

A modular architecture for Unicode text compression

Adam Gleave
St John's College



UNIVERSITY OF
CAMBRIDGE

*A dissertation submitted to the University of Cambridge
in partial fulfilment of the requirements for the degree of
Master of Philosophy in Advanced Computer Science*

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
UNITED KINGDOM

Email: arg58@cam.ac.uk

10th June 2016

Declaration

I Adam Gleave of St John's College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 11503

Signed:

Date:

This dissertation is copyright ©2016 Adam Gleave.
All trademarks used in this dissertation are hereby acknowledged.

Acknowledgements

I would like to thank Dr. Christian Steinruecken for his invaluable advice and encouragement throughout the project. I am also grateful to Prof. Zoubin Ghahramani for his guidance and suggestions for extensions to this work. I would further like to thank Olivia Wiles and Shashwat Silas for their comments on drafts of this dissertation. I would also like to express my gratitude to Maria Lomelí García for a fruitful discussion on the relationship between several stochastic processes considered in this dissertation.

Abstract

Conventional compressors operate on single bytes. This works well on ASCII text, where each character is one byte. However, it fares poorly on UTF-8 texts, where characters can span multiple bytes. In this thesis, I modify compressors to operate on Unicode characters.

Previously developed Unicode compressors fail on encountering binary data. This is problematic as UTF-8 texts are often embedded in a binary format. The first contribution of this thesis is a reversible transformation mapping the input to a sequence of *tokens*, representing either Unicode characters (where the input is UTF-8) or an error condition that precisely specifies the invalid input (in all other cases).

There are millions of tokens, making it challenging to learn their distribution by a simple frequency counting approach. Fortunately, Unicode characters are allocated in *blocks* of the same script. Most texts will use only one or two scripts, reducing the vast code space to a more manageable size. The second contribution of this thesis is a demonstration that Pólya trees can take advantage of the structured nature of Unicode to rapidly learn which blocks are used.

Finally, I apply the above techniques to the PPM and LZW compressors, producing Unicode-optimised versions of these algorithms. Using tokens improves the compression effectiveness of LZW on my test data by 20% for Unicode files, at the cost of 2% for ASCII texts. My implementation of PPM achieves 13% greater compression on Unicode files than a state-of-the-art compressor, PPMII. It fares 7% worse than PPMII on ASCII texts, but 5% of this is due to my implementation using a less sophisticated model, with only 2% attributable to token overhead.

Both PPM and LZW are widely used, making this modification of immediate practical benefit. Moreover, it seems likely this technique could be successfully applied to many other compressors. Further work is needed to verify this, but it is encouraging that PPM and LZW enjoy similar benefits, despite having substantially different designs.

Contents

1	Introduction	1
2	Compressing UTF-8 text	5
2.1	Unicode and UTF-8	5
2.2	Existing Unicode compression methods	6
2.3	Transforming UTF-8 data	9
2.3.1	Tokens as integers	11
3	Models over tokens	13
3.1	Histogram learning with smoothing	14
3.2	Pólya trees	15
3.3	Applications	18
4	PPM	21
4.1	The original algorithm	22
4.2	Update exclusion	24
4.3	Methods for histogram learning	24
4.3.1	Choosing a method	24
4.3.2	The PPMG method	25
5	LZW	27
5.1	Existing implementations	27
5.2	My extension	28
6	Evaluation	31
6.1	Test data	32
6.1.1	Goals	32
6.1.2	Files selected	32
6.2	Single-symbol models	36
6.3	LZW	39
6.4	PPM	43
6.4.1	Selecting parameters	43
6.4.2	Optimal parameters	45
6.4.3	Robustness of parameters	48
6.4.4	Effectiveness of the token compressor	51

7	Conclusions	59
7.1	Summary	59
7.2	Future work	60
7.2.1	Optimised implementation	60
7.2.2	Applications to other algorithms	60
7.2.3	Compression of other data formats	60
A	Test results	63
A.1	Compression effectiveness	63
A.2	Machine specification	63
	Bibliography	67

List of Figures

2.1	Token types	10
3.1	A finite Pólya tree	16
6.1	Compression effectiveness of PPM against depth	46
6.2	Compression effectiveness of PPM against α and β	50
6.3	Compressor resource consumption	56

List of Tables

2.1	UTF-8 byte sequences	6
6.1	Test data	33
6.2	Test data, by file format	34
6.3	Effectiveness of single-symbol compressors	37
6.4	Effectiveness of LZW-family compressors	40
6.5	Summary of LZW compression effectiveness	41
6.6	Training data for PPM	44
6.7	Parameters used for PPM	45
6.8	Training data for PPM, by character length	46
6.9	Optimal parameters for PPM over training and test data	49
6.10	PPM effectiveness depending on parameter choice	49
6.11	PPM compression effectiveness against depth	49
6.12	Effectiveness of PPM-family compressors	52
6.13	Summary of PPM compression effectiveness	54
A.1	Effectiveness of all compressors (part 1)	64
A.2	Effectiveness of all compressors (part 2)	65
A.3	Compressors tested in this dissertation	66

Chapter 1

Introduction

Compression effectiveness is an objective measure of an algorithm's 'intelligence', in the sense of its predictive accuracy. Human test subjects are able to compress English text at a rate of around 1.25 bits per character (Cover and King 1978). By contrast, the best compression algorithms achieve a rate of 1.81 bits per character.¹ Closing this gap would represent a major achievement for artificial intelligence.

As well as being of theoretical interest, text compression is of considerable practical importance. You've most likely used a compressor many times today, as the majority of websites use HTTP/1.1 compression (W3Techs 2016b).² Compressing web pages reduces bandwidth consumption and, consequently, page load time. Text compression is also used in data archival to reduce storage costs.

The Unicode encoding scheme UTF-8 has become the dominant character set for textual data, used by 87.0% of websites (W3Techs 2016a). Whereas legacy encodings such as ASCII represent each character by a single byte, UTF-8 maps characters to sequences of between one to four bytes. Yet text

¹Based on results for the PAQ compressor over English-language, ASCII-encoded texts, as given in table 6.13.

