Intelligent Text Compression

Stanford CS224N Custom Project

Ella Hofmann-Coyle
Department of Computer Science
Stanford University
ellahofm@stanford.edu

Mauricio Wulfovich
Department of Computer Science
Stanford University
mauriw@stanford.edu

Blake Pagon
Department of Computer Science
Stanford University
bpagon@stanford.edu

Abstract

We build a system that applies techniques used for the autocomplete task as described in Poesia et. al. [1] to the task of natural language text compression. We utilize models' learning of contextual language understanding to compress and decompress text in a way that achieves a greater compression rate than standard methods (i.e. gzip) with high accuracy (low information loss).

1 Key Information to include

- External collaborators (if you have any): N/A
- Mentor (custom project only): Gabriel Poesia
- Sharing project: No, we are not sharing this project with another class

2 Introduction

Compression and machine learning are tightly linked. A model that predicts the posterior probabilities of a sequence given its complete history can be used for optimal data compression. In the case of natural language, the surrounding context is very informative about which word is likely. Take a document containing the sentence, "The child came home from _", where the last word is masked with '_' to reduce the document size. It's easy to see that the next word has a high probability of being "school". However, outputs like "Ballet" are less likely but also probable, which would reduce our system's accuracy. So we can go a step further and rather than choosing the likely word from the set of all possible English words, we can reduce the space of possible compressed words. For instance, suppose the table of possible compression words only includes "language" and "logistics". Then for a sentence like "the course _ are explained on the website", it's quite clear that the sentence had the word "logistics" before compression.

In this paper, we develop a novel compression method to choose which words to mask. We also develop a compression method inspired by the one in [1]. We then develop a model that reads compressed documents and outputs the decompressed documents. We evaluate the tradeoff between compression and accuracy and measure the compression gains. We are interested in this topic because compression is essential to efficiently transmit information across the globe.

3 Related Work

There are a variety of papers that using machine learning models for text compression. Since we attempt a lossy compression approach, the most similar work to ours is "Pragmatic Code for Autocomplete," [1]. The authors note that in traditional auto-complete, showing users multiple possible predictions results in cognitive and efficiency costs. Therefore they propose using ambiguity in a controlled fashion to reduce users' typing effort. They develop a method for users to type code using one-character abbreviations. Their model expands it, using context, with high accuracy. The authors of the paper utilized these advancements to create a more efficient interface for typing code, improving normalized typing speed.

DeepZip Compressor [2], a lossless compression approach, utilizes Recurrent Neural Network (LSTM/GRU) models to capture long-term dependencies and predict the next word. The system does not store the model weights so as not to include them in the compressed size. Therefore, the decoder and encoder only perform a single pass (1 epoch) through the data. Their encoder model is exclusively initialized using a random seed, which is communicated to the decoder. So after the single epoch, both the encoder and decoder RNN have the same weights. Since their decoder simply performs a single pass, their model is a much worse predictor than the pre-trained BERT transformer we utilize. Likewise, since we perform lossy compression, we can compress the text more than DeepZip. Moreover, our best model's mispredictions still communicate the documents' original meanings, as discussed in our analysis.

4 Approach

We approach the problem by building an end-to-end encoder/decoder system. Given a document, our model generates a compression (at which point it computes the compression score) and then takes the compression and processes it with the decoder to generate the reconstructed decompressed document. To develop the most performant system, we have modularized the encode/decoder to experiment with several different compression/decompression strategies.

4.1 Compression Methods

- **Local: Naive Baseline Method**: Every 10 words, we replace the longest word with a compression token. We compress tokens by replacing them with an underscore ('_') in order to minimize file size, and simply update these tokens to the mask token before making decompression predictions.
- **Global: Modified Poesia et al. Method**: This approach was inspired by Poesia et al.'s method to choose compression candidates, however, it differs in that it does not abbreviate words to their first character. It instead simply replaces all compression candidates in the corpus with the mask token. Each word in the corpus is scored by multiplying the number of times the word occurs in the dataset by the length of the word. Formally, for every word w in dataset X tokenized into a list of words, w_i we calculate the score as such:

