INDIVIDUAL PROJECT

UNIVERSITY OF SUSSEX

SCHOOL OF ENGINEERING AND INFORMATICS

Compression of Natural Language

Author:
Guy Aziz
267649

Supervisor:
Prof. Ian Mackie

April 29, 2024

Contents

1. Introduction

Human language contains many redundancies and regularities. It is possible, for example, for a person to infer from an incomplete sentence a missing letter or word, or to spot an error through an inconsistency between text and context.

This is done, first, through the abstraction of an underlying, concise representation of the text, and then through a re-generation of the elaborated text from the conceptual understanding. In humans, one's ability to induce a general pattern and deduce its application to a specific case in this way is often an indication of their comprehension of a text, and of having a grasp of the underlying meaning.

Because of this, it is believed by some researchers (most notably Marcus Hutter) that the ideal lossless compression of a text would require comprehension, and that the first is therefore an AI-complete problem.

This project aims to explore the relationship between compression and comprehension.

To view the code used in this project, visit `https://github.com/Guy29/FYP`.

1.1 Motivation

Occam's razor is a general principle in science and rationality commonly attributed to William of Ockham (1290-1349) which states "entities should not be multiplied beyond necessity", usually interpreted to mean "the simplest theory is usually the correct one."

It is easy to take this principle for granted, as it seems to work in both science and in our daily experience. But why this should be the case, i.e. why we live in a regular enough world that simpler models of it tend to be more correct than more complex ones, is not obvious, and it has been historically pointed out by many philosophers (most famously David Hume) that knowledge gained in this way stands on shaky foundations. (Henderson, 2018)

Solomonff's theory of inductive inference provides a formalism for Occam's razor. It posits an agent making observations in a world that operates by an unknown algorithm, and based on that premise shows that the agent would do well to assume that the length of the algorithm by which its world runs in effect follows a probability distribution that assigns shorter (and therefore simpler) algorithms more probability. This probability distribution is known as the universal prior.

The argument Solomonoff uses can be understood as follows: if the agent considers all algorithms of the same complexity as equally likely, then the algorithms can be divided into subsets where each subset is functionally equivalent. Because there are more ways to implement simpler algorithms than more complex ones, they accrue more probability mass. For example, a crime investigator who knows that Alice likes apple pie and Bob doesn't may consider the following hypotheses (of equal complexity) for the disappearance of an apple pie:

Need a better example here

1. Alice stole it, wearing a red shirt.
2. Alice stole it, wearing a blue shirt.
3. Bob stole it, after having a change of taste.

If each of these hypotheses is given the same probability initially, based on being of equal complexity, then a grouping of the first two as functionally equivalent (the colour of the shirt being irrelevant) makes it twice as likely that Alice is the culprit.

Solomonoff's theory of inductive inference uses the concept of Kolomogorov complexity, which refers to the length of the shortest program that would produce a specific output. For example, the Kolomogorov complexity of the string "1111 . . . 11111" is lower than that of "wp9j8 . . . fd27c", as the first is more regular. The actual design of compression algorithms can be thought of as a way of empirically determining the Kolomogrov complexity of data by finding an algorithm that produces it which is shorter than the data itself.

In the same way that Occam's razor lacked formalism and proof until Solomonoff, the concept of intelligence similarly lacks formalism in modern computer science, and psychologist R. J. Sternberg remarks "there seem to be almost as many definitions of intelligence as there were experts asked to define it." (Legg, Hutter, et al., 2007)

Hutter (2000) proposes a formalism of an intelligent agent which he terms AIXI that combines the above ideas as well as ideas from reinforcement learning. AIXI is a theoretical agent which, in each time step,

1. Makes an action a_i.
2. Receives an observation o_i and a reward (positive or negative) r_i.
3. Generates all possible algorithms by which its world can run which would have predicted all of its observations and rewards so far, and weighs the probabilities of those algorithms inversely to their length (i.e. it applies the universal prior).
4. Uses the most likely algorithms (or models of its world) to simulate the world, predict potential observations and rewards for potential future actions, and thereby decide on its following actions to maximize its reward.

It can be seen that the above description of AIXI requires an agent that can effectively create a concise, compressed world-model that generates the observations it has made of its world so far (i.e. arriving at the simplest explanation for the underlying mechanisms of the world it inhabits, where simplicity indicates low Kolmogorov complexity), and that having such a model is most predictive of its success. It is for this reason that Hutter believes that intelligence can be defined in terms of data compression.

Based on this hypothesis, Hutter (2006) created the Hutter Prize, intended to "encourage development of intelligent compressors/programs as a path to AGI". The prize rewards improvements in data compression on a specific 1 GB text file, titled `enwik9`, which is extracted from the English Wikipedia, chosen on the reasoning that "Wikipedia is an extensive snapshot of human knowledge. If you can compress the first 1GB of Wikipedia better than your predecessors, your (de)compressor likely has to be smart(er) [...] while intelligence is a slippery concept, file sizes are hard numbers."

This project examines some of the intuitions that relate compression and comprehension of natural language, and provides an overview of some of the existing theory and of its practical applications.

1.2 Professional and Ethical Considerations

To the best of my knowledge, there are no professional or ethical considerations, as given by the BCS code of conduct (`https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/`) that would constrain any part of this project. The core of the project is a review and discussion of existing techniques, and practical applications will likely be limited to improvements on compression algorithms. All data used in this project is in the public domain.

1.3 Structure of this report

This report approaches compression of natural language from a few different angles. These approaches share some essential concepts, but each also requires its own bit of background. For readability, I've opted to structure the report in a similar way, giving a high-level overview of essential abstract concepts in the Overview chapter but also beginning each section of the Investigation chapter with its own specialized background.

2. Overview

In this section, I provide an overview of some of the essential concepts used in this project, illustrated through the following thought experiment:

Carol, Dave, and Trent are playing a game. In each iteration of the game, Trent will present Carol with some raw data in some format (for example, results of coin flips, or randomly chosen letters) and Carol will try to communicate the data to Dave (who is in the next room) as briefly as possible. Carol and Dave are communicating through a channel that allows only 0s and 1s to be sent. Carol and Dave each know the format of the data and are allowed to coordinate on a strategy for communication before the game starts. In fact, Carol and Dave are so good at this coordination that we may assume that everything Carol knows before the game starts, Dave also knows.

For an example, let us say that the data Trent is presenting to Carol is the result of a sequence of fair coin flips in the format "heads, tails, tails, ..." Because they each know the format, a simple strategy would be to use a **code** where heads are denoted by 0s and tails are denoted by 1s over their communication channel. We refer to the result of each coin flip as a **symbol**, and in this example we see that the **code rate** is 1 bit per symbol.

In this example, symbols are **independent and identically-distributed random variables (i.i.d.)** because each coin flip has the same 50-50 chance of coming up as heads or tails (identical distribution) and because the result of one coin flip doesn't affect any other (independence).

2.1 Information and entropy

Before attempting to compress a work of literature, let us first think about individual letters.

In another instance of the previous game, Trent presents Carol with a sequence of letters, each randomly chosen from a large English text. As before, symbols are independent and identically-distributed, but in this case, the distribution is not uniform, since the letter E occurs much more often than the letter Q.

As well as saying that Q has a low probability, it is sometimes said that Q has high **surprisal**. Because of this, Carol and Dave would be wise to privilege the letter E with a short representation in their code (say `00`), because it will occur often, and vice versa. Because this will leave Q with a longer code, we also say that Q has high **(self-)information**.

In his seminal paper "A Mathematical Theory of Communication", Shannon (1948) defines the **information content** of a symbol as length of its binary representation when the symbol code is chosen optimally (estimated as $-\log p(x)$, where $p(x)$ is the probability of that symbol), and defines the **entropy** of a random variable (such as our symbols) as the average amount of information per symbol, given by

$$H(X) = \mathbb{E}[-\log p(X)] = -\sum_{x\in\mathcal{X}} p(x)\log p(x)$$

for the discrete random variable X. In simple terms, the entropy of a variable is the average number of yes/no questions one needs to ask to determine the value of that variable when choosing one's questions optimally. For a fair coin, one needs to ask only one yes/no question to learn the outcome, and so the entropy of that outcome is 1 bit (sometimes called 1 shannon).

