Compression Through Language Modeling

Antoine El Daher
aeldaher@stanford.edu
James Connor
jconnor@stanford.edu

1 Abstract

This paper describes an original method of doing text-compression, namely by basing the compression algorithm on language models, and using probability estimates given by training files to build the codewords for the test files. Often, when a compression program sees words like ”he is”, it never automatically determines that a word like ”going” is much more likely to follow a word like ”be”, and the reason for this is that the compression algorithm sees bits, and has no prior knowledge regarding which words follow which other words. This is something that we attempt to address in this paper. We will first describe how to build a very efficient compression algorithm, based on a language model. Then, we will address the issue of categorizing the input document in order to be able to use a more specific dictionary; we will also deal with documents that have multiple topics, using a sliding context window, that adapts itself based on the observed words, and makes a better blend of the prior and observed probabilities. We show compression ratios (defined as output file size over input file size) of 0.36, and that for text, our algorithm is very competitive with very well-established algorithms, such as gzip, bzip2, WinZIP.

2 Introduction

File compression is widely applicable as a way to store data using less space. We present a method of file compression specialized for compressing text. All file compression works by exploiting low entropy in the input sequence in order

to represent it as a shorter, higher entropy sequence. Shannon demonstrated that lossless compression of an i.i.d. sequence $\{X_i\}$ where $X \sim P$ is fundamentally limited by:

$$\begin{aligned} \frac{\text{\# compressed bits}}{\text{\# signal bits}} &\geq H(X) \\ &= -\sum_{x \in dom(X)} p(x) log(p(x)) \end{aligned}$$

Generally applicable compression methods (e.g. Huffman, Lempel-Ziv, arithmetic) try to estimate the distribution P using the file to be compressed. However, often this estimate is far from the true data distribution because there isn't enough data to accurately estimate P. We try to use prior knowledge to exploit structure in the distribution P to get a more accurate, lower entropy estimate. For example, if we were to compress text by estimating a distribution over 1's and 0's, we would probably find that the proportion of 1's and 0's is about even and so

$$\begin{aligned} H(X) &\approx -.5log(.5) - .5log(.5) \\ &= log(2) \\ &= 1 \end{aligned}$$

and so we would achieve almost no compression.

By using higher block lengths (e.g. $n > 1$ bits instead of bit by bit), we could achieve better compression using Huffman coding. Also, Lempel-Ziv automatically estimates appropriate block lengths from data. Instead, knowing that the input data is text, we use a language model trained on a large corpus to estimate the distribution P. We achieve better results because text is relatively low entropy and using a language model allows us to better exploit this. In the Theory section, we prove that a low-perplexity language model is equivalent to a low-entropy estimate of P. In the Work section, we describe our base language model. In the Extensions section, we describe specializing our language model to text within specific contexts, both statically and dynamically. We also present a method for compression of Java code using parsing that we think is a good model for better compression of text. We obtain compression ratios that are close to 3-to-1.

3 Theory

3.1 Huffman Coding

A Shannon code for a sequence $\{X_i\}$ assigns codes where $length(code(x)) = \lceil log(\frac{1}{p(x))} \rceil$. In this way,

$$\begin{aligned} E[length(code(x))] &= \sum_{x \in dom(X)} p(x) \lceil log(\frac{1}{p(x)}) \rceil \\ &\leq \sum_{x \in dom(X)} p(x) \left(log(\frac{1}{p(x)}) + 1 \right) \\ &= H(X) + 1 \end{aligned}$$

A Huffman code is similar to a Shannon code but always achieves a code with expected code-lengths less than or equal to those resulting from the Shannon code (equality when $\forall x \lceil log(\frac{1}{p(x))} \rceil = log(\frac{1}{p(x))}$). Intuitively, Huffman coding makes up for the waste in the Shannon code from taking the ceiling. Huffman coding is based on a simple greedy algorithm:

1. Gather counts of all symbols $x \in dom(X)$.
2. Find the two symbols, x_{l1} and x_{l2}, with fewest counts.
3. Merge these two symbols into a new, composite symbols x', linked to x_{l1} with a 1 and to x_{l2} with a 0.
4. If more than one symbol remains, recurse to (2). Otherwise, we have a Huffman code where each symbol x is encoded by the sequence of bits in the links that lead through composite symbols to that x.