  $$w_{score} = (\sum_{w_i \in X} 1\{w_i == w\}) * len(w)$$

  We then select the N highest scoring words as the list of words our system compresses via masking. By selecting words that are the largest and most often to occur as words to mask, we are aligning our compression with the goal of reducing the output file size. We leave N as a hyper-parameter of our compression system.
- **Global: Updated TF-IDF Method**: In this method, we use a slightly modified version of the TF-IDF algorithm to score words. Our scoring is defined as follows

  $$w_{score} = (\sum_{w_i \in X} 1\{w_i == w\}) * IDF_w * len(w)$$

  where IDF_w is the inverse document frequency of word w defined as for document d in dataset X (note we are looking samples in our dataset rather all words in all samples of our dataset level):

  $$IDF_w = \log(\frac{1}{\sum_{d \in X} 1\{w \in d\}})$$

- **Preliminary Characterization of Global Compression Methods** Before running our system, we characterized our two global compression methods on the entire dataset. While an algorithm that selects on average a high number of tokens to compress/mask gets achieves better (smaller) compression size, this can come at the expense of accuracy. However a model that is more selective, compressing fewer tokens, may have higher decompression accuracy, will likely worse performance on the compression goal. We thus look at these two methods over-compression risk and under-compression risk. We define as a sample at risk of over-compression if $\geq 80\%$ of the tokens compressed. A sample is at risk of under compression if it has 0 compressed tokens. We define space saving as

$$1 - \frac{compressed\_size}{uncompressed\_size}$$

. We also calculate compression rate which is as defined as

$$\frac{compressed\_size}{uncompressed\_size}$$

The below table examines the two methods performance:

Table 1: Compressor Characterization

Method	Avg. Compression	# under-compressed	# over-compressed
Poesia	**254**	242	
Updated TF-IDF	425	**160**	

We see that the TF-IDF has significantly fewer over-compressed samples, but the Poesia method does significantly better when it comes to producing fewer under-compressed documents.

From the histograms in Fig. 1, we see this supported and that both methods achieve similar Gaussian distributions of number of compressions in a document. The primary difference is that the TF-IDF compression's distribution's mean is shifted to the right increasing the average distribution by approximately 1% and the number of non-compressed documents is higher by a factor of 2. *This indicates that Poesia compression may be a best choice to optimize compression where TF-IDF may the best choice for the goal of decompression accuracy*. When we analyzed the words appearing in our compressed list generated by both methods, we decided to constrain compressed words such that UNK tokens and tokens with fewer than 4 characters are not inlcuded in the outputted list.

4.2 Decompression Methods

- We frame decompression as masked language modeling, thus making the HuggingFace pre-trained BERT base cased model [3] a suitable choice for our baseline decompression model.
  - **Baseline Local Approach:** For windows of tokens of a set size (currently we use 10 tokens), we mask a word in that window, then pass the entirety of the masked text to the model to predict the masked tokens' decompression by adding an additional prediction head on top of the pretrained BERT model, as done by the HuggingFace BertForMaskedLM model [3]. Because the masked token could be anything in the vocabulary, our output size needs to be equal to the size of the entire vocabulary. We update the document to include the decompression of each token and return that as our decompression of the input. We calculate accuracy at this point by comparing the number of correctly decompressed tokens in a document over all masked/decompressed tokens.
  - **Global Approach:** With the global approach, we update our model architecture by applying a dense layer on top of BERT-generated embeddings. This dense layer has a modified output size that is equal to the number of tokens in the compression vocabulary. In order to produce predictions with this model, we first tokenize the document (using

