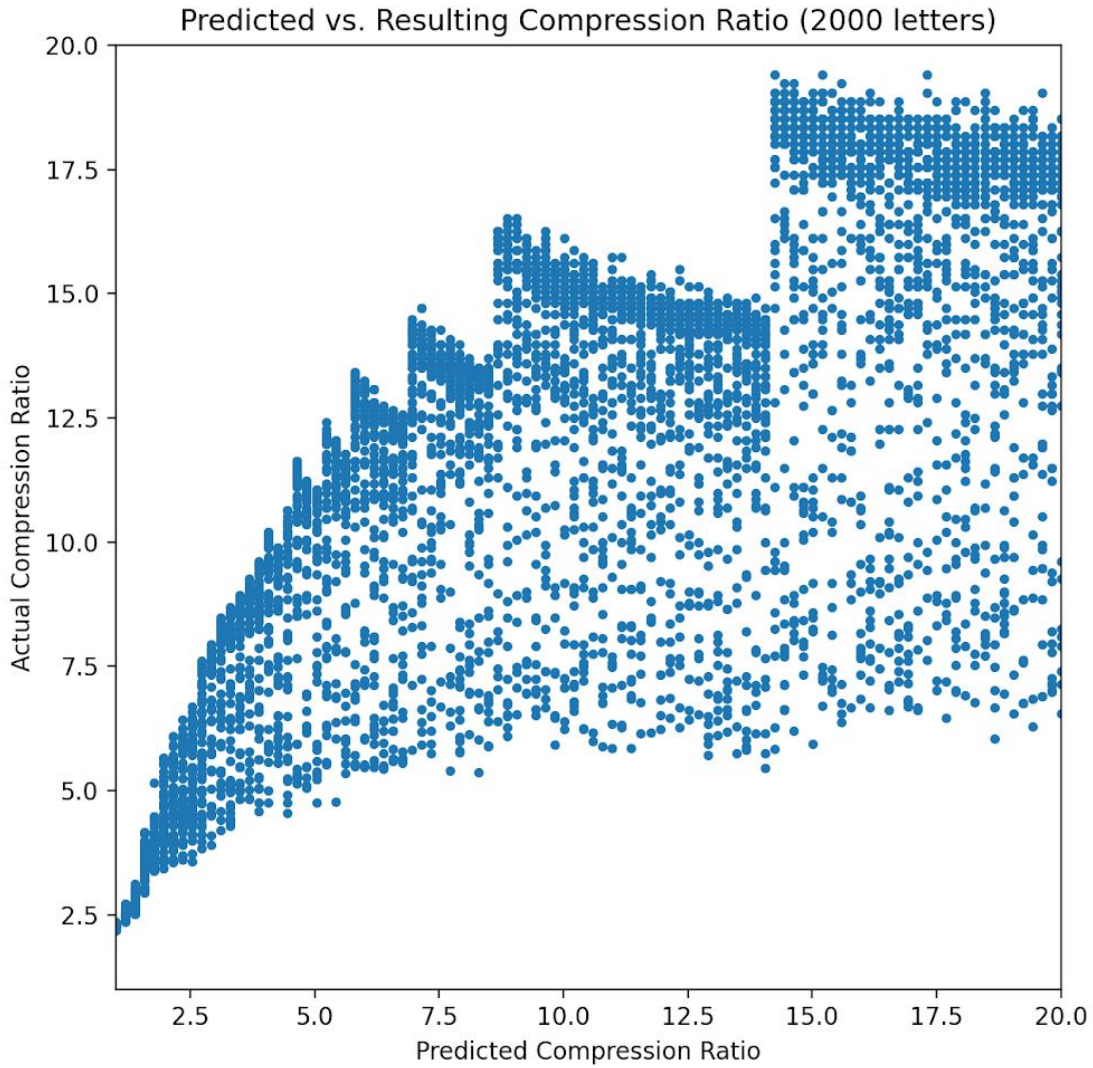
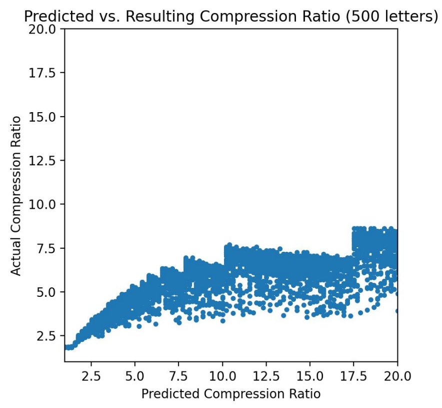
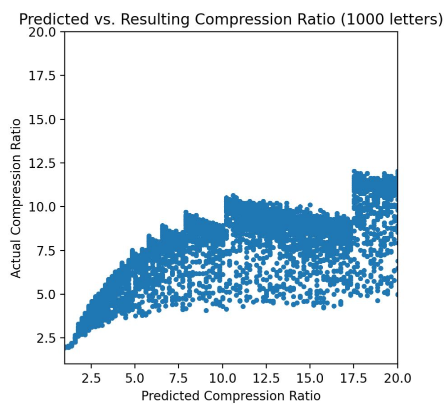
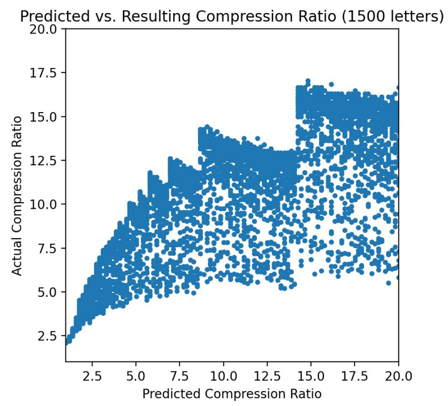


Figure 6. Skew Plot for strings of length 2000.

As can be seen, the skew is still considerable. However, it is heading in a generally positive direction, much better than generating completely random compressions


Predicted vs. Resulting Compression Ratio (500 letters)
Actual Compression Ratio
Predicted Compression Ratio
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0


Predicted vs. Resulting Compression Ratio (1000 letters)
Actual Compression Ratio
Predicted Compression Ratio
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0



References

[1] pwlf: piecewise linear fitting

https://pypi.org/project/pwlf/