For the English language, the entropy of individual letters based on their frequency is known to be approximately 4.14 bits. This value indicates that, if one person were to select a letter at random from an English text and another person were to try to guess it by asking only yes/no questions optimally, they would need to ask 4.14 questions on average.

The importance of this value is related to Shannon's **source coding theorem**, given in the same paper, which shows that no code can be chosen for which the code rate is less the entropy of the symbols in question.

In our example with randomly-chosen English letters, the calculated entropy of English letters indicates that no code that Carol and Dave can choose can have a lower code rate than 4.14 bits per symbol. But how might they go about constructing a minimal code?

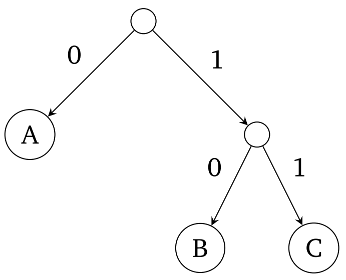
2.2 Entropy coding

An **entropy coding** is a consistent way of assigning binary sequences to symbols. Entropy codes usually aim at being minimal in the sense that an encoding of a stream of symbols that the code is designed for should have an average code rate that approximates the symbols' entropy.

The codes are generated from an input mapping between the potential values of each symbol and that value's probability. For example, a coding algorithm may take as its input the mapping $\{A \to 0.8, B \to 0.15, C \to 0.05\}$ and output the mapping $\{A \to \texttt{0}, B \to \texttt{10}, C \to \texttt{11}\}$. Two of the most commonly used entropy codings are Huffman coding and arithmetic coding.

2.2.1 Huffman Coding

Huffman coding works by recursively combining low-probability symbols into a set of symbols that share the same prefix. For example, when given the input $\{A \to 0.8, B \to 0.15, C \to 0.05\}$, Huffman coding combines the symbols B and C into a group that will share the prefix `1` and which, as a whole, has probability 0.2 .



For this project, I use my own implementation of Huffman coding in the `Huffman` class available for viewing at `libraries/codes.py`.

Consider moving the bit about implementation to another section

2.2.2 Arithmetic Coding

Arithmetic coding is a method where the cumulative probability of symbols is calculated (i.e. $\{\emptyset \to 0, \{A\} \to 0.8, \{A, B\} \to 0.95, \{A, B, C\} \to 1\}$) and a range within the interval $[0, 1)$ is given in a binary format to indicate one of the symbols.

My implementation of arithmetic coding is also available at `libraries/codes.py`.

Finish writing about this

Consider moving the bit about implementation to another section

2.3 Regularity and the importance of context

Let's now imagine yet another instance of the game. In this version, the data Trent gives to Carol is a sequence of English letters making up a novel. In this case, our symbols are again letters, and the previously used code for randomly chosen English letters can be used with some success. However, that code is no longer optimal.

This is because the symbols given by Trent are no longer independent or identically distributed, that is, letters cross-correlate. For example, whenever the letter Q appears in the sequence of symbols, there is a high probability that the following symbol will be U, by the structure of English. This **regularity** can be exploited in Carol and Dave's choice of code, so that whenever the symbol Q is encountered, the symbol U is given a shorter code within that **context** to reflect its high probability.

In the extreme case where each Q is necessarily followed by a U, Carol and Dave's code may not assign U a code at all, as it can be immediately inferred whenever a Q is encountered. In this case, it can be said that the regularity of QU has been *abstracted out* in the encoded version of the text. The result of this abstraction is a version of the text with less regularity and with higher entropy, that is, a more information-dense representation.

Talk a little bit more about why we should expect compressed data to look like noise.

2.4 Prediction and compression

Mention (and maybe include) Shannon's reasoning with the clones.

Somewhere in this report, include and explain Shanon's predictor-interpreter model and visual.

Mention that the value of prediction is that you only have to encode how reality deviates from the prediction

Maybe mention predictive processing, confabulation

Decide on what to include from the "Practice" file

3. Investigation

This chapter documents my application and investigation of some of the ideas presented in the Overview.

3.1 Dataset

Since this project examines the compression of natural language, it requires some raw material to work with. For this purpose, I use a collection of texts downloaded from Project Gutenberg, as they're easy to download and all in the public domain.

I initially manually download 97 of the top 100 texts (3 of them not being available as txt files), but as I needed a larger dataset for section 3.5.1, I automated this process and downloaded all 96200 texts available on the website. The set of 97 texts was assumed to contain only English texts, but luckily this was not the case, as will be explored in section 3.5.

Although python libraries such as `nltk` provide corpora for natural language processing, they are usually much smaller (only 18 texts in `nltk`'s case) and I opted for directly using Project Gutenberg for more control.

The downloaded dataset is viewable at `FYP/Data` in the project repository and the scripts used to obtain it are in `Code/dataset`.

3.2 Benchmarking compression algorithms

For a rough bound on what's practically possible given commonly used tools, I start by analyzing the performance of various existing non-text-specific compression algorithms on the Project Gutenberg dataset. The result of this benchmarking is displayed in table 3.1.

Algorithm	Size (bytes)	Compression ratio
LZMA	20815738	3.745084
bzip2	21055590	3.702422
gzip	28702450	2.716029
zlib	28840353	2.703042
No compression	77956688	1.000000

Table 3.1: Compression algorithm benchmarks

The best performing compression algorithm is the Lempel–Ziv–Markov chain algorithm (LZMA), with a compression ratio of 3.745. Given that the vast majority of characters in Gutenberg texts are represented in a single byte (except for unicode characters), this compression method indicates that, at most, each character in these texts contains on average $8/3.745 = 2.14$ bits of information.

This figure is obtained empirically and establishes an upper bound on the amount of information per character in English text, but how do we establish the actual amount of information per character and thereby estimate the best compression ratio that can be hoped for? In other words, how do we calculate the entropy of English text?

3.3 N-gram prediction

As mentioned in sections 2.1 and 2.3, the entropy of a single letter in English is 4.14 bits, but that is when the letter is considered out of context.

In his paper "Prediction and entropy of printed English" Shannon (1951) checks the effect of context by giving subjects a varying amount of context from a text (1 to 100 previous letters) and asking them to guess the next letter, keeping track of how often the first guess, second guess, etc. is correct. In the case where subjects were given 100 previous letters, they guess accurately on their first try 80% of the time and on their second try 7% of the time. After some analysis, he estimates that - when a letter is considered within its context - English contains 0.6 to 1.3 bits of information per letter.

To check this value, I apply the method of analysis given in the 1951 paper using modern tools.

The method considers n-grams (groups of n consecutive characters) in a text and defines the entropy of n-grams in English as

$$G_n = -\sum_{b \in B_n} p(b) \log p(b)$$

where B_n is the set of all possible n-grams and $p(b)$ is the frequency with which the n-gram b occurs in English text. For example, $G_1 = p('e') \log p('e') + p('t') \log p('t') + ...$ = 4.14 is the entropy of 1-grams, that is, single letters.

Building on this, Shannon evaluates the average entropy of the last character in the n-gram given the previous $n-1$ characters. This is

$$\begin{aligned}
F_n &= \sum_{r \in B_{n-1}} p(r)(\text{entropy of the last character } c \text{ given the prefix } r) \\
&= \sum_{r \in B_{n-1}} p(r)(-\sum_{c \in C} p(c|r) \log p(c|r)) \\
&= -\sum_{r \in B_{n-1}, c \in C} p(r,c) \log p(c|r) \\
&= -\sum_{r \in B_{n-1}, c \in C} p(r,c) \log p(r,c) + \sum_{r \in B_{n-1}, c \in C} p(r,c) \log p(r) \\
&= -\sum_{b \in B_n} p(b) \log p(b) + \sum_{b \in B_{n-1}} p(b) \log p(b) \\
&= G_n - G_{n-1}
\end{aligned}$$

F_n is the differential or marginal entropy at the last letter of an N-gram, that is, how much uncertainty there is about that letter when the previous $n-1$ letters are known, and correspondingly how much information it carries.