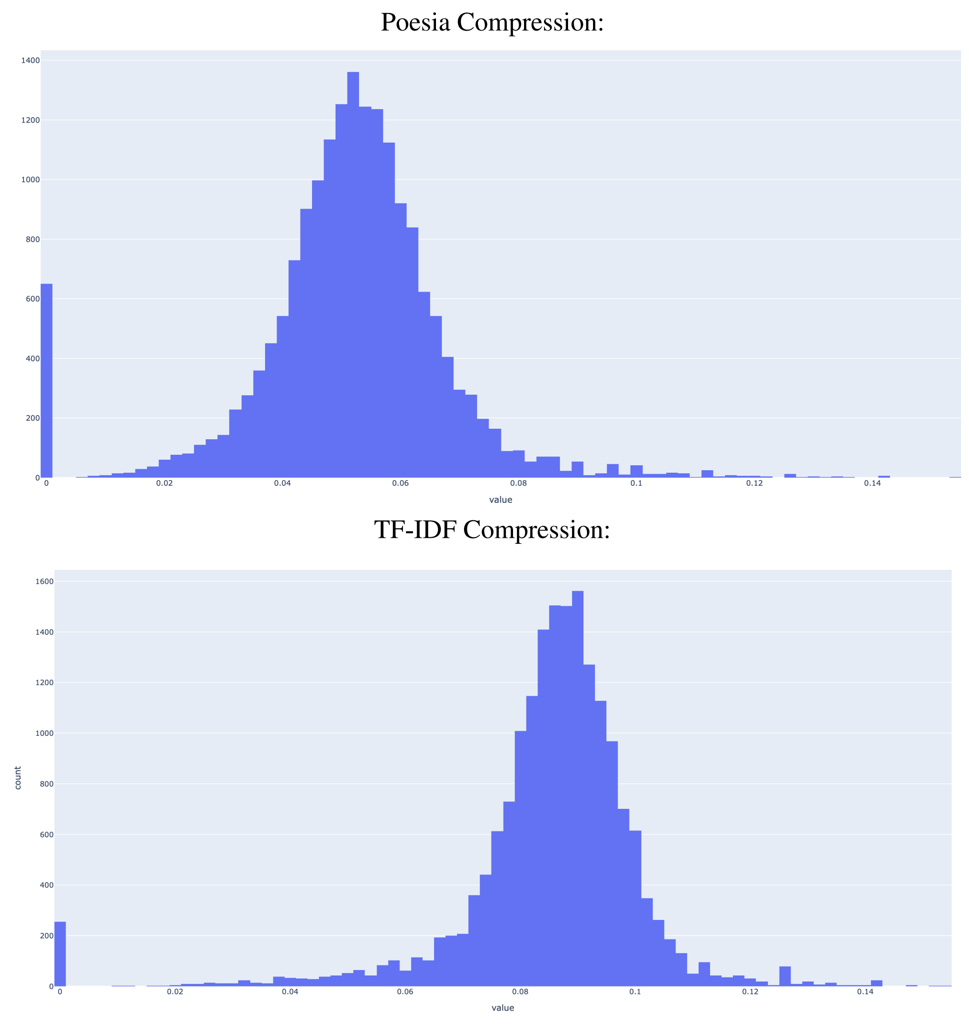

Figure 1: Histograms of Number of Compressed Tokens per Sample

the HuggingFace bert-base-cased BertTokenizer [4]) and then convert our 1-byte compression token ('@') to the traditional '[MASK]' token used by HuggingFace. We then convert the tokens to ids, pass them into our model, extract the predictions and apply softmax to retrieve the generated probabilities for various words in the decompression vocabulary. One thing to note is that because the output vocabulary of our model is smaller than the vocabulary as a whole, we need another layer of redirection from the model output. Thus, the outputs of our model are actually indices that then map into our list of words in the vocabulary that our compression scheme is willing to mask. Thus, we use the output of our model to replace masked tokens with tokens that correspond to the index in the compression vocabulary. Once we have replaced masked tokens with the tokens corresponding to words in the compression vocabulary, we then compute a cross entropy loss on the predictions for the masked tokens and their respective true tokens. In the final output, we insert the decompressed tokens back into the input document to generate the final decompression for the input.