²As of 24th May, 2016. Based on a survey of the top ten million most popular websites by Alexa rankings, from <http://www.alexa.com>.

compression has not caught up with this shift, with most compressors still operating on individual bytes.

In this dissertation, I show that existing compressors can be modified to operate over Unicode characters rather than bytes. I find this modification substantially improves compression effectiveness on Unicode text files, while imposing only a small overhead on other types of data.

My first contribution, in chapter 2, is a reversible transformation between sequences of bytes and *tokens*. UTF-8 streams are mapped to a sequence of tokens representing Unicode characters. Decoding will fail on segments of the data that are not UTF-8 encoded. Such regions are mapped to a sequence of error tokens that precisely specify the original input.

This transformation introduces a new problem. A compressor now needs to learn the input distribution over millions of tokens, rather than just 256 bytes. My second contribution, in chapter 3, is a token model that rapidly learns the input distribution by exploiting the hierarchical nature of Unicode.

My final contribution, in chapters 4 and 5, is to show how two compressors, PPM and LZW, can be modified to operate over any countable alphabet and base distribution. PPM is a context-sensitive text compressor and LZW is a dictionary coder. The two algorithms are of substantially different design, demonstrating the generality of my approach.

Combining these contributions yields variants of PPM and LZW that are optimised for UTF-8 text. In chapter 6 I find compression effectiveness on my UTF-8 test data improves, relative to the original algorithms, by an average of 6.1% for PPM and 20.4% for LZW. This improvement comes at a small cost of around 2% on ASCII files.³

³Calculated from the ratio of mean compression effectiveness, given in tables 6.5 and 6.13. For LZW, I compared LPT and LUB. For PPM, I contrasted PPT with PUB.

I also compare PPT, my variant of PPM, to state-of-the-art compressors. On UTF-8 text I find PPT outperforms compressors with similar resource requirements, and is competitive with those using orders of magnitude more CPU time and memory. Finally, in chapter 7 I conclude and discuss the outlook for future work.

Chapter 2

Compressing UTF-8 text

The goal of this dissertation is to develop effective compressors of UTF-8 text. I start in section 2.1 by giving some background on Unicode and UTF-8. Next, in section 2.2 I survey the existing work on Unicode compression. Finally, in section 2.3 I describe a reversible transformation for UTF-8 which makes UTF-8 text easier to compress.

2.1 Unicode and UTF-8

The Unicode Standard defines how to encode multilingual text (The Unicode Consortium 2015). The latest version at the time of writing, Unicode 8.0.0, contains 120 672 characters from all the world’s major writing systems, as well as many archaic and historic scripts. The standard specifies a unique number for each character, its *code point*.

Unicode defines three *encoding schemes* (UTF-8, UTF-16 and UTF-32) that map code points to a sequence of one or more bytes. For interchange of Unicode text, UTF-8 is the *de facto* standard, used by 87.0% of websites (W3Techs 2016a). Fewer than 0.1% of websites use UTF-16 or UTF-32, although these encodings are commonly used as an internal representation of characters in programming language APIs.

Code point	Byte 1	Byte 2	Byte 3	Byte 4
0xxxxxxx	0xxxxxxx			
yyy yyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Table 2.1: The mapping between code points and byte sequences for UTF-8. From *The Unicode Consortium* (2015, table 3.6).

UTF-8 is a variable-length encoding scheme, mapping code points to sequences of one to four bytes, as specified in table 2.1 (Yergeau 2003). Observe that the first row maps code points between 0_{16} and $7F_{16}$ to the ASCII character of the same value. This property is sometimes called “ASCII transparency”, and is a key reason for UTF-8’s popularity.

Furthermore, UTF-8 is a self-synchronising code, as the range of valid values for the leading byte in a sequence is disjoint from those of the trailing bytes. Additionally, for many alphabets it is a reasonably efficient encoding, since it maps lower code points (which tend to be more frequently used) to shorter byte sequences.

Any byte sequence that does not match one of the rows in table 2.1 is ill-formed. The UTF-8 byte sequences that would map to the surrogate code points $D800_{16}$ to $DFFF_{16}$ are also taken to be ill-formed.¹ Finally, any byte sequences corresponding to code points above $10FFFF_{16}$, the upper bound of the Unicode code space, are also considered ill-formed by the UTF-8 specification.

2.2 Existing Unicode compression methods

To accommodate all the world’s scripts, Unicode has a substantially larger code space than legacy character encodings, which were specific

¹The surrogate code points have a special meaning in UTF-16, but are not valid Unicode scalar values, so are forbidden in UTF-8 and UTF-32.

to individual alphabets. An unfortunate side-effect is that texts tend to be somewhat larger when encoded in UTF-8 rather than legacy encodings. This inflation motivated the development of two Unicode-specific compression methods, the Standard Compression Scheme for Unicode (SCSU) and Binary Ordered Compression for Unicode (BOCU-1).

Both algorithms exploit the fact that the Unicode code space is divided into *blocks* of characters belonging to the same script. Most texts use characters from only a handful of blocks. So although the overall Unicode code space is vast, adjacent characters in a text will tend to have nearby code points. SCSU maintains a *window* onto a region of the code space, encoding characters by an index into this window. BOCU-1 encodes each character by its *difference* from the previous character.

SCSU is a Unicode technical standard (Wolf et al. 2005). Its state includes a window of 128 consecutive Unicode characters. If a character lies within the window, it can be encoded in a single byte, representing an index into the window. This works well for small alphabetic scripts such as Greek, where the window can cover all or most of the block. However, this windowing method fares poorly on larger logographic scripts such as the Chinese *Hanzi*. For such scripts, the SCSU encoder can switch *modes* into an (uncompressed) UTF-16 stream.

SCSU compares unfavourably with general purpose compression algorithms, with my tests in section 6.2 showing it to be between two to four times less effective than bzip2. Furthermore, it can only operate on Unicode data: it will fail on files that are a mixture of binary and text, for example. However, it does have very little fixed overhead, making it efficient for compressing short strings.

Scherer and Davis (2006) developed an alternative, BOCU-1, which encodes each character by its *difference* from the previous character. Tests by Scherer and Davis show it achieves similar compression ratios to SCSU. It has the advantage of being directly usable in emails, as it is compatible with the text Multipurpose Internet Mail Extensions (MIME)

media type. Furthermore, the lexical order of the output bytes is the same as the code point order of the original text. This makes it suitable for applications, such as databases, where strings must be stored in a sorted order.