To estimate the values of G_n and F_n for various n, I concatenate all texts from the Gutenberg dataset and use python's `Counter` tool to track how many times each n-gram occurs. The counts of n-grams are then used to calculate the n-gram entropies G_n and their differences give the values of F_n, The results are shown in tables 3.2 and 3.3.

N	N-gram entropy	Marginal entropy
1	4.668763	4.668763
2	8.174800	3.506037
3	11.028968	2.854169
4	13.345969	2.317001
5	15.323687	1.977718
6	17.094342	1.770655
7	18.704355	1.610013
8	20.150312	1.445958
9	21.415078	1.264766
10	22.484581	1.069503

Table 3.2: Entropies and marginal entropies for substrings which are n characters long

N	N-gram entropy	Marginal entropy
1	11.348724	11.348724
2	18.610973	7.262249
3	22.132126	3.521153
4	23.172127	1.040001

Table 3.3: Entropies and marginal entropies for substrings which are n words long

Table 3.2 shows the entropies for substrings of the text which are n letters long, along with the marginal entropy at the last letter, conditional on knowledge of the previous $n-1$ letters, while table 3.3 shows entropies for substrings which are n words long, along with the marginal entropy at the last word.

As can be seen in both tables, more knowledge about the context in which a letter or word appears decreases the information content and increases the predictability of that letter or word.

For example, the entry 1.04 on the fourth line of table 3.3 indicates that when three consecutive words of a text are known, the fourth word contains approximately 1 bit of information, that is, there are on average only two candidates for what the fourth word could be.

The data displayed in table 3.2 empirically validates the estimate of 0.6-1.3 bits per letter. As English text is often encoded in 1 byte per letter, this indicates an ideal compression ratio (ratio of uncompressed text size to compressed text size) of approximately 8.

3.3.1 Practical application

Next, I examine Shannon's concept of the Ideal N-gram Predictor, which he conceptualizes as a lookup table with the keys being the set of all possible [n-1]-grams and the values being an ordering of all possible following symbols ranked by their likelihood. For example, a 3-gram predictor would have among its keys the 2-gram "qu" and in the value corresponding to that key all possible following letters starting with "e" and going down in terms of likelihood (as "que" is the most common 3-gram in English text that starts with "qu").

In the original formulation, a text is compressed for transmission by replacing each letter in it with its rank given the previous [n-1]-gram. For example, the word "queue" may be transmitted as the numbers 15,1,1,1,1, indicating that "q" was the 15th most likely letter at the start of a text, "u" was the 1st most likely letter to follow, etc.

I test the practical application of this idea by using Huffman and arithmetic codes instead of integers, assigning the most likely symbols (within a given context) the shortest codes, and check to see how well these implementations perform in terms of compression ratio. To do this, I create the following classes in my code

- A `Code` superclass, subclassed by the `Huffman` and `Arithmetic` classes, each of which handles the construction of a code given the probabilities of the symbols, as well as handle encoding and decoding of symbols.
- A `Predictor` abstract class, subclassed by `NGramPredictor` (and in section 3.4 by `LSTMPredictor` and `RNNPredictor` as well). `Predictor` objects must have a `train()` method which trains the predictor on a basis text from which it detects the regularities in the language and a `probabilities_given_context()` method which takes a context as its input and outputs a mapping of symbols to their probabilities within that context.
- A `Compressor` class. Objects of this class are initialized with a `Predictor` and a `Code` object, and provide `encode()` and `decode()` methods, leveraging its associated `Predictor` to estimate the probabilities of symbols within a given context, and its associated `Code` to generate the corresponding entropy coding for that context and encode/decode the input text.

In particular, the `NGramPredictor` class is initialized with

- A basis text from which the predictor infers the patterns of the language of the text (and which it is trained on immediately)
- A window size. This sets the value of n for which it will examine n-grams in the text. For example, a window size of 6 means that the predictor will construct a lookup table mapping 5-grams and to the probabilities of different symbols being the completion of that 5-gram into a 6-gram.

The code for the libraries I create for this task can be found at `Code/libraries` and an application of them at `Code/n_gram`. In the first, I implement the `Huffman` and `Arithmetic` classes from scratch. I initially considered using functions defined in the python library `bitarray` for Huffman codes (namely `decodetree` and `huffman_code`), but opted not to as these did not provide all the functionalities I wanted and did not conform to the desired interface. Through some experimentation, it also seems that my implementation is faster to run that `bitarray`'s.

To test my code, I begin by creating a `Predictor` object trained on War and Peace with a window size of 6, and from it create the corresponding `Compressor` object (I will refer to this as WPC). I then use WPC to encode and decode simple phrases to ensure that the operations reverse correctly. The `encode` method takes a sequence of bytes (ideally in natural language) and compresses them using the lookup table constructed from the 6-grams found in War and Peace. Figure 3.1 illustrates a simple use case.

```
>>> inigo_text     = b'Hello. My name is Inigo Montoya. You killed my father. Prepare to die.'

>>> inigo_encoding = war_and_peace_compressor.encode(inigo_text)
>>> print(len(inigo_encoding), inigo_encoding.hex())
53 fa25b6d6760fb9c1ffff7082b8fffffe5bfffd2dfdcc1f718efffc37fff1ae0aec1fde23ffff9341b30a062e4ffff9be8d35199406

>>> inigo_decoding = war_and_peace_compressor.decode(inigo_encoding)
>>> print(len(inigo_decoding), inigo_decoding)
70 b'Hello. My name is Inigo Montoya. You killed my father. Prepare to die.'
```

Figure 3.1: Using WPC to encode and decode a few sentences.

It is also possible to use the `decode` method on randomly generated bytes to obtain text that follows the regularities which the predictor has learned, as in figure 3.2.

```
>>> print(war_and_peace_compressor.decode(randbytes(50)+b'\0', context=b'Nicholas').decode('utf8'))
"Nicholas, shrewd look, said in commanded. But unableness came of peasantly appointing to get as he shoulders,
Princesses,” said she"
```

Figure 3.2: Using the `decode` method on a randomly generated input

The performance of a `Predictor` object depends on the text it has been trained on, and words and expressions which it is familiar with would therefore have shorter representations in its output encoding. For example, figure 3.3 shows the surprisal of `war_and_peace_predictor` when trying to predict the characters in the first paragraph of The Adventures of Sherlock Holmes.

A few observations on figure 3.3,

- For the first few symbols, the predictor performs badly as it has no context from which to infer the next symbol.
- The predictor performs especially badly for proper nouns. This is to be expected, as the words "Sherlock Holmes" and "Irene Adler" are not ones which it would have come across in War and Peace.
- The letter "k" at the end of "Sherlock" was expected with 0.5 probability by the model. On further investigation it seems this is because the 5-gram "erloc" appears in W&P twice, in the words "interlocutor" and "interlocking", and half of these have a "k" following "erloc".
- The predictor often performs better on the ending of words than their beginning.

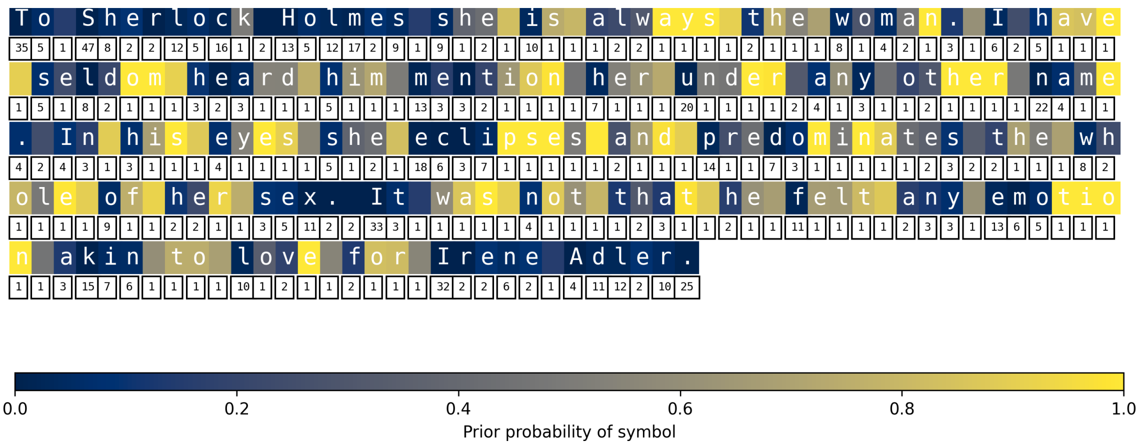


Figure 3.3: Surprisal of the `war_and_peace_predictor` at each symbol in the opening passage of Sherlock Holmes. The prior probability given by the model to each symbol is indicated by its color, while how highly it ranked this symbol in terms of probability is indicated by a number below the symbol.