(Cover and Thomas) [1]

3.2 Perplexity and Entropy

Perplexity is defined as:

$$\begin{aligned} \left(\prod_{i=1}^{n} p(x_i) \right)^{-1/n} &= \left(2^{log(\prod_{i=1}^{n} p(x_i))} \right)^{-1/n} \\ &= 2^{(-1/n) \sum_{i=1}^{n} log(p(x_i))} \end{aligned}$$

And note that by the law of large numbers,

$$\begin{aligned}(1/n)\sum_{i=1}^{n} -log(p(x_i)) &\approx E[-p(x_i)] \\ &= H(X)\end{aligned}$$

Therefore the perplexity is asymptotically equal to $2^{H(X)}$. Therefore finding a language model that gives good perplexity and finding a low-entropy distribution that allows for good compression are equivalent tasks.

4 Language Model

4.1 Character Compression

We can represent text as a sequence of ASCII characters, estimate a distribution over these characters, and then compress the text using huffman coding on this distribution. We can also capture higher order dependencies by conditioning the distribution on previous characters by counting character n-grams.

We implemented a fixed huffman code for character n-grams based on n-gram counts from a large corpus of text. Then we can use this fixed code to compress text without having to store a new huffman table for every document. If the character distribution in the compressed document matches the prior character distribution, this coding is optimal. Otherwise, it would be better to create a unique huffman code. We used the character coding as a backoff for word coding, as described below, and used a fixed huffman code.

Though very high order character n-grams (Cover and King [3]) would be able to capture dependencies between words, this approach would suffer from data sparsity. Since there are very strong dependencies between words, capturing these dependencies would result in a lower entropy distribution and better compression. So the approach we present in the next section, which uses word n-grams, is more desirable. Still, when we come across an unknown word, we need to be able to compress it even if it doesn't appear in our fixed training dictionary. So we backoff to our character langauge model to encode words unseen when estimating the word n-gram distribution.

4.2 Word Compression

This is where the ideas from Natural Language Processing come in to extend standard compression, bringing the compression ratio down.

4.2.1 Description

A normal compressor will always look at the input as a sequence of words, or a sequence of bits, and will attempt to use some tricks, based on no prior probability distribution of this input. For a general file compressor, this is a reasonable thing to assume, but what if one was asked to compress text? Then we'd know that for example, certain word trigrams are far more likely than others, even if we hadn't seen any occurence of them previously in the text. Let's explain this point some more: for humans, the trigram "will have to" is far more likely than the trigram "will have eat"; even though this is essentially in the English grammar, we rely mostly on our knowledge of what sentences can or can't be made, without having to "read through" the text and learn the grammar as we go.

This is what we encapsulate in Word Compression; we train a language model, and then estimate the probability of each trigram in the input list; the Huffman trees are now based on the probability distribution of our training set instead of the probability distribution in the test set, so that we do not have to include any Huffman table or similar overhead.

4.2.2 Algorithm description

The compression algorithm works in 5 steps:
1- Train a language model for unigrams, bigrams and trigrams
2- For each bigram, find the probability of the third word being any observed word, or UNK, using linear interpolation as well as absolute discounting
3- For each bigram, build a Huffman tree based on the distribution of the 3rd word
4- Also build a unigram Huffman tree for the words, based on their probabilities
5- Go over the words, replacing them with the corresponding bits in the Huffman tree; if the words are unknown, code the UNK token, then revert to a lower order model; if even the unigram model considers this word unknown, then revert to character compression, described in the previous section.
Throughout this section, we will carefully explain each of the steps mentionned above, as well as all the optimizations that we implemented to make them run in reasonable time, even though they work on a considerable amount of data