Despite these benefits, BOCU-1 has proven even less popular than SCSU. This is perhaps due to the lack of a formal specification, with the normative reference being a C implementation of the algorithm.

The Unicode Consortium advises that anything other than short strings should be compressed using a general-purpose compression algorithm such as bzip2 (The Unicode Consortium 2010). The authors of SCSU appear to have anticipated this, as one of their design goals is that the algorithm's output should be suitable for "input to traditional general purpose data compression schemes" (Wolf et al. 2005, section 1).

However, Atkin and Stansifer (2003, table 6) find that the use of SCSU or BOCU-1 as a preprocessing step *decreases* compression effectiveness compared to directly compressing UTF-16 with bzip2.² Their conclusion is that "as far as size is concerned, algorithms designed specifically for Unicode may not be necessary".

In this dissertation, I show there is an alternative approach. It is possible to extend many general-purpose compression algorithms to operate directly over Unicode characters. This avoids the expense of creating new compression techniques, which SCSU and BOCU-1 show is difficult to do successfully. At the same time, it enables substantial gains in compression effectiveness compared to the original algorithm.

This method was previously considered by Fenwick and Brierly (1998). They modified LZ77 – a dictionary coder – to operate in a 16-bit mode over symbols in UCS-2 (a precursor to UTF-16), reporting a 2% improvement in compression effectiveness. LZ77 is known to perform


²Although seemingly paradoxical, this result is entirely expected. SCSU can map different characters to the same byte sequence, if the window position has changed. This reduces file size, but will confuse other compressors if used as a preprocessor.

poorly on text relative to algorithms such as PPM, so a 2% gain is insufficient to make it competitive. Furthermore, UCS-2 was never widely adopted, limiting the applicability of this technique.

2.3 Transforming UTF-8 data

Before compression, I decode UTF-8 text into a sequence of Unicode code points. The compressor can then operate directly over these code points. However, the system must be able to compress arbitrary input sequences, whether or not they are valid UTF-8. For example, it might be necessary to compress UTF-8 texts embedded inside a binary format such as a tar archive. Note that SCSU and BOCU-1 do not handle such cases, severely limiting their applicability.

To address this problem, I developed an invertible UTF-8 decoder and encoder that maps between sequences of bytes and *tokens*. A token represents either a Unicode code point, or an error condition that precisely specifies the invalid input. Figure 2.1 shows the different types of tokens. Valid streams of UTF-8 text will be transformed to a sequence of `UnicodeCharacter` tokens terminated by an end-of-file marker `EOF`.

Conventional UTF-8 decoders handle invalid byte sequences by substituting a special *replacement character* U+FFFD, usually typeset as . This practice destroys the original data, and therefore is not an option for an invertible decoder. Instead, my representation in figure 2.1 encodes error conditions so that the original invalid input sequence can be reconstructed exactly.

A subclass of `IllegalCodePoint` is used for malformed sequences that *can* be decoded to a code point, but are forbidden by the UTF-8 standard. UTF-8 is able to encode integers up to $1FFFFFF_{16}$, but the upper bound of the Unicode code space is $10FFFFFF_{16}$. Any values that lie above this

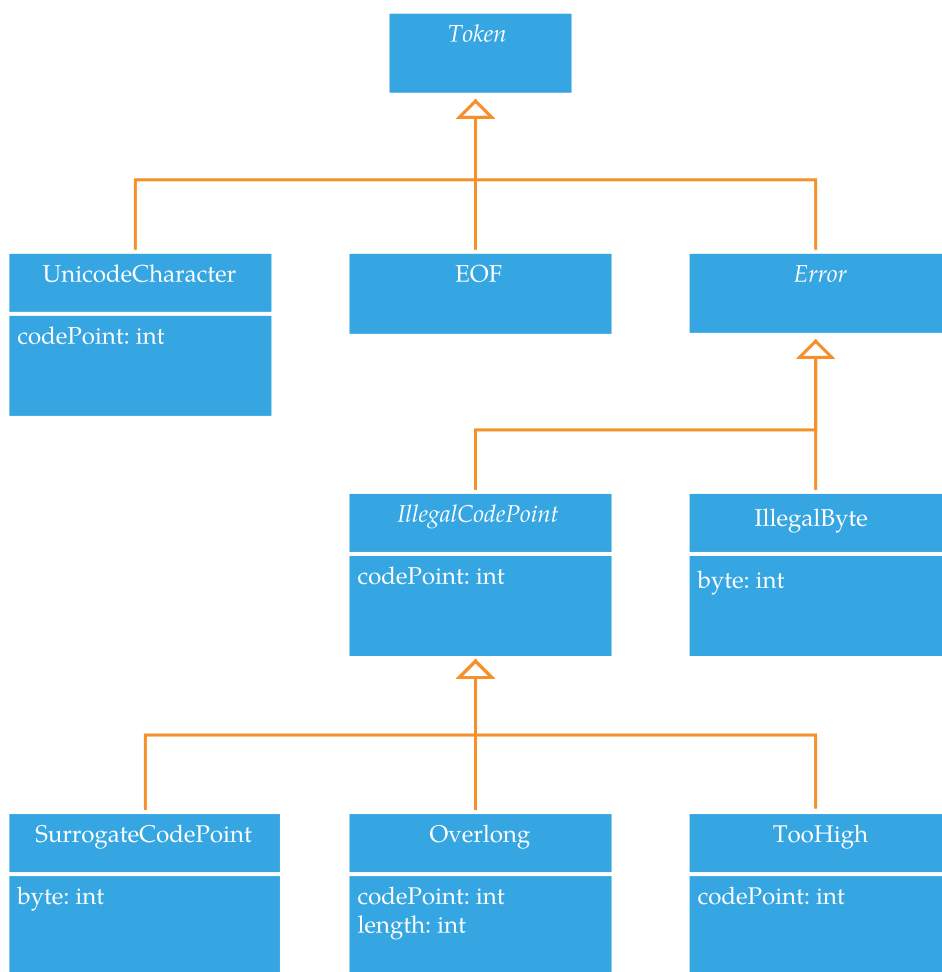


Figure 2.1: UML class diagram of the tokens used to represent input byte streams. Abstract classes are indicated by a name in italics. The concrete classes are UnicodeCharacter, EOF, SurrogateCodePoint, Overlong, TooHigh and IllegalByte.

bound are represented by `TooHigh`. There are also the *surrogate* code points, from $D800_{16}$ to $DFFF_{16}$ (inclusive), which are only meaningful in UTF-16 and are explicitly disallowed in UTF-8. These are represented by `SurrogateCodePoint`.