- For commonly found phrases such as "was not that he", the predictor's first guess for the following letter is very often correct.

Figure 3.3 gives the probability rank of each symbol underneath it. For example, a rank of 2 indicates that the symbol would have been the predictor's second guess based on the previous 5-gram.

Figure 3.4 and table 3.4 show the rank distribution when encoding the entire text of The Adventures of Sherlock Holmes.

Rank	1	2	3	4	5
Probability	0.568561	0.146148	0.075059	0.046406	0.031536
Human performance	0.58	0.19	0.05	0.01	0.04
Zeta distribution ($6/\pi^2 r^2$)	0.607927	0.151982	0.067547	0.037995	0.024317

Table 3.4: In the first row, this table shows how often the WPC is correct in its first guess, second guess, etc. for the last letter in a 6-gram. The second row gives values given by Shannon (1951) obtained through experiments with English speakers similarly given 5 letters of context and asked to predict the full 6-gram. The third row gives the values of the zeta distribution with parameter 2 for comparison.

The predictor trained on War and Peace accurately predicts a symbol based on the previous 5-gram on its first try 56.86% of the time, and the frequency of the ranks follows a Pareto distribution with a scale parameter of 1 and a shape parameter of 1, i.e. the probability density function is $1/r^2$. The values for the discrete version of this distribution (the zeta distribution with parameter 2) is included in table 3.4 for comparison.

When initializing a `Compressor` object, one has a choice between using Huffman coding or arithmetic coding (by passing either the `Huffman` or the `Arithmetic` class as a parameter). These differ in how they assign binary codes to symbols, with Huffman coding being more economical - the average code length being usually minimal - and arithmetic coding being "fairer", in the sense that a symbol with 2^{-n} probability of occurring would have a code that is approximately n bits long, that is, it preserves the probability distribution of the symbols.

For example, if symbol "a" occurs with probability 0.9 and symbol "b" appears with probability 0.1, Huffman coding would assign them the codes "0" and "1" respectively, while arithmetic coding would assign them "0" and "11101". Because of this difference, decoding a random sequence of bits using the Huffman code would result in an output which has an equal number of "a"s and "b"s, whereas decoding the same input using the arithmetic code would result in an output that is approximately 90% "a"s and 10% "b"s per the original distribution.

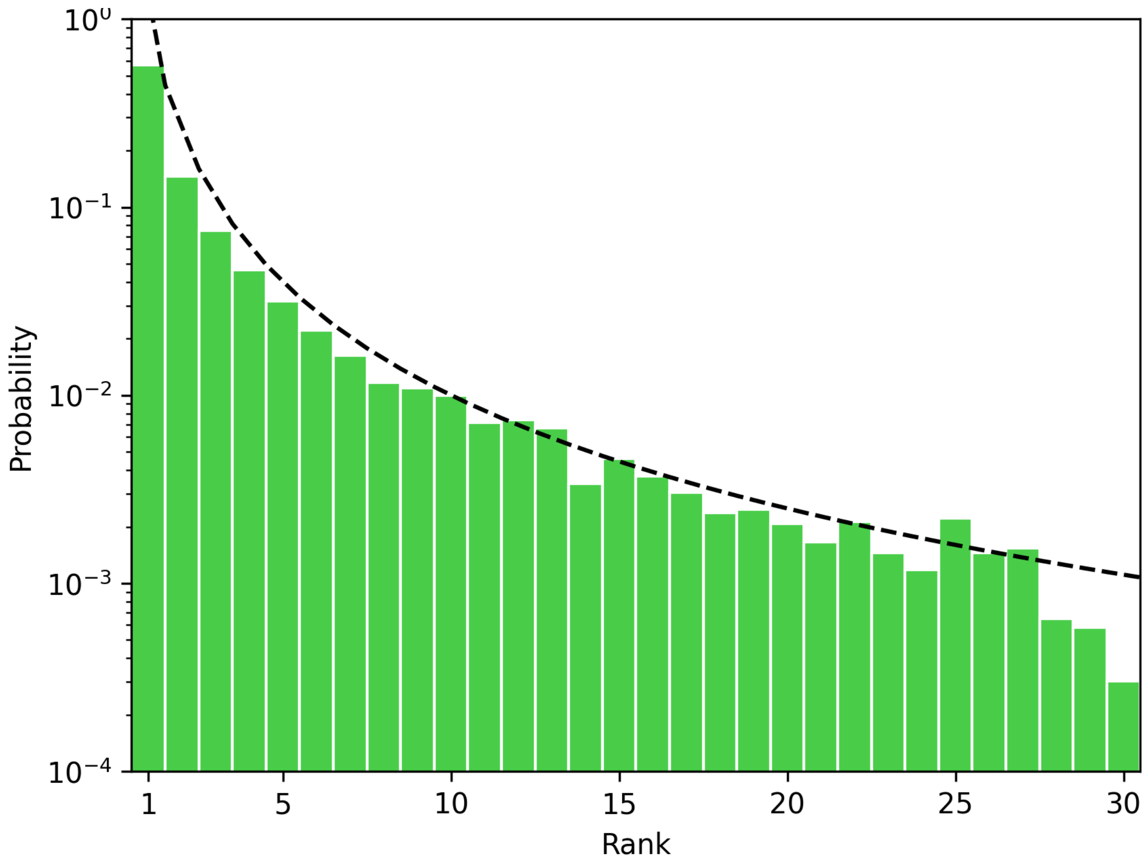


Figure 3.4: In this histogram, each bin represents a symbol rank from 1 to 30, and the height of the bin represents the frequency at which this rank occurs in the WPC's encoding of The Adventures of Sherlock Holmes. The dashed line shows the probability density function for the Pareto distribution with a scale of 1 and a shape of 1.

Table 3.5 shows the performance of WPC on some input texts using each code. For all examined texts (including Don Quixote, which is not in English), the compressed version is no larger than the original, and in fact this method typically performs better than gzip and zlib (see table 3.1). As expected, Huffman coding yields significantly better results than arithmetic coding. Note also that (apart from its own basis text), WPC performs best on the text of Crime and Punishment, which is the text that is closest to it historically and geographically, and which is therefore likely to contain many of the same regularities.

Text	Huffman compression ratio	Arithmetic compression ratio
War and Peace	4.258278	3.404772
Crime and Punishment	3.019220	2.068427
Alice's Adventures in Wonderland	2.946588	2.028157
Middlemarch	2.895632	1.950565
The Count of Monte Cristo	2.894389	1.947010
Don Quixote	2.715589	1.833225
The King James Version of the Bible	2.284923	1.543722
The Complete Works of William Shakespeare	1.984536	1.289506

Table 3.5

3.3.2 Conclusions and future work

Mention: Shannon tries doing prediction backwards and it works pretty well. This is possible with the ‘Predictor‘ class. Is it possible to combine both forwards and backwards prediction?

Mention the potential application of the context given to ‘Predictor‘ objects not being previous characters, instead being an indicator of topic or part of speech for example

Mention confabulation

Compare found zeta distribution to Zipf’s law. Why should the parameter be approximately 2 for a 6-gram? What’s the general law here?