5 Experiments

5.1 Data

We are using the WikiText2-v1 dataset via the HuggingFace API [5]. According to HuggingFace, "The WikiText language modeling dataset is a collection of over 100 million tokens extracted from

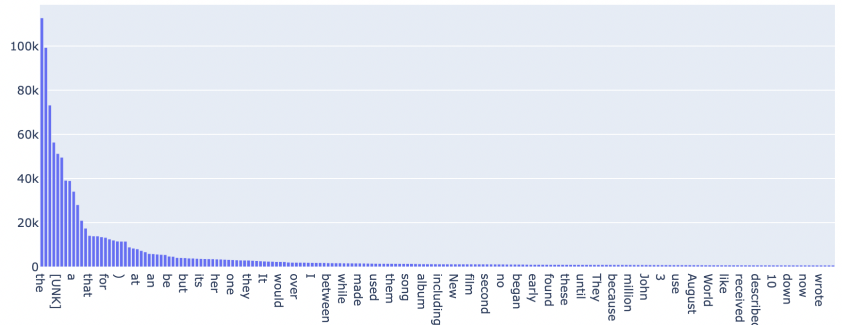

Figure 2: Histogram of top 200 most frequent words

the set of verified Good and Featured articles on Wikipedia." We chose this dataset since Wikipedia articles contain many long-term dependencies, providing strong examples of potential input to our compression algorithms and decompression models. The train split is comprised of 36718 examples, the val split has 3760 examples, and test set contains 4358 examples. We preprocessed the data to remove empty examples, examples that exclusively contain titles, and examples smaller than 10 words long. After preprocessing the dataset, we had 16184 train examples, 1726 validation, and 1938 test. These examples consist of paragraphs extracted from Wikipedia articles. Fig 2. depicts a histogram of the top 200 words in our resulting dataset.

Here are some insights that informed our experiment defintions:

- 2.6% of the words in our dataset are '[UNK]'.
- The 100 most common words in our training data occur as much as the rest of the words combined
- There are 43,023 unique words in our training data, and 14,956 unique words in our validation set.

Before runnning any of our global compression experiments, we generate the compression word lists as described in the methods section using the Wikitext dataset. To generate comprehensives inverse document frequency (IDF) scores for the TF-IDF compressor, we used the Tensorflow Wikipedia English [6] dataset as samples from Wikitext2-v1 are paragraphs from the document, not the document themselves. The Tensorflow Wikipedia dataset had some irregularities, thus we performed cleaning on the data before generating IDF scores.

5.2 Evaluation method

We will evaluate the model on our validation (during fine-tuning) and test splits (to get final statistics) using two primary metrics: accuracy and compression size. Our accuracy metric will be the total correct decompressed tokens divided by total decompressed tokens in the compressed text. We also calculate the total accuracy (number of correct tokens (masked and unmasked) over number of tokens in a document) as this provides insights as to how our overall system performs. In our final model, we will compare our model's compression size with that of gzip. This way, we can see how we are doing globally as well as in comparison with standard compression techniques. For this project, we are targeting a smaller compression metric (i.e. goal is size of compression $<<<$ size of gzip compression). We are eager to explore what the trade-off in accuracy and compression may look like.

5.3 Experimental details

- **Local Exeriments:** We are currently using a pre-trained bert-base-cased model for decompression and are not finetuning on our baseline. We are using 10 as our context window size. In our next steps, one improvement on our decompression that we are considering is finetuning the pre-trained model on the wiki-text2 dataset with randomized masking on the input. We will report fine-tuning hyper-parameters in our final report if we proceed with this strategy.