Finally, a character can be encoded using more bytes than is necessary, by padding its code point with leading 0s. For example, the character 'A' with code point 01000001_2 could be encoded by a single byte of the same value, but could also be represented by $11000001_2\ 10000001_2$. This class of error is indicated by `Overlong`, with `length` denoting the number of bytes in the malformed sequence.³

All other malformed inputs are decoded to a sequence of `IllegalByte` tokens. A minimal representation could dispense with `IllegalCodePoint` and just use `IllegalByte`. However, including `IllegalCodePoint` makes it possible to compress malformed UTF-8 data with similar effectiveness to valid UTF-8 streams.

The mapping is injective – each token represents at most one byte sequence – and is therefore reversible. My implementation is also surjective, which ensures there are no unused tokens.⁴ This property is achieved by restricting the range of values that attributes can take on; see `token.scala` in the source code for details.

2.3.1 Tokens as integers

The previous description treats tokens as *objects*, with the attributes specified in figure 2.1. It is often computationally more convenient to operate over integers rather than objects. There are finitely many tokens N , so clearly a bijective mapping ϕ between tokens and integers

³It is possible to have a surrogate code point encoded using more bytes than is necessary. This case is represented as `Overlong`, not as `SurrogateCodePoint`.

⁴Unused tokens would increase the size of compressor data structures, and potentially decrease compression effectiveness if they were assigned probability mass by the model.

in the range $[0, N - 1]$ is possible. I chose the following mapping, which is simple to compute and preserves the hierarchical nature of the representation.

Each token type T (i.e. a concrete class from figure 2.1) has an associated bijective mapping ϕ_T from instances of T to integers in the range $[0, N_T - 1]$, where N_T is the number of tokens of type T . For example, $\phi_{\text{UnicodeCharacter}}$ maps each character to its code point.⁵

I define my overall bijective mapping ϕ in terms of these type-specific mappings by:

$$\phi(x) = \sum_{S \in \text{LEFT}(T)} N_S + \phi_T(x), \quad (2.1)$$

where $\text{LEFT}(T)$ is the set of all token types to the left of type T in an in-order walk over the tree in figure 2.1. Effectively, this mapping partitions the region $[0, N - 1]$ into contiguous subregions of length N_T for each token type T , in the order (from left-to-right) they occur in the tree.

This mapping has the appealing property of allocating semantically related tokens to nearby integers. Since the tree in figure 2.1 is an inheritance hierarchy, similar types are closer together in the in-order walk and so are allocated nearby regions. Furthermore, for tokens x and y of the same type T , their difference in the overall mapping $\phi(x) - \phi(y)$ is equal to their difference in the type-specific mapping $\phi_T(x) - \phi_T(y)$. Adjacent Unicode characters will thus be mapped to consecutive integers by ϕ .

⁵The mapping is similarly trivial for most other types. See `token.scala` in the source code for details.

Chapter 3

Models over tokens

In the previous chapter, I described a reversible transformation from streams of bytes to sequences of *tokens*. Each token can represent either a Unicode code point or a specific error condition. In this chapter, I investigate models over these tokens.

The simplest possible model is a uniform distribution, assigning equal probability to every token. There are $N = 2,164,993$ possible tokens, so this would compress at a rate of $\log_2 N \approx 21.046$ bits per character. For scripts such as the Chinese *Hanzi*, this is an improvement over UTF-8, which uses three bytes – or 24 bits – per character.

However, a uniform distribution is typically a poor choice. Some scripts, such as the Latin alphabet, are used more widely than others, such as Egyptian hieroglyphs. Furthermore, characters in the same script may occur at different frequencies. The letter ‘e’, for example, is far more common than ‘z’ in typical English texts.

A better approach is to assign probabilities to characters based on how frequently they occur in some corpus. But the distribution of an individual text may differ substantially from the corpus average. In particular, texts are typically written in a single script, concentrating the probability mass in the contiguous block of Unicode characters

corresponding to that script.¹

Since no single distribution can be a good fit for all texts, I investigate *adaptive* models that attempt to learn the source distribution. I start by outlining a simple histogram learner in section 3.1. Next, I describe *Pólya trees* in section 3.2, a learning procedure that tends to assign similar probabilities to characters of the same script. I conclude in section 3.3 by discussing the suitability of the two approaches in different applications.

3.1 Histogram learning with smoothing

Predictions of the next character x_{N+1} can be made from the *empirical distribution* of the input x_1, \dots, x_N observed so far:

$$P(x_{N+1} = x | x_1, \dots, x_N) = \frac{n_x}{N}, \quad (3.1)$$

where n_x is the number of occurrences of x in the input sequence. However, for small N the empirical distribution may be a poor predictor. For data compression, the *zero-count problem* is particularly troublesome. A symbol that has yet to be observed is assigned a probability of zero, and so cannot be encoded.

Smoothing techniques are used to address these problems. One widely-used approach, *additive smoothing*, pretends every symbol x in the input alphabet χ has been observed $\alpha/|\chi|$ times, where $\alpha > 0$ is a constant, giving:

$$P(x_{N+1} = x | x_1, \dots, x_N) = \frac{n_x + \alpha/|\chi|}{N + \alpha}. \quad (3.2)$$

This defines a distribution that is a weighted average between the uniform and empirical distribution. As more observations are made, increasing

¹The frequency of characters tends to be similar for texts of the same language, although it varies slightly. In an extreme case, *lipograms* exclude one letter of the alphabet entirely.

weight is placed on the empirical distribution, at a rate inversely proportional to α . Additive smoothing can be generalised to allow any base distribution H , not just the uniform distribution, producing:

$$P(x_{N+1} = x | x_1, \dots, x_N) = \frac{n_x + \alpha H(x)}{N + \alpha}. \quad (3.3)$$

This model, which I shall denote $HL(H, \alpha)$, avoids the zero-count problem entirely. Furthermore, for well-chosen H it can provide good probability estimates even for small N .