3.4 Machine learning-based compression

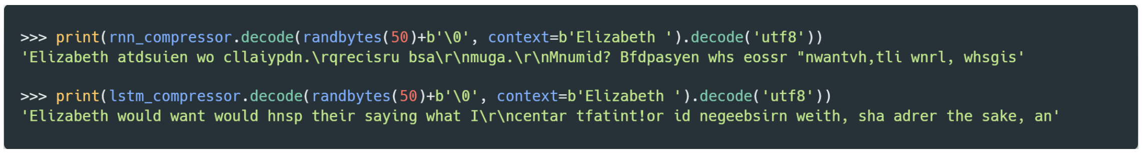
As discussed in the previous sections, any mechanism which can detect regularities in a stream of data and predict future observations (symbols) from past ones (context) can also be used for compression by using it to estimate the probabilities of potential future observations and using these to assign shorter codes to the most likely of them.

While n-gram language models are very simple and easy to train, they are limited by their pre-specified (and often short) context, namely the previous [n-1] symbols in a text, with symbols being either characters or full words. For this reason, statistical models such as n-gram language models have generally been superseded by neural network-based models such as recurrent neural networks (RNNs), long short-term memory models (LSTMs), and most recently large language models (LLMs) such as BERT and GPT, which can retain information across larger contexts and better capture long-range dependencies in a text.

For RNNs, this is accomplished by having the network’s previous outputs act as additional inputs when it observes a new symbol, giving it a simple form of memory. An improvement on this model is LSTMs, which are a subtype of RNNs that can learn to select relevant information to keep within its memory, and which generally outperform simple RNNs. Lastly, transformer-based LLMs use an attention mechanism to weight the relative importance of all words in the input data when making a prediction.

In this section I perform simple experiments with one simple RNN and one LSTM, once again subclassing the `Predictor` class and plugging it (as well as an entropy coding) into a `Compressor`. The code for this section can be found at `Code/machine_learning`.

As before, it is possible to feed the `decode` method noise to see what regularities the model has learned about.



```
>>> print(rnn_compressor.decode(randbytes(50)+b'\0', context=b'Elizabeth ').decode('utf8'))
'Elizabeth atdsuien wo cllaiypdn.\rqrecisru bsa\r\nmuga.\r\nMnumid? Bfdpasyen whs eossr "nwantvh,tli wnrl, whsgis'

>>> print(lstm_compressor.decode(randbytes(50)+b'\0', context=b'Elizabeth ').decode('utf8'))
'Elizabeth would want would hnsp their saying what I\r\ncentar tfatint!or id negeebsirn weith, sha adrer the sake, an'
```

Figure 3.5: Using the `decode` method on a randomly generated input

With both models the text is much less coherent than with the n-gram model, although the models have still learned which byte values are characters, how often spaces are used, some common collocations, and in the case of the LSTM model some common words. It is likely that with more training or a wider context window the models would be better able to capture the regularities in English, but due to constraints on time and computational power it was not possible to test this in more depth.

As part of the process of training, my code serializes and saves the models periodically. One of the significant advantages of machine learning models over n-gram based models is that the trained version is much smaller (approximately 4 mb in the case of the LSTM and 1.5 mb in the case of the RNN).

My implementation of these models was unfortunately too slow to be practicable on large texts, although I suspect efficient implementations are possible. For an estimate of compression ratios, I use an excerpt from Sherlock Holmes, which neither model was trained on. This test indicates compression ratios of 1.253 and 2.008 for the RNN- and the LSTM-based compressors respectively.

3.4.1 Discussion

Mention autoencoders and latent spaces

Write about potential use of LLMs, at least in future work subsection

3.5 Co-compression

In sections 3.3 and 3.4, we explored leveraging an understanding of a text's regularities for compression. This section explores the opposite idea, namely using an existing compression algorithm as a black box to understand regularities in text.

One point of commonality between compression and comprehension is the need for parsimony. This is expressed by Occam's razor, which states that "entities must not be multiplied beyond necessity" and commonly understood to mean that "the simplest explanation is usually the best one".

Compression algorithms, for example the family of Lempel-Ziv algorithms, in fact operate by eliminating the multiplication of entities through the creation of a *codebook*, a mapping between often-repeated bits of data and what is essentially an abbreviation for each. If the relationship between compression and comprehension holds, we should expect to be able to use an existing compression algorithm like LZMA to get a rudimentary understanding of text.

Jiang et al. (2023) showed that it's possible to use gzip for text classification, based on

> "the intuitions that
>
> 1. compressors are good at capturing regularity;
> 2. objects from the same category share more regularity than those from different categories"

To illustrate this using the same dataset of 97 Gutenberg texts, I use the following scoring methods to estimate the similarity of a pair of texts:

$$add(y|x) = \frac{C(xy) - C(x)}{C(y)}$$

where $C(t)$ is the compressed length of t expressed in bytes, and xy is the concatenation of the texts x and y. The function $add(y|x)$ is then a measure of the additional information given by y when x is taken as a known basis.

To give an intuition for why the measure is defined in this way, it's helpful to use an example. Suppose that x is the text of the complete works of Shakespeare and that y is the text of Romeo and Juliet. Since x contains y as a substring, we should expect that a good compression algorithm would compress the concatenation xy in little more space than is required for just x, as the entire text of y can be signified with a single symbol in the codebook, compressed once, and referred to twice.

Because of this, we should expect the value of $add(y|x)$ in this case to be close to 0, corresponding to the fact that y does not really add information to x. Conversely, since the complete works of Shakespeare contain a lot of information not contained in just Romeo and Juliet, we should

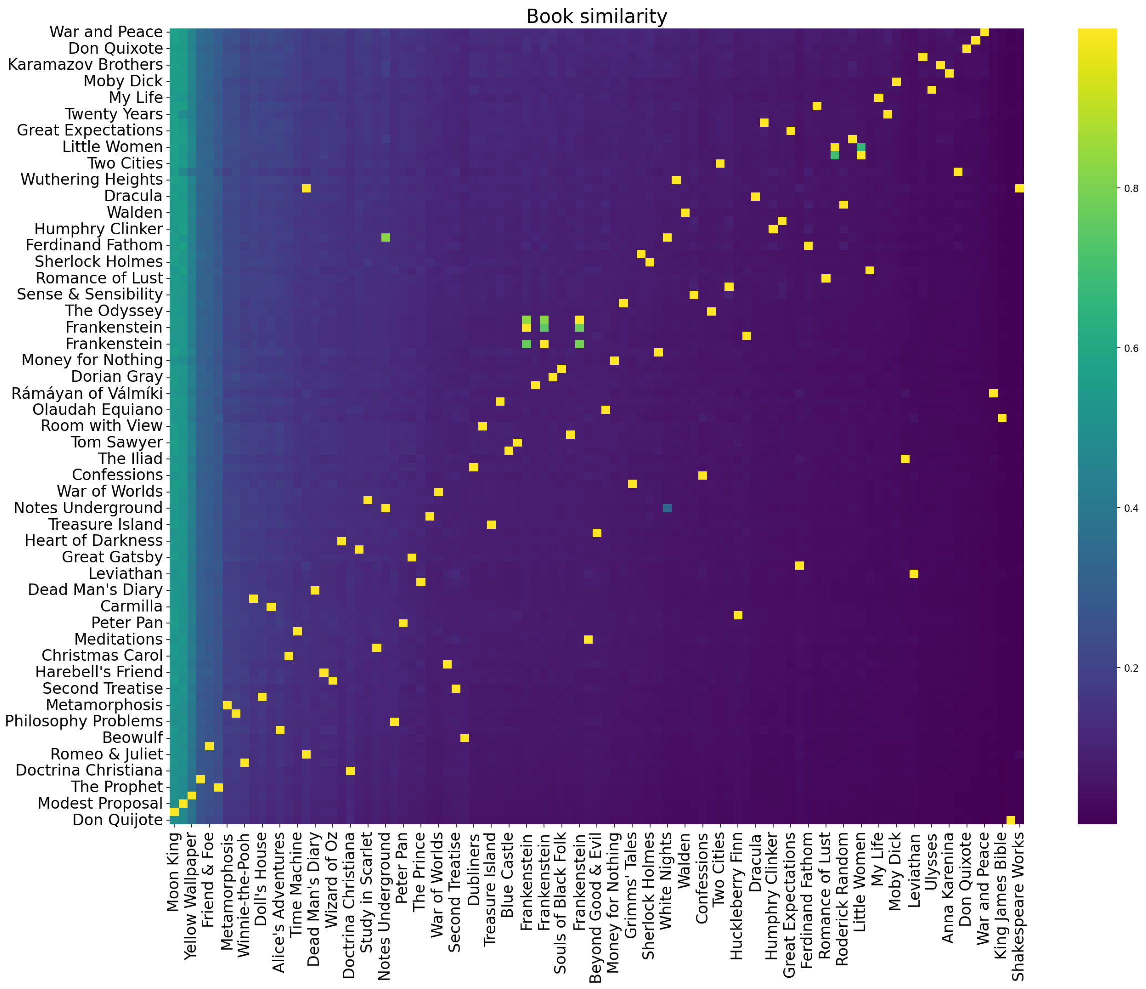


Figure 3.6: Similarity scores for pairs of texts. The texts on the vertical axis are ordered most-predictive-first, and those on the horizontal axis are ordered most-predictable-first.

expect $add(x|y)$ to be close to 1, indicating that our compression algorithm's knowledge of the text of Romeo and Juliet only makes a small dent in the number of additional bits it needs to represent the works of Shakespeare.