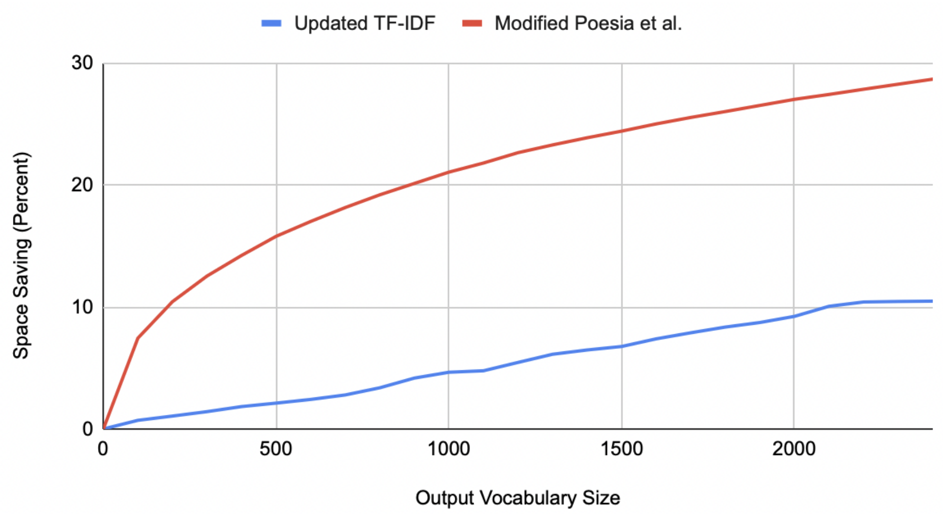

Figure 3: Space Saving vs Output Vocabulary Size

- **Global Experiments:** For all of our global compressor experiments we fine tune our decompressor for 2 epochs, a batch size of 32, with a learning rate of 1e-3, cross-entropy loss and Adam optimization. We saw examining our train/val curves when fine-tuning, our decompressor converges a little after the first epoch. We ran 6 experiments: 3 using the global Poesia compressor and 3 with the global TF-IDF compressor both with a compression list size of $N = 300, 1000, 2000$

5.4 Results

Table 2: Experiment Results

Method	Compression	Overall Accuracy	Compression Accuracy
Baseline	93.95 % (of original size)	92.44%	29.66%
Poesia, N=300	86.03 %	94.81%	62.74%
Poesia, N=1000	75.88%	88.03 %	45.84%
Poesia, N=2000	**68.74%**	82.40%	36.96%
TF-IDF, N=300	90.37%	**99.81%**	**81.56%**
TF-IDF, N=1000	85.24%	98.66%	72.30%
TF-IDF, N=2000	74.09%	96.17%	62.17%

We now examine how our experiments compared to Gzip and especially when our output compressions are applied to GZip. The second score on the below table is a result of applying our model to a text to compress it, and then applying Gzip to our compression.

Table 3: Compression Performance with and without GZip

Method	Compression	Compression + Gzip
Gzip	**58%**	**NA**
Poesia, N=300	87.50 %	51.29%
Poesia, N=1000	78.93%	46.07%
Poesia, N=2000	**72.97%**	**42.11%**
TF-IDF, N=300	98.57%	57.17%
TF-IDF, N=1000	95.35%	55.82%
TF-IDF, N=2000	90.76%	53.52 %

6 Analysis

Based on the results from Table 2, we can see that, in general, a larger output vocab size leads to a lower accuracy in predicting the masked tokens. This makes sense as having to make predictions over a larger collection of words will inevitably leave room for more ambiguity between situations in which words in the vocab can be used.

Another point to note is that the Poesia compression is consistently able to perform better than TF-IDF when it comes to compression size. This makes sense when looking at Figure 3. We see that across all vocabulary sizes, Poesia compression consistently achieves significantly higher space savings (i.e. compresses documents more) than TF-IDF. From the characterization shown in Table 1, we see that Poesia compression more aggressively compresses, compressing more tokens per sample than TF-IDF compression.

While Poesia compression may perform better in terms of compression, TF-IDF clearly performs much better when it comes to accuracy, likely due to fact that fewer compression tokens leads to less decompression ambiguity as mentioned above. This can be seen in the following original example excerpt from our dataset:

"'Playing a doctor is a strange experience. [UNK] you know what you're talking about when you don't is very bizarre but there are advisers on set who are fantastic at taking you through procedures and giving you the confidence to stand there and look like you know what you're doing.'"

Poesia compression with an output vocabulary size of 2000 leads to a masking and prediction of

"'Playing a doctor is a strange @. [UNK] you @ @ you're talking @ @ you don't is @ bizarre but @ are advisers on set who are fantastic at @ you @ procedures and @ you the confidence to stand @ and look @ you @ @ you're @.'"

"'Playing a doctor is a strange idea. [UNK] you know what you're talking about what you don't is really bizarre but there are advisers on set who are fantastic at making you through procedures and give you the confidence to stand back and look after you know when you're doing.'"