Historical note. The construction of eq. (3.3) is called a *Pólya urn*, and is equivalent to a *Chinese restaurant process*. The Pólya urn is in turn a sequential construction of a *Dirichlet process*. The theoretical background of Dirichlet processes is beyond the scope of this dissertation. See Teh (2010) for a thorough and comprehensive treatment or Frigýik et al. (2010) for a more gentle introduction.

3.2 Pólya trees

An alternative approach to histogram learning over an alphabet χ is a finite Pólya tree. This method uses a balanced binary search tree. Each symbol $x \in \chi$ has a corresponding leaf node l_x , and can be uniquely represented by the path from the root to l_x , denoted $\text{PATH}(x)$. Each internal node i has a *bias* $\theta_i \sim \text{Beta}(\alpha_i, \beta_i)$ representing the probability of taking the left branch. Let $\theta = (\theta_1, \dots, \theta_n)$, where n is the number of internal nodes. The probability of a symbol $x \in \chi$ is defined to be:

$$P(x | \theta) = \prod_{i \in \text{PATH}(x)} B(b_i | \theta_i), \quad (3.4)$$

where $B(k|p) = p^{1-k}(1-p)^k$ is the PDF of the Bernoulli distribution, and $b_i \in \{0,1\}$ is the branching decision at internal node i . Since the Beta

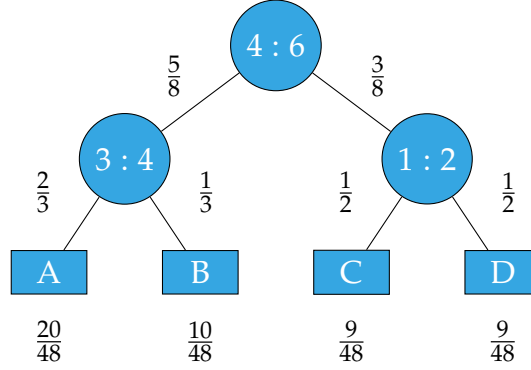


Figure 3.1: An example of a finite Pólya tree over the alphabet $\chi = \{A, B, C, D\}$. The parameters of the Beta prior at each internal node i are set to $\alpha_i = \beta_i = 1$. Internal nodes are labelled with $L_i : M_i$, where L_i is the number of times the left branch was taken and M_i is the total number of branching decisions. Edges are labelled with the predictive probability of the Beta distribution given L_i and M_i . Each leaf node $x \in \chi$ is labelled with its predictive probability $P(x)$ from eq. (3.5).

distribution is a conjugate prior of the Bernoulli distribution, the biases θ_i can be marginalised giving the predictive probability:

$$P(x) = \prod_{i \in \text{PATH}(x)} B\left(b_i \mid \frac{\alpha_i + L_i}{\alpha_i + \beta_i + M_i}\right), \quad (3.5)$$

where M_i is the total number of branching decisions and L_i is the number of times the left branch was taken at node i . An example Pólya tree is shown in figure 3.1.

Let $\text{PRE}(x, y)$ denote the common prefix of $\text{PATH}(x)$ and $\text{PATH}(y)$ (or the empty sequence ϵ , if there is none). We can then factor $P(x)$ as:

$$P(x | \theta) = \left(\prod_{i \in \text{PRE}(x, y)} B(b_i | \theta_i) \right) \left(\prod_{i \in \text{PATH}(x) \setminus \text{PRE}(x, y)} B(b_i | \theta_i) \right), \quad (3.6)$$

and similarly for $P(y)$.

Since the tree is ordered, the closer symbols $x, y \in \chi$ are to each other,

the longer their prefix $\text{PRE}(x, y)$.² Their probabilities $P(x)$ and $P(y)$ will thus share many initial terms, and so $P(x)$ and $P(y)$ will tend to take on similar values. This is apparent in figure 3.1. ‘B’ is assigned a higher probability than ‘C’ or ‘D’, despite all three symbols occurring once each, since ‘B’ is next to the frequently occurring symbol ‘A’.

Unicode code points are allocated in *blocks* of characters from the same script. Most texts use only a small number of scripts, so the histogram of code points in a file will be concentrated over a handful of blocks. This suggests the Pólya tree, with suitably chosen parameters, may be able to more rapidly adapt to typical distributions over Unicode symbols than other histogram learners.

There are over a million tokens, so a naive representation of a Pólya tree would take up a significant amount of memory. Fortunately, there is a simple sparse representation of the tree. It is only necessary to store the counts L_i and M_i for each node. Furthermore, $L_i \leq M_i$ for all nodes i and, if a node j is in the subtree rooted at i , then $M_j \leq M_i$. These properties justify ‘pruning’ any parts of the tree that have zero counts.

Pólya trees were first applied to compression by Steinruecken (2014, section 4.3.1), who used them to learn the distribution of bytes in ASCII text. For very small inputs, he found a benefit from using a Pólya tree rather than the histogram learner described in section 3.1. However, their compression effectiveness was almost indistinguishable when the input was longer than a few thousand bytes (Steinruecken 2014, figure 4.3).

Steinruecken highlighted that Pólya trees might be well-suited to learning distributions over Unicode characters, but did not pursue this approach further. To the best of my knowledge, this dissertation contributes the first implementation of a Pólya tree learner over Unicode characters. Another novel contribution is the use of the Pólya tree as a base distribution for more sophisticated models, discussed further in the next section.

²To be precise, the prefix length is non-decreasing on $|x - y|$.

Although not needed for this dissertation, the Pólya tree generalises to a version with infinite depth. This general form is able to express both discrete and continuous distributions, depending on the parameter choice. It even includes the Dirichlet process as a special case, when the parameters α_i, β_i of the Beta distributions are set to decay exponentially with depth (Müller and Rodriguez 2013, section 4.1). See e.g. Mauldin et al. (1992) for further information.