The measure used above contrasts with Normalized Compression Distance (NCD) as used by Li et al. (2004) , in that it is directional, i.e. that $add(x|y)$ is not necessarily equal to $add(y|x)$.

We may also define a more intuitive similarity score as follows

$$similarity(y|x) = 1 - add(y|x) = \frac{C(x) + C(y) - C(xy)}{C(y)}$$

This score giving a measure of the similarity of y to x. In this case, $similarity(a|b) = 1$ would indicate that a is fully described by b (as it contains no additional information), and $similarity(a|b) = 0$ indicates that a is unlike anything seen in b. Again, note that $similarity(y|x)$ is not necessarily equal to $similarity(x|y)$.

This similarity score is plotted for pairs of texts in figures 3.6 and 3.7, each plot giving x on the horizontal axis and y on the vertical axis, the color of the cell representing $similarity(x|y)$. In figure 3.6, the texts on the vertical axis are sorted by how predictive they are on average, and the texts on the horizontal axis are sorted by how predictable they are. In figure 3.7, the same texts are instead sorted by file size on both axes.

The figures were produced the `co_compression/stats.py` script in the repository for this project. Creating the figures involved experimenting with each of 4 compression algorithm libraries (`zlib`, `lzma`, `gzip`, and `bz2`) to find the most suitable for co-compression. Since one of the

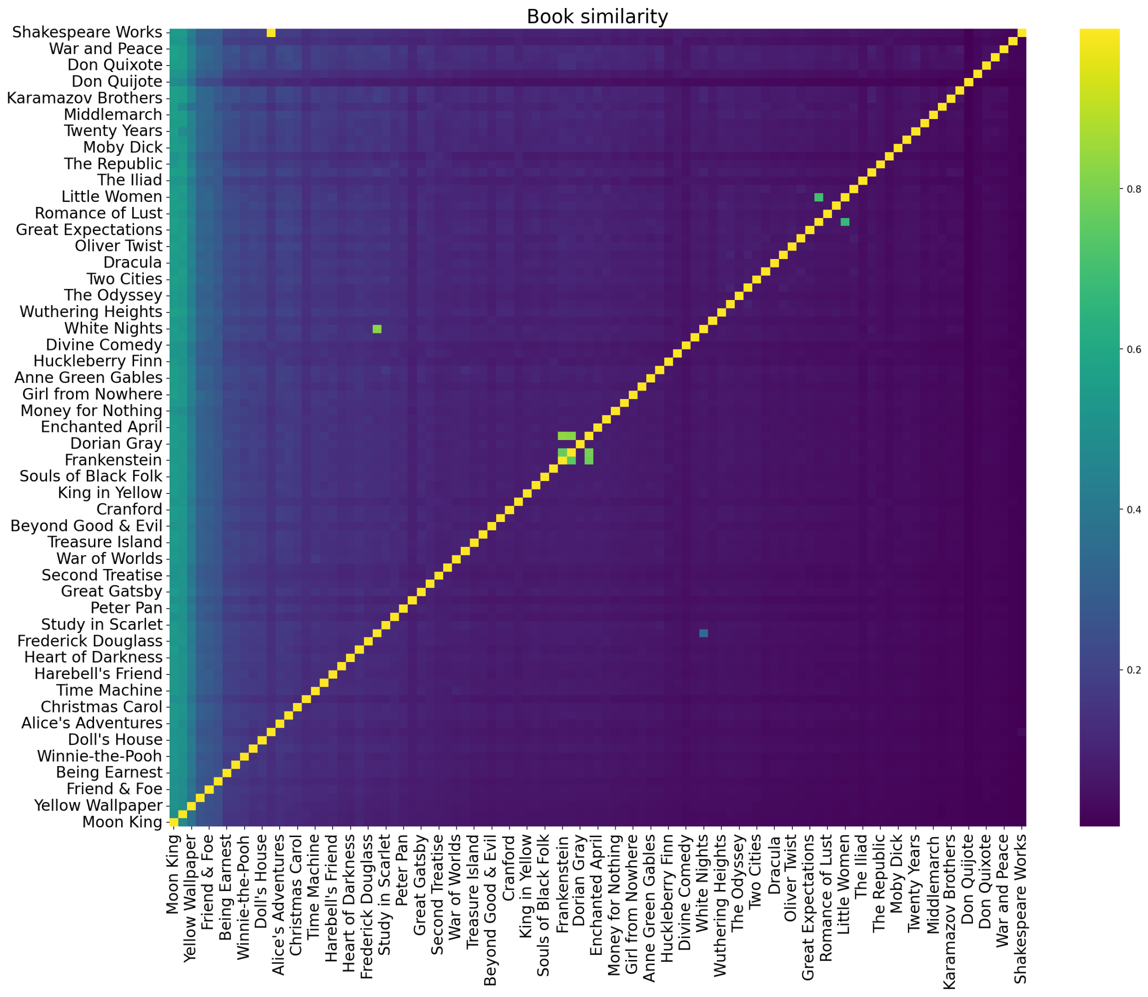


Figure 3.7: Similarity scores for pairs of texts. The texts on both axes are sorted by file size.

functions that `stats.py` performs is to co-compress all possible pairings of texts from a set of 97, the results are serialized and saved regularly to avoid need for recalculation. The script uses the `seaborn` and `matplotlib` python libraries for generating the figures, and `pandas` is used to process and tabulate the data.

The following observations can be made based on figures 3.6 and 3.7:

- In general, larger texts are more predictive.
- In general, smaller texts are more predictable.
- There are a few very bright dots which do not lie along the diagonal in 3.7. These appear because the dataset includes three versions of Frankenstein which have a high similarity score with each other, as well as the texts for Romeo and Juliet, and the Complete Works of Shakespeare. For this last pair, the latter strongly predicts the former but the former only weakly predicts the latter.
- There is a strong horizontal dark line in both graphs, which represents a text that is unpredictable regardless of what other text it is paired with. On inspection, this turns out to be Don Quijote, which is in fact unlike the other texts because it is in Spanish, whereas most other texts are in English.
- As can be seen in 3.6, when texts are sorted most-predictive-first on the y-axis and most-predictable-first on the x-axis, texts generally lie along the diagonal. This indicates that, within a set of texts, there is a trade-off between how predictive a text is and how predictable it is. At least part of this effect has to do with file size, as larger texts tend to contain a larger set of the possible words and expressions of a language.

- There is a visible pattern of strong vertical and horizontal lines in 3.7 (i.e. there is a low level of noise), indicating that texts tend to be "generally predictive" or "generally predictable" within a given collection.

3.5.1 Practical application

As can be seen in the previous figures, it is always the case for any two pairs of texts x and y that "the whole is less than the sum of its parts", i.e. that $C(xy) < C(x) + C(y)$. This opens up the question of whether it is possible to obtain a better compression of y by encoding only that information which it adds to x. If this is possible, we should expect that this information should be of the size $C(xy) - C(x)$ which, by the previous equation, is necessarily less than $C(y)$.