4.2.3 Training the language model

We decided to code our program in C++, since we are more efficient with that language. The first part of training the language model is simply to maintain counts for each unigram, bigram and trigram that we encounter. This can be done very efficiently using a map data structure, quite similarly to what was done in the first assignment. A difference here though, is that instead of having a map containing all the possible trigrams, it is now preferable that for each bigram, we have a map containing all of the words that could follow this bigram, along with their respective probabilities. The reason for this decision will be made obvious later on, in the section about building the huffman tree. We essentially trained our language model on the Penn Treebank.

4.2.4 Word probabilities

Unlike standard Huffman coding, in this case the probability distribution of a given token (in this case, a single word), is dependent on the two (or one) previous words. So for every given bigram, (or unigram) we need some kind of distribution over how likely each word is to follow it. For example, the words ”to have”, might have a probability of being followed by ”been” of 0.1, by ”seen” of 0.2, by ”done” of 0.3, and by UNK of 0.4. So to get this, all we have to do is smoothing on the distributions; to accomplish this, we decided to use linear interpolation along with absolute discounting, which resulted in a perplexity of 340 for the TreeBank test set. So at this stage, we have for every bigram, a smoothed probability distribution, that includes the unknown tokens.

4.2.5 Building the Huffman Tree

For every bigram now, since we have a probability distribution, it is very easy to build a Huffman tree, which is proven to be optimal given a correct probability distribution (and achieves entropy). This can be done by pairing the two elements with smallest probability into a single node, then reiterating the process; the algorithm is well known and is described in many papers. For example, for ”to have”, if our smooth distribution says: ”been” - 0.1, ”seen” - 0.1, ”done” - 0.2, ”UNK” - 0.3, ”told” - 0.3, then ”been” and ”seen” will be combined into a single node (call it A), then A and ”done” will be combined into a single node B (with probability 0.4); after that ”UNK” and ”told” will be combined into a node C. The result of this is that the code for UNK will be 00, the code for ”told” 01, the code for ”been” 101, ”seen” 100, and ”done” 11. This makes the expected length 0.1 * 3 +

0.1 * 3 + 0.2*2 + 0.6*2 = 2.2 bits, which is provably optimal for the code. Note that this is done by looking at the probability distribution of the words that follow every possible bigram.

4.2.6 Building the Unigram Huffman Tree

The same process is applied for building the Huffman tree for the unigrams, generating a unigram probability distribution on the words, smoothing it, then building the corresponding huffman trees.

4.2.7 Performing the Compression

Once the huffman trees are built, performing the compression becomes a relatively easy task. We go through the words, looking at the two previous words as we go (initially two start tokens); we look up the word in the huffman tree that is conditioned on those previous two words. If the word is found, we simply insert its Huffman code. If the word is not found, then we write the code for the UNK token, then revert to the unigram model. This is a fairly straightforward and easy way of compressing, and we simply replace each word by its code. If the word also has never been seen before, then we introduce another UNK token, and then revert to character-based compression. As will be shown in the results section, this method yields very impressive results, and is lossless.

There is also an inherent relationship between the perplexity of the file that we are compressing and the actual compression ratio that it achieves, which we will also make explicit in the results section.

4.2.8 Performing the Decompression

Understanding the compression scheme leads to an easy understanding of the decompression scheme. Initially, we start off with the same training data as we had during the compression, so that we are aware of all the probabilities, and hence know which word/character every Huffman code represents. We then start going through the word, looking up the corresponding Huffman tree, and seeing which word that code corresponds to, and output it, updating the previous ”context” accordingly. This can be done quite quickly, by performing a look-up for each of the bigrams. In case the word read is ”UNK”, we know that we automatically need to revert to a unigram model, and do so accordingly. The underlying idea here is

that the Huffman trees are both built based on the training data, and hence both the compression and the decompression scheme have them available for use.