3.3 Applications

A compressor can be built from a probabilistic model M using *arithmetic coding* (Witten et al. 1987). The resulting compressor is optimal (within two bits) on inputs distributed according to M .

A sequence of independent and identically distributed (iid) symbols with known distribution D is particularly simple to compress. For each symbol in the input sequence, compress it using arithmetic coding over D .

In practice, D is unknown, forcing the use of adaptive techniques such as the histogram or Pólya tree learner from the previous sections. Each token x_N is compressed using arithmetic coding on the predictive distribution conditioned on x_1, \dots, x_{N-1} . The closer the predictive distribution is to D , the greater the compression effectiveness.

Unfortunately, inputs are rarely *iid*. Text certainly isn't independent: 'u' is much more likely to occur after 'q' than after 'z', for example.

To accommodate dependency, it is common to use context-sensitive models, with distributions *conditional* on the preceding symbols. These models can learn, for example, that although 'u' rarely occurs in the input data, it often occurs in contexts ending with 'q'.

Context-sensitive models are often parametrised by a *base* distribution over tokens. Initially, the predictive distribution is equal to the base distribution. As more observations are made, the predictive distribution

converges to the empirical distribution. Two techniques are commonly used to combine the base and empirical distributions: *blending* and *escaping*.

In a blending model, the predictive distribution is a weighted average of the base and empirical distribution. The weight placed on the empirical distribution increases with the number of observations.

With escaping, the base distribution is only used for unseen symbols. The alphabet χ is extended to include a special *escape symbol* ESC. Let S denote the set of symbols previously observed. When a symbol $x \in \chi$ is seen for the first time (i.e. $x \notin S$), first ESC is encoded in the context-sensitive model. This is followed by x being encoded using the base distribution H conditioned on x not having been previously observed, that is $H|x \notin S$, a technique called *exclusion coding*.

The histogram learner $HL(H, \alpha)$ from section 3.1 (with base distribution H and concentration parameter α) could be used as a base distribution in more complex models. However, for the context-sensitive models PPM and LZW considered in this dissertation (which both use escaping) there is no benefit from using $HL(H, \alpha)$ as the base distribution rather than H directly. This is because $HL(H, \alpha)$ and H assign equal probabilities to any unseen symbol x when all observed symbols are excluded (i.e. conditioning on $x \notin S$).

By contrast, the Pólya tree learner is well-suited to use as a base distribution. It can make good predictions about the probability of symbols that have never before been seen, by taking into account the frequency with which neighbouring symbols have occurred. For example, suppose the Pólya tree has seen many Chinese *Hanzi*. It will then assign a higher probability to seeing a new *Hanzi* character compared to a character from another script.

Chapter 4

PPM

In the previous chapter I surveyed models that do not use context. These models define a single distribution over the symbol alphabet based on the number of times each symbol has occurred in the input stream. Context-free models have the benefit of simplicity. However, they fail to capture dependencies between adjacent characters in text. The distribution of symbols following the letter ‘q’ is quite different from the distribution following the letter ‘t’, for example.

Context-based models try to resolve this problem by learning separate symbol distributions for different contexts. In an input sequence (x_i) , the length- k context of symbol x_n is the sequence $(x_{n-k}, x_{n-k+1}, \dots, x_{n-1})$. The simplest approach is to use contexts of fixed length N , constructing a histogram for each context from the data.

In this model, what value should the order N be set to? Higher-order models will make better predictions, provided accurate statistics are known in each context. Lower-order models have worse performance in the long run, but quickly learn the distribution over the input data.

Cleary and Witten (1984) bypass this problem in their Prediction by Partial Matching (PPM) algorithm by using variable length contexts. The longest previously encountered context is used to encode each symbol.

If the symbol is unseen in that context, a special *escape symbol* is coded, and the context is shortened by dropping the oldest symbol. This process repeats until a context containing the symbol is found. If the symbol has never been observed in the input sequence, it will not be present even in the empty context, in which case the algorithm escapes to a *base distribution* (often a uniform distribution).

This design means PPM is not tied to any particular symbol alphabet. Rather, the alphabet is implicit in the support of the base distribution. Making PPM operate over tokens is therefore as simple as using a distribution over tokens, such as one of the models from the previous chapter.

In section 4.1 I describe the original PPM algorithm by Cleary and Witten (1984). Many variants of PPM have been developed since then. In sections 4.2 and 4.3 I describe two modifications that are incorporated in the implementation used for this dissertation.

4.1 The original algorithm

PPM is parametrised by a maximum context depth D . To encode a symbol x , PPM starts with the D -length context preceding x . (Unless x is one of the first D symbols, in which case it uses as many preceding symbols as are available.) If x is unseen in this context, an escape symbol (ESC) is coded. (The ESC symbol is a virtual symbol that is treated as part of the alphabet.) The context is then shortened to length $D - 1$, by dropping the symbol furthest away from x , and the process is repeated recursively.

In most cases, a context is eventually reached where x has been seen. In this case, it is coded directly with the histogram learned for this context. If it is the first time symbol x occurs in the input data, x will not be present even in the empty context of length 0. In this case, once the empty context

is reached a further ESC is encoded, and x is then directly coded using the base distribution.

This generic approach does not specify how the histogram in each context is used to assign probabilities to symbols. Many methods have been proposed: I survey the most notable approaches in section 4.3. For simplicity of exposition, I start by describing ‘method A’ from Cleary and Witten (1984), which assigns the following probabilities:

$$P(x) = \frac{n_x}{N+1} \quad \text{and} \quad P(\text{ESC}) = \frac{1}{N+1}, \quad (4.1)$$

where n_x is the number of times symbol x occurs in the current context, and N is the total number of symbols present in that context.

Encoding an escape symbol does not just serve to switch to a shorter context level. It also conveys information: symbol x is *not* any of the symbols \mathcal{S} that were observed in that context.¹ PPM uses *exclusion coding* to take advantage of this. By treating the symbols in \mathcal{S} as if they had zero counts, the probability mass is redistributed to other symbols. Letting $S = \sum_{x \in \mathcal{S}} n_x$, the probability assignments for method A become:

$$P(x) = \begin{cases} \frac{n_x}{N-S+1} & x \notin \mathcal{S} \\ 0 & x \in \mathcal{S} \end{cases} \quad \text{and} \quad P(\text{ESC}) = \frac{1}{N-S+1}. \quad (4.2)$$

Exclusion coding is widely used, but the PPM literature usually gives probabilities in the plain form of eq. (4.1) in preference to the explicit form of eq. (4.2). I will follow this convention and use the plain form in the remainder of the dissertation.