Intuitively, this means that if two parties - Carol and Dave - each have a copy of the works of Shakespeare, and Carol wants to send over Romeo and Juliet, she can simply encode it by giving the range of pages on which it appears in their shared reference text, a much shorter representation than sending the actual text of Romeo and Juliet.

Is this possible to implement in practice? The fact that Lempel-Ziv algorithms work by going through the data in one pass and creating a codebook as they go indicates that $D(xy)$ should contain $D(x)$ as a prefix, where $D(t)$ stands for the compression of t. This is, in fact, more or less the case with LZMA, and I was able to create a simple tool that implements this idea, available at `libraries/co_compressor.py` .

The `CoCompressor` class I implement is instantiated with two arguments: a reference text r and a compression algorithm. Its `compress` method takes a text t to compress and outputs $D(rt)$ with the prefix $D(r)$ removed, and its `decompress` method takes a text compressed in this way, prefixes it with $D(r)$ and decompresses it, reversing the process.

Co-compressors instantiated in this way differ in their power depending on the choice of reference text. See figure 3.8 below for a comparison of the performance of co-compressors trained on different reference texts.

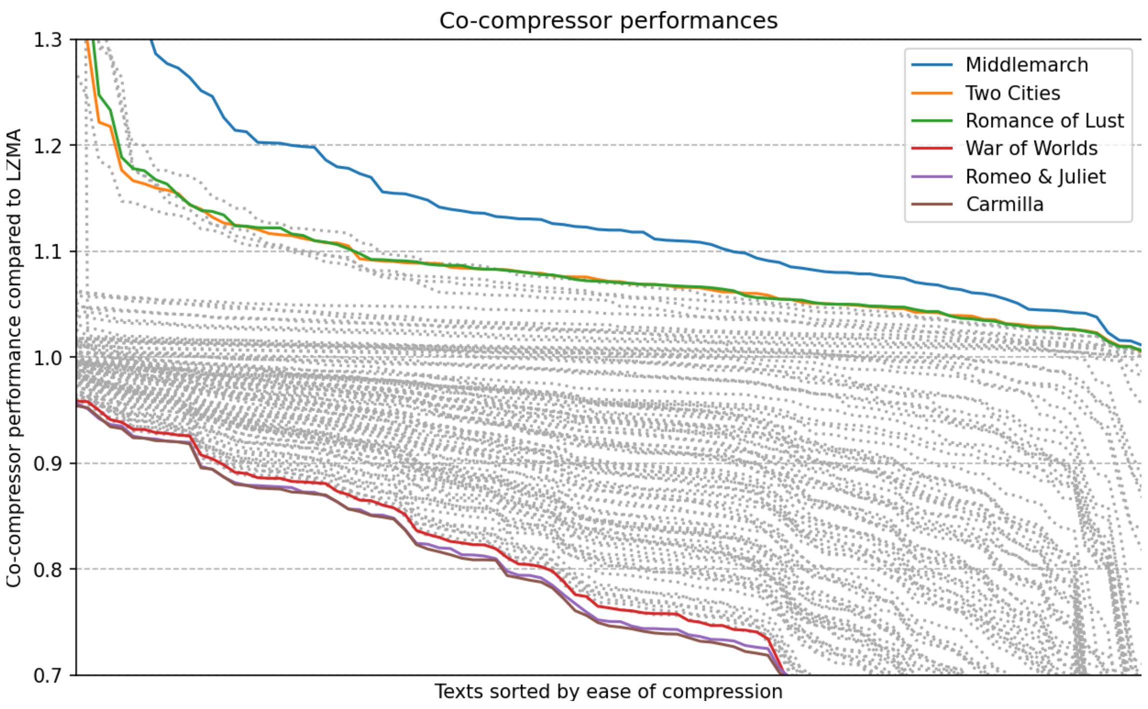


Figure 3.8: The performance of instances of the `CoCompressor` class trained on 97 different texts. Each line represents a `CoCompressor`. The value on the vertical axis is the Relative Compression Ratio (RCR), calculated as the size of the encoding produced by naive LZMA divided by that produced by the `CoCompressor`, while the horizontal axis goes through possible inputs for comrpession.

There are a few remarkable things about this figure:

- Each co-compressor shows a tan- or sigmoid-like curve, performing exceptionally well on texts which are very similar to it (left side of the plot) and steeply dropping down in performance for sufficiently different texts.
- There is very little criss-crossing between the lines in this plot. That is, for any pair of co-compressors, one will usually outperform the other *on every input* in the dataset. Why there should be a strict hierarchy of this sort is not clear, as one might expect that a specific co-compressor would be specialized for texts which are similar to it but not for others.
- The best performing co-compressor in this dataset is the one trained on Middlemarch (performing 12% better than naive LZMA in the median case), followed by A Tale of Two Cities (7.3%) and The Romance of Lust (7.2%).

It is tempting to explain away the unusual performance of the Middlemarch co-compressor (henceforth CC_{MM}) as an artifact of the chosen dataset, that it might be situated (historically or otherwise) "in the middle" of the dataset, sharing some features with both preceding and following literary works.

One strong piece of evidence against this is that, as noted above, the performance of co-compressors is strictly hierarchical, and this effect extends all the way to the left of the plot where the co-compressor is fed the text it performs best on (that is, its own text) as its input. Examining this, we notice that

$$RCR_{MM}(MM) > RCR_t(t) \qquad \forall t \in G$$

where $RCR_a(b)$ is a measure of the performance of the co-compressor trained on text a when fed input b (as defined in figure 3.8), and G is the Project Gutenberg dataset. That is, CC_{MM} is not only the most performant co-compressor on all the texts in the dataset, it is also the co-compressor that performs best on its own reference text.

This observation gives a simple way to find other good reference texts for co-compressors: one simply calculates $RCR_t(t)$ for each candidate and finds values of t that yield the highest performance.

I implement this method on a much larger dataset of 92600 texts downloaded from Project Gutenberg, this time only using `CoCompressor` objects trained on a text to compress the same text, and measure the relative compression ratio (RCR) for each text, where

$$RCR_a(b) = \frac{length(LZMA(b))}{length(CC_a(b))}$$

that is, the performance of the co-compressor relative to naive LZMA.

Figure 3.9 shows how the 92600 texts in this dataset cluster in terms of their RCR. Note that the vertical axis is logarithmic. If the hypothesis that texts with a high RCR are good co-compressors, we should expect some of them to outperform Middlemarch if figure 3.8 is re-plotted to include these texts.

This is, almost, what we find in figure 3.10. The 30 texts with the highest RCR were selected for inclusion in this plot. As can be seen here, all of these outperform naive LZMA to some degree, with Middlemarch still performing best in the median case. Since this method of measuring RCR works, it may be possible to use it to automatically construct a basis text for a co-compressor that gives a better compression ratio.

3.5.2 Conclusions and future work

Idea: see if there's a chain of bootstrapping. Middlemarch compresses everything well, but is there something that compresses Middlemarch well? If so, use that as an earlier step in the bootstrapping