4.2.9 Optimizations

Several optimizations were used to make the algorithm faster. We used an STL map to map words to codewords immediately, so that each word could be encoded in at most logarithmic time. When constructing the tree, the possible bigrams and trigrams were always stored in balanced search trees, to make querying as fast as possible, particularly because we were using a very large amount of data, that was too bulky to move around.

5 Extensions

This section describes some of the extensions that we implemented to be able to obtain a better compression rate, based on more natural language processing techniques. Even though the algorithm above was very good for general text, it could be made better by specializing it to certain kinds of texts, as will be described in the following sections.

5.1 Specific Contexts

Certain texts or articles often mention specific topics more than others; sometimes they even only talk about a single topic. This is essentially what we tried to encapsulate; given a text, we first categorize it into either sports, politics or business. Once that's done, we instead use a specific sports dictionary, politics dictionary or business dictionary; since such dictionaries are usually smaller than a global dictionary, and are very likely to contain the more specific words that don't appear in the larger dictionary, we were confident that this would give a boost in compression.

We therefore decided to code an article classifier, which we trained on articles from CNN.com, subdivided into 3 groups: sports, politics and business, for illustration purposes. We figured it would be a good idea to have a maximum entropy model categorize the latter, but ended up using something simpler, but similar. The idea behind the algorithm was that we would read through each of the training data, and then find the normalized probability of occurence for each of the words; then, when going through the test set, we would simply find the

log probability of an article being in a specific category, by summing up the log probabilities of each of the words being in a particular category. As will be shown in the results section, this worked quite well, and we were able to categorize the words quite efficiently.
Note that for the probability of a word being in an category simply used the ratio of the probability of that word being in the training set category, over the sum of the probabilities of the words over all the categories. We also experimented by taking the exponentials of the probability over the sum of the exponentials of the probabilities, to make it closer to the maximum entropy model, but this didn't result in any improvement, at least on our test set.

5.2 Sliding Context Window

The idea behind this algorithm is that some texts tend to be centered about more specific topics, and that those topics might change throughout the text, but would do so quite smoothly. A context window, is a sliding window of say, 1000 words, where the probability distribution for the Huffman trees described in the "Word Compression" section, is instead biased towards the words that have been seen in the window.

5.2.1 Bayesian Distribution Estimation

Bayesian methods use a prior distribution on the parameters of the data probability distribution. Instead of using the maximum-likelihood estimates for parameters, which can often overfit, a Bayesian prior allows for a smoother fit that is much less prone to overfitting. Bayesian methods are especially useful when trying to estimate a distribution using a small amount of data and where we have a good estimate of the prior distribution.

Adjusting the word distribution to specific contexts is a case well-suited for Bayesian methods. Our sliding context window uses a strong Dirichlet prior (Koller and Friedman [2]) over parameters θ:

$$\theta^n \sim Z(\alpha_1, \ldots, \alpha_n)\theta_1^{\alpha_1} \cdot \ldots \cdot \theta_n^{\alpha_n}$$

where α_i are the prior pseudo-counts estimated from a large corpus and Z is a normalizing constant. The posterior distribution over parameters, given the data in the sliding context window, is also Dirichlet, since the Dirichlet prior is a conjugate prior:

$$\theta^n \sim Z(\alpha_1 + \beta_1, \ldots, \alpha_n + \beta_n)\theta_1^{\alpha_1+\beta_1} \cdot \ldots \cdot \theta_n^{\alpha_n+\beta_n}$$

where β_i are weighted data counts within the sliding context window. As the context window slides over text, we re-estimate the posterior distribution.