Practical implementations of PPM must be able to efficiently compute the summary statistics n_x and N . A *trie* data structure is particularly well suited to this, see e.g. Salomon (2012, section 5.14.5) for more details.

¹The exception is if no symbols have been observed in the context, that is $N = 0$. But in this case, escaping is predicted with probability 1, so there is no overhead.

4.2 Update exclusion

In the original implementation by Cleary and Witten, after observing a symbol x its count n_x is incremented in the current context and all *shorter* contexts. The resulting counts n_x correspond to the actual number of times x has occurred in that context.

An alternative is to increment the count n_x only in context levels at or above the context in which x is encoded. This modification is termed *update exclusion* and was proposed by Moffat (1990, section III.B), who found it to improve compression effectiveness by 5%. It also has the benefit of being more computationally efficient than the original, since in the common case the procedure terminates early.

Update exclusion makes the probability of a symbol depend on the *number of contexts* in which it occurs, rather than its raw frequency. Further explanation of the probabilistic interpretation of update exclusion is provided by MacKay and Bauman Peto (1995).

4.3 Methods for histogram learning

4.3.1 Choosing a method

The PPM algorithm defines an adaptive and context-sensitive probability distribution over input symbols. This distribution, and the assumptions it makes, is sometimes called the probabilistic model of the algorithm. In the first part of section 4.1 I described a generic PPM algorithm whose probabilistic model depends on a method assigning symbol probabilities based on the histogram of the current context. The latter part of section 4.1 used the example of ‘method A’, often called PPMA.

Cleary and Witten (1984) claim that “in the absence of *a priori* knowledge, there seems to be no theoretical basis for choosing one solution [method]

over another". This view has resulted in a proliferation of methods, most notably PPMA and PPMB by Cleary and Witten (1984), PPMC by Moffat (1990), PPMD by Howard (1993), PPME by Åberg et al. (1997), and PPMP and PPMX by Witten and Bell (1991).

These methods have been designed for PPM compressors operating over bytes. A method appropriate for bytes might not perform well over the much larger space of tokens described in section 2.3. But to enable a direct comparison between byte-based and token-based compressors, it is desirable to use the same family of methods. Accordingly, I selected PPMG, a method taking two continuous parameters which generalises PPMA, PPMD and PPME (Steinruecken 2014, section 6.4.2).

Although I agree with Cleary and Witten that there is no way to design a method without making some *a priori* assumptions, with PPMG the optimal parameters can be determined *a posteriori* from the data. These optimal parameters, for a sufficiently representative training corpus, may be a useful initialisation of the algorithm for other data. It is also possible to choose parameters dynamically, optimising them as the text is compressed, as demonstrated by Steinruecken et al. (2015).

For simplicity, I use static parameters (selected by optimising over training data). I find the optimal parameters vary significantly between bytes and tokens, confirming the benefit of a principled approach to method construction. See sections 6.4.1 and 6.4.2 for more details.

4.3.2 The PPMG method

PPMG takes a discount parameter $\beta \in [0, 1]$ and concentration parameter $\alpha \in [-\beta, \infty)$, assigning probabilities:

$$P(x) = \frac{n_x - \beta}{N + \alpha} \cdot \mathbb{1}[n_x > 0] \quad \text{and} \quad P(\text{ESC}) = \frac{U\beta + \alpha}{N + \alpha}, \quad (4.3)$$

where n_x is the number of observations of x in the current context, N is the total number of observations in the current context and U is the number of unique symbols observed in the current context.

The parameter α can be interpreted as a pseudocount for the escape symbol ESC. However, the number of unique symbols varies between contexts: more letters might occur after 'e' than after 'q', for example. It can also vary among files: a text in a logographic script will use a larger subset of Unicode characters than one in an alphabetic script, for example. There is therefore no universally optimal choice for α .

The parameter β allows the escape probability to *adapt* depending on the input. ESC is assigned a larger probability when the number of unique symbols U observed in this context is large. The more the size of the character set varies among different contexts and files, the larger the optimal value of β .

The parameters have a similar interpretation to those in a Pitman–Yor process. See Steinruecken (2014, section 6.6.3) for more details.

Chapter 5

LZW

In this chapter I describe Lempel-Ziv-Welch (LZW), a popular compression algorithm used, for example, in the GIF and PDF file formats. Invented by Welch (1984), LZW is a variant of the LZ77 and LZ78 algorithms due to Ziv and Lempel (1977, 1978).

PPM, described in the previous chapter, typically achieves greater compression ratios than LZW, especially on text. However, LZW can be more efficient, with a smaller runtime and memory footprint. Furthermore, LZW is an example of a wider class of *dictionary coding* algorithms, making it an interesting case study.

In section 5.1 I outline the original LZW algorithm and commonly implemented variants. Generalising LZW to operate over tokens is more challenging than for PPM. I state the problem and my solution in section 5.2.

5.1 Existing implementations

LZW compresses the input sequence by encoding substrings as their indices in a dynamically constructed dictionary. Specifically, the

dictionary is initialised to contain every possible byte value 00_{16} to FF_{16} . The longest string w in the dictionary matching the head of the input is located, and its index emitted. w is then removed from the input. If the input is empty, the algorithm terminates. Otherwise, the input can be written as $x :: s$, where x is a single byte, s a (possibly empty) sequence of bytes and $::$ denotes string concatenation. Add $w :: x$ to the dictionary, and repeat the process (with the updated dictionary) to compress $x :: s$.

The decompressor reconstructs the dictionary in lockstep with the compressor. However, when $w :: x$ is added to the dictionary in the compressor, only w has been encoded, not x . To accommodate this, the decompressor leaves the last byte blank in newly inserted words. This final byte is filled in using the first byte of the next word to be decoded.