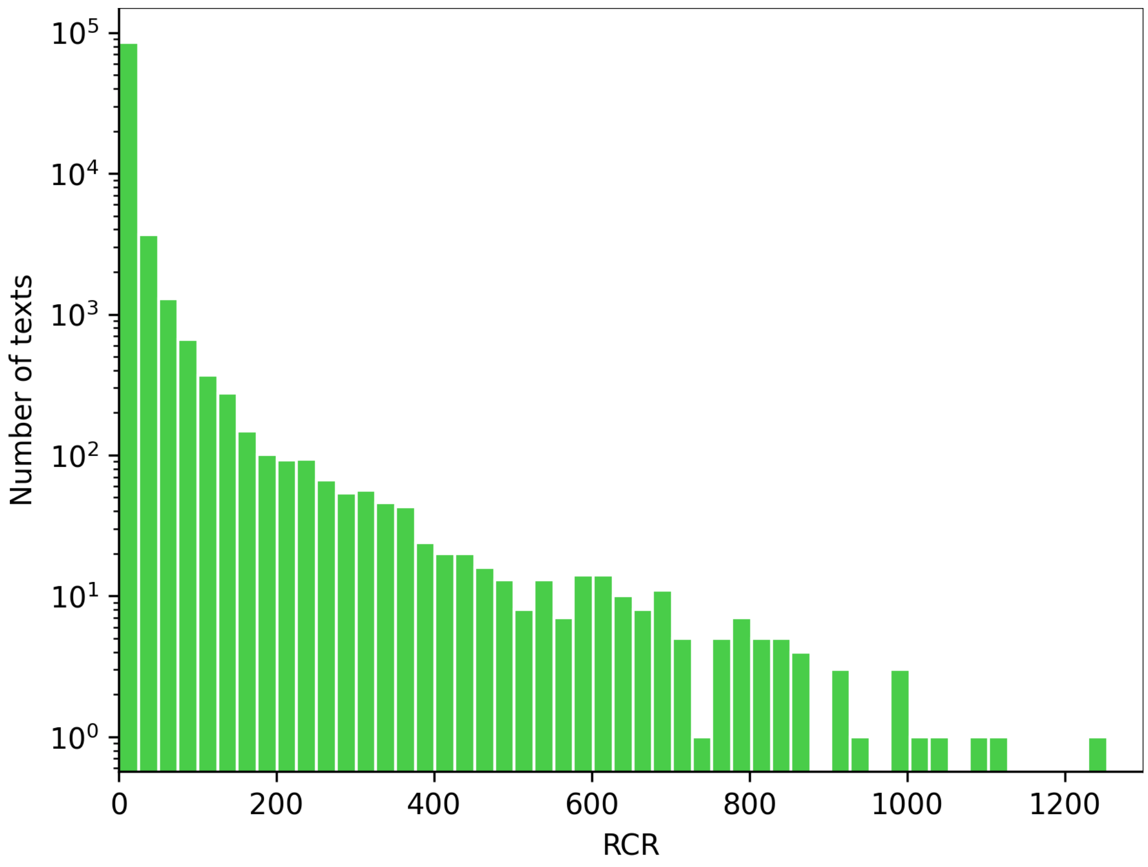


Figure 3.9: A histogram representing the self-compression score (RCR) of 96200 Project Gutenberg texts.

Idea: how well does the first half of Middlemarch compress the second half? If the resulting compression is short enough, this facilitates bootstrapping even more: use part 1 to compress part 2, use the total to compress another text. You could also do it in more splits.

Idea: co-compressors take compressors as one of their initializers, but they are themselves compressors. Is there any use in stacking them?

Idea: try to add some noise to a compressed version of something and see if it decompresses into anything

Idea: can you automatically generate a text ex nihilo that will be a good co-compressor using RCR as a metric?

Potential explanation for good co-compressors: file size has sweet spots. There is a repeated pattern of performance getting worse and then suddenly better as file size grows, investigate.

Idea: if the `CoCompressor` class takes a compression algorithm and improves it, can this method be used recursively on itself (starting with some very naive compression) to get something that performs significantly better?

Why Middlemarch??

Idea for practical compression: start compressed file by indicating what you want to use as a basis text, then give compressed version. Potentially, stack these: if you need basis A to get B and basis B to get C, have the file format allow passing both B and C using basis A.

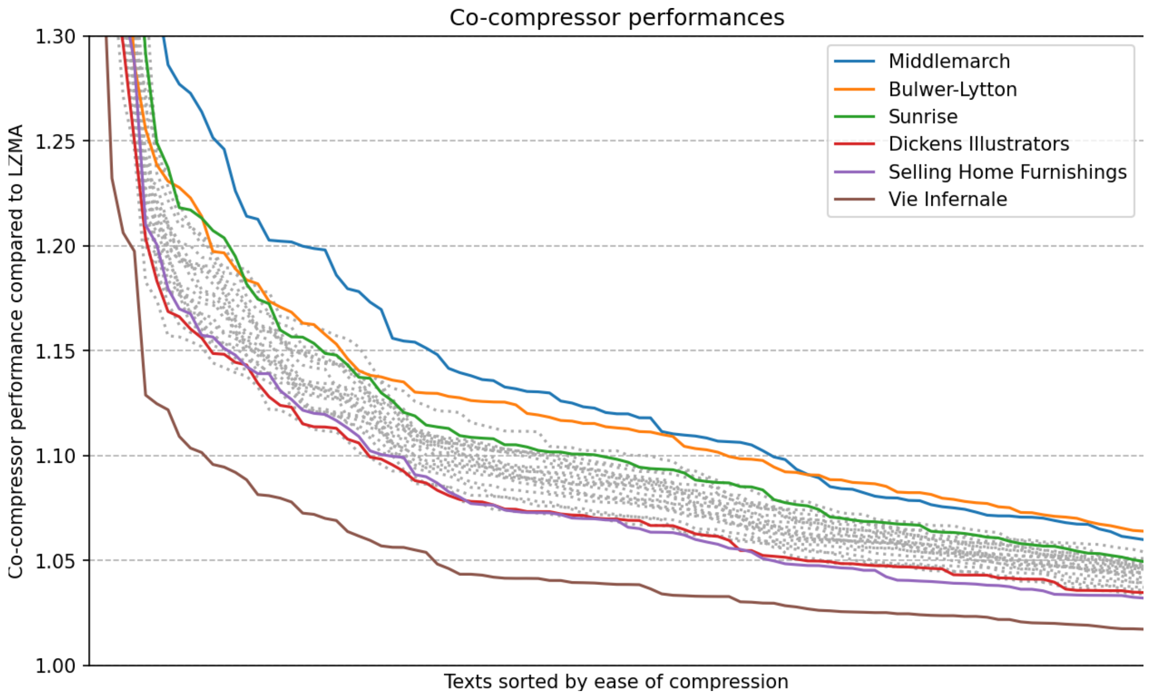


Figure 3.10: The performance of instances of the `CoCompressor` class trained on 30 texts selected for their high RCR out of a set of 96200 candidate texts. Each line represents a `CoCompressor`. The value on the vertical axis is the ratio of the size of the encoding produced by naive LZMA to that produced by the `CoCompressor`, while the horizontal axis goes through possible inputs for comrpession.

3.6 Natural language compilation

3.6.1 Montague grammar

In 1970, logician Richard Montague posited that there is “no important theoretical difference between natural languages and the artificial languages of logicians” and that it is “possible to comprehend the syntax and semantics of both kinds of language within a single natural and mathematically precise theory.” (Montague et al., 1970) and (Montague, 1970)

Montague gives a grammar which can be used to parse English, and a specification for a parser based on this grammar is given by Friedman & Warren (Friedman & Warren, 1978).

Scrap this section or move to chapter 4

4. Discussion and future work

Tie it all up

Bibliography

Friedman, J. & Warren, D. S. (1978) A parsing method for Montague grammars. *Linguistics and Philosophy*. 2 (3), 347–372.

Henderson, L. (2018) The problem of induction.

Hutter, M. (2006) 500’000€ Prize for Compressing Human Knowledge by Marcus Hutter. Available from: `http://prize.hutter1.net/`

— (2000) A theory of universal artificial intelligence based on algorithmic complexity. *arXiv preprint cs/0004001*.

Jiang, Z., Yang, M., Tsirlin, M., Tang, R., Dai, Y. & Lin, J. (2023) “Low-Resource” Text Classification: A Parameter-Free Classification Method with Compressors, 6810–6828.

Legg, S., Hutter, M., et al. (2007) A collection of definitions of intelligence. *Frontiers in Artificial Intelligence and applications*. 157, 17.

Li, M., Chen, X., Li, X., Ma, B. & Vitányi, P. M. (2004) The similarity metric. *IEEE transactions on Information Theory*. 50 (12), 3250–3264.

Montague, R. (1970) English as a formal language.

Montague, R. et al. (1970) Universal grammar. *1974*, 222–246.

Shannon, C. E. (1951) Prediction and entropy of printed English. *Bell system technical journal*. 30 (1), 50–64.

Shannon, C. E. (1948) A mathematical theory of communication. *The Bell system technical journal*. 27 (3), 379–423.