5.2.2 Algorithm

This algorithm is in a sense, adaptive to the input that it is given; here's how it works: we maintain a window of a specific size, which for now, we'll assume to be 1000. We then slide the window through the text, and perform the compression as we used to do it before; however, every time a new word enters the window, we increase the counts of the corresponding unigrams, bigrams and trigrams, and every time a word "falls out" of the window, we decrease them. Whenever we increase a count, we re-compute the Huffman tree for the specific bigrams entirely Let us look at a simple example:
If we are given that for the bigram "to have", the possible followers are: "been" - with count 1, "seen" - with count 2, and then we get in the input, "He needs to have been eating, but not to have drunk", then as we go through the window, the "been" count will be updated, and then recomputed. The word "drunk" will be introduced with count 1, so that while we're in its context window, it won't be refered to as an UNK token.
So the adaptivity of this algorithm comes from the fact that words/trigrams that have appeared nearby always tend to be more likely to re-appear in a 1000-word window; also trigrams that are unknown don't need to be coded as UNK, then backed-off to be coded again. This basically uses some kind of local memory which is reminescent of the Lempel-Ziv algorithm.

5.2.3 Performance

Because of the high computational complexity of rebuilding the Huffman tree after every single word, the algorithm that we obtained was pretty slow, processing around 30 sentences per second, as opposed to around 1000 for no adaptivity. As such, we were able to perform some simple tests, but not on large enough datasets for us to make a good comparison.

5.3 Parsing

6 Results

The results that we obtained were pretty good, and highly competitive with the other existing algorithms for compression, even though they didn't use any extremely advanced tricks or techniques for minimizing entropy; all they used was simple Huffman coding, which is a small step in the general compression algorithms.

We tested the program on four distinctive text files: the Penn Treebank test file, the Penn Treebank validation file, the BLLIP test file, and the BLLIP validation file, training the compression algorithm, respectively, on the Penn Tree training file, and on the BLLIP tree training file. Figure 1 shows the results that we obtained. From these results, one can see that our performance comes very close to

File Name	*Original*	*Bzip2 Compression*	*Gzip Compression*	*Text-based Compression*
Treebank-sent.test.txt	558324	145853	198522	202186
Treebank-sent. validate.txt	554908	147401	199373	201645
Blip-sent.test.txt	5931314	1514353	2105318	2151026
Blip-sent.validate.txt	5957667	1519737	2112540	2134206

Figure 1: Results on standard databases

that of gzip, even though, as we mentionned before, no particular optimization is made.

We then decided to watch how the compression rate evolves as we add more words to the training data. To no suprise, as shown in figure 2, having a larger training data makes for better compression. The reason for this is clearly that we are now more likely to observer trigrams that we have encountered before. Words that take more bits to be compressed are simply words that are unknown. The reason that this happens is that we first have to encode the UNK token for trigrams, then encode the UNK token for unigrams, then finally revert to character compression, before actually getting the character correctly. As we conjectured previously, there seems to be a strong correlation between the perplexity of the training set, and the compression that we are able to achieve. To investigate this further, we

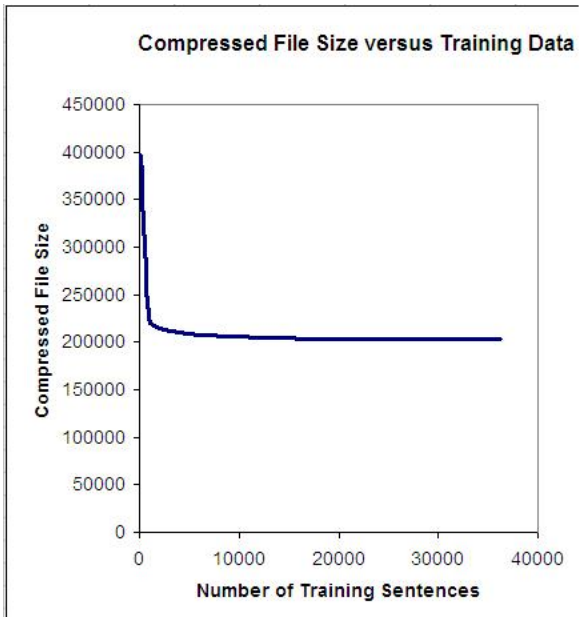

Figure 2: Learning Curve for Compression

drew a plot of the measured perplexity of the test set, versus the compression that we obtained. Unsurprisingly, this formed an increasing curve, as shown in figure 3. We also had diverse results that were described throughout the text. For example, categorizing the articles happened to have a very high accuracy, which we don't report since we only ran it on a few samples (all of whom it got correctly).