While TF-IDF compression with an output vocabulary size of 2000 leads to a masking and prediction of

"'Playing a doctor is a strange experience. [UNK] you know what you're talking about when you don't is very bizarre but there are advisers on set who are fantastic at taking you through procedures and giving you the confidence to @ there and look like you know what you're doing.'"

"'Playing a doctor is a strange experience. [UNK] you know what you're talking about when you don't is very bizarre but there are advisers on set who are fantastic at taking you through procedures and giving you the confidence to stand there and look like you know what you're doing.'"

The inherent aggressiveness of Poesia compression can be seen in sections of text like "look @ you @ @ you're @." Meanwhile, TF-IDF did not mask any portion of that text. With Poesia masking out large continuous portions of text, there is no wonder that it's accuracy suffers. With enough time and context, a human could potentially figure out the correct translation of that masked text because we understand that the phrase "look like you know what you're doing" is common, but a model, even one pretrained with BERT, will likely struggle.

From our 6 model configurations we see that Poesia Compression with N=2,000 (the largest compression token vocab size) was most performant for our compression objective, reducing files to 68.74% of their original size. However, when it comes to our accuracy objective TF-IDF with N=300 and 1000 achieve the best overall decompression accuracy, with 99.95% and 99.97% accuracy each. However, the TF-IDF compressor with N=300 decompresses each mask token with 10% higher accuracy than TF-IDF N=1,000. This again, is likely due to reduced ambuiguity by having a smaller set of words to predict across. What's notable, that despite this large difference in performance on a token level, the difference in overall accuracy is minimal. Notice that TF-IDF with N=1,000 is over 10% more effective at compressing a document. While decompression accuracy and compression rate are at odds with one another, the N=1,000 model for the TF-IDF compressor strikes a balance between the two objectives although N=300 is a strongest candidate for creating a system that is closest to being lossless..

Despite this balance, we see that the more aggressive Poesia compressor achieves an additional 16% on with only a 8% dip in accuracy. While the dip in accuracy is significant, the compression benefits don't completely discount this compressor.

As our compressor is intended as a practical tool, we also compare it against the popular Gzip method. Gzip is lossless and therefore achieves perfect compression and overall accuracy and can compress a document to 58% of its original size on average. While our most performant configuration, Poesia N=2,000, is significantly less performant when compressing than Gzip (a difference of 14%), we notice something interesting in Table 3 when examining the compression performance of our system when it works in tandem with Gzip. When we compressed input with our model and then compressed that output with Gzip, we found that we were able to beat Gzip alone across all 6 experiments. Notably Poesia N=2,000 achieved a compression rate that is 16% better than that of Gzip.

7 Conclusion

Through our experiments we explored and characterized the trade off between decompression accuracy and compression size that compression systems face, and framed the compression problem as a world level NLP masked language problem. We found that a more aggressive compressor (Poesia) does the best job reducing file size, however, at the expense of decompression accuracy. We also found that larger compression vocabularies increase compression also at the expense of decompression accuracy. We also found that while none of our compressors outperform Gzip along, they all can outperform standard compressors (Gzip) when applied in tandem with those compressors. When evaluating holistic on both objectives, we found that TF-IDF with N=300 does the best job of balancing these trade-offs as it is able to reduce file size when applied with Gzip to 42.11% of the original file size while maintaining 99.81% overall compression accuracy and 81.56% compression accuracy.

For future work, we would like to further explore hyper-parameters to further increase our compression accuracy without having to further constrain our compression. Another way we would like to improve the accuracy is by further evolving our compressors such that words in our compression list have low similarity score as our BERT model would then have a much easier time reconstructing the original text, since all the options are used in very different contexts and thus are to draw a decision boundary across.

References

[1] Gabriel Poesia and Noah D. Goodman. Pragmatic Code Autocomplete. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

[2] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa. Deepzip: Lossless data compression using recurrent neural networks. In *2019 Data Compression Conference (DCC)*, pages 575–575, 2019.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[4] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[5] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *CoRR*, abs/1609.07843, 2016.

[6] Wikimedia Foundation. Wikimedia downloads.