7 Conclusion and Further Work

Throughout this paper, we've described several complementary approaches at dealing with natural language processing based compression. We began by training a language model to be able to estimate the probability of words appearing after a certain bigram, and then building a Huffman tree off that. We then implemented a character based compression for unknown words. Once that was done, we experimented with more original ideas. The first of these ideas was to have a preprocessing step that would categorize the file that needed to be compressed, and then pick the corresponding dictionary; we were able to obtain a very high accuracy when categorizing the words, using a fairly simple technique. The second idea was to introduce a sliding window, which in addition to the usual things

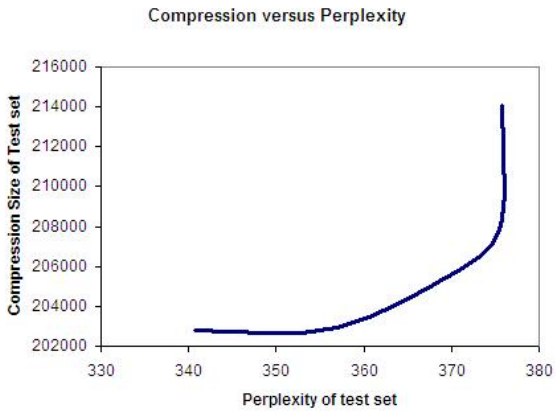

Figure 3: Perplexity and Compression Correlation Graph

that the language model does, would adapt the word probabilities to the observed ones.

We believe that such an idea has a strong chance of improving already existing compression schemes, which, even though are adaptive, rarely actually use information about the word trigrams. It is difficult to compete with an extremely well-established compression software like gzip, but our results were pretty close to those of gzip, without implementing any of the additional ideas that gzip uses. As far as compression algorithms go, having a subject based compression can be a very good idea for some cases. For example, in a News server, sports articles could be compressed using a sports dictionary, politics using a politics dictionary, and so on; this would result in a very strong compression ratio. Finally, our algorithm has no overhead, and doesn't need to store information like a Huffman table; as such, it could be used immediately on a communication channel without including any other information. For further work, we would like to explore dealing with many spaces, and other special characters; as it is now, we assume that all words or tokens are separated by a single space, and we deal with commas, columns, etc, as being simply words like any other; other word that we would have liked to do was to integrate some of the advanced features of gzip into our code, and see what performance would result, but the point of the paper was mostly to show that using language models for compression works, and that's why we tried to do.

References

[1] Cover, Thomas. *Elements of Information Theory,* John Wiley & Sons, 1991.

[2] Koller, Friedman. *Structured Probabilistic Models,* Unpublished (text for CS 228 - Probabilistic Models in A.I.), 2005.

[3] Cover, King. *A Convergent Gambling Estimate of the Entropy of English,* IEEE Transactions on Information Theory, Vol. IT-24, No. 4, July 1978.

8 Appendix: using the code

The compression software that we made is fairly easy to use. The first command that should be run (after starting testbed) is ”lmtrain”, which will read the treebank-train.sent.txt file, and train the model, as well as generate all the required Huffman trees as efficiently as possible. Once that's over with, you can compress any file by typing: ”lmcompress file1 file2”; we recommend ”lmcompress treebank-test.sent.txt output.dat”, and then you can decompress with ”lmdecompress output.dat treebank-test.decompressed”. There are other commands, such as ”testall MAXSENT” (e.g. ”testall 1000”), which will train the model with MAXSENT sentences, perform compression, and report perplexity and the compressed file size. The code is worth taking a look at, particularly because of the numerous optimizations that we chose not to mention in the report.