There is a choice of how to encode the dictionary indices. Welch (1984) originally proposed representing each index as a 12-bit unsigned integer. Such fixed-length codes are inflexible, limiting the dictionary to only 4096 entries in the 12-bit case, and wasteful, using more bits than necessary.

A simple alternative is to scale the number of bits b as the dictionary size N grows, such that $2^{b-1} < N \leq 2^b$. This approach is used in the Unix `compress` command (Salomon 2012, section 6.14). However, except when $N = 2^b$, this still wastes up to one bit per index. Consequently, the implementation in this dissertation uses arithmetic coding, which encodes each index in $\log_2 N$ bits (Witten et al. 1987).

5.2 My extension

In this section, I show how LZW can be modified to operate over the tokens described in section 2.3. The naive approach simply initialises the

dictionary to contain every token rather than every byte. However, this inflates the size of the dictionary, increasing the number of bits needed to encode each index. Since most tokens will never be observed in a particular file, this is extremely wasteful.

My implementation resolves this problem by escaping to a *base model* over tokens (such as one of the models from chapter 3) the first time a symbol is seen. The dictionary is initialised to contain a single entry ϵ , the empty string. When the compressor encounters an unseen symbol x , it encodes ϵ , and uses the base model to encode x . As with any substring seen for the first time, x is added to the dictionary, so the escape procedure is used at most once for each symbol. I use exclusion coding, described in section 3.3, to take advantage of this property.

An approach similar to escaping was envisaged by Welch (1984, page 11):

The compression string table could be initialized to have only the null string. In that case, a special code is necessary for the first use of each single-character string. This approach improves compression if relatively few different symbols occur in each message.

However, he does not provide any further details on this method. My tests in section 6.3 find that escaping improves compression effectiveness even when using a byte alphabet. I therefore suspect Welch never implemented escaping, as otherwise he would have likely mentioned it more prominently in his paper. To the best of my knowledge, this dissertation contributes the first implementation and evaluation of LZW with escaping.

Chapter 6

Evaluation

The *effectiveness* of a compression algorithm is the ratio between the size of the compressed output and that of the uncompressed input. It is often expressed in bits (of output) per byte (of input). Algorithms are typically evaluated by measuring their effectiveness on a *corpus* of test data. The use of a corpus allows compression results to be compared objectively. I describe the corpus used in these tests in section 6.1.

Following sections present experimental results. I start by investigating the effectiveness of different token distributions, in section 6.2. I then measure their effectiveness when used as a base distribution for LZW, in section 6.3, and for PPM, in section 6.4.

Appendix A includes several tables that may be useful while reading this chapter. Table A.3 on page 66 briefly describes all the compressors tested in this dissertation. A side-by-side comparison of the compression effectiveness of these algorithms is given on pages 64 to 65.

6.1 Test data

6.1.1 Goals

The files in a corpus should be representative of the inputs that are likely to be encountered when deployed. Given the focus of this dissertation, UTF-8 texts should form the majority of the corpus. However, it is important to capture the diversity of UTF-8 texts. One variable is the *language* used in the text. Another important factor is the *file format* the text is embedded in: e.g. plain text, HTML, or part of a tar archive.

In practice, it is inevitable that a specialised compression algorithm will sometimes be required to compress other types of files. A text compressor cannot be expected to perform as well on binary inputs, such as an image or executable file, as a general-purpose compressor. However, it is reasonable to expect that the text compressor does not fail, and that it still achieves acceptable compression effectiveness. The corpus should therefore include a small number of non-text files.

Furthermore, the corpus should contain files of a variety of sizes to reflect real-world usage patterns. Tests on smaller files reveal how quickly the model is able to learn, and the accuracy of its initial prior.

Finally, all files in the corpus must be freely redistributable. This ensures others are able to reproduce the results in the dissertation. I hope that the corpus I assemble can be used as the basis of future experiments. Although there exist standard corpora for ASCII-encoded English language text (discussed in the next section), there is currently no equivalent for Unicode texts.

6.1.2 Files selected

The Canterbury corpus by Arnold and Bell (1997) has become a *de facto* standard for evaluating compression methods. It consists of 11 files, listed

File	Size (bytes)	Description
alice29.txt	152 089	<i>Alice in Wonderland</i> , English
asyoulik.txt	125 179	<i>As you like it</i> , Early Modern English
cp.html	24 603	HTML source, English
fields.c	11 150	C source
grammar.lsp	3 721	LISP source
kennedy.xls	1 029 744	Excel spreadsheet
lcet10.txt	426 754	Technical writing, English
plrabn12.txt	481 861	<i>Paradise lost</i> , English poetry
ptt5	513 216	CCITT test set (fax)
sum	38 240	SPARC Executable
xargs.1	4 227	GNU manual page, English

(a) *The Canterbury corpus* (Arnold and Bell 1997).

File	Size (bytes)	Description
<i>Single language</i>		
beowulf.txt	160 140	<i>Beowulf</i> , Old English poetry
dostoevsky.txt	1 934 112	<i>Crime and punishment</i> , Russian novel
genji.txt	1 454 767	<i>The tale of Genji</i> , Japanese novel
genji02.txt	67 586	2nd chapter of <i>The tale of Genji</i>
kokoro.txt	484 562	<i>Kokoro</i> , Japanese novel
obiecana.txt	1 230 426	<i>Ziemia obiecana</i> , Polish novel
<i>Mixed language</i>		
dictionary.txt	763 082	Chinese to English dictionary
license.html	37 319	Ukrainian legal text
<i>Binary formats</i>		
koziem.tar	1 720 320	kokoro.txt and obiecana.txt
genji.tar	1 484 800	Chapters from <i>The tale of Genji</i>

(b) Foreign language texts (UTF-8 encoded). Available for download from <https://github.com/AdamGleave/UnicodeCorpus>.

Table 6.1: The test data used in my experiments.

Group	Files
ASCII	alice29.txt, asyoulik.txt, cp.html, fields.c, grammar.lsp, lcet10.txt, plrabn12.txt, xargs.1
Unicode	dostoevsky.txt, genji.txt, genji02.txt, kokoro.txt
Mixed	beowulf.txt, obiecana.txt, dictionary.txt, license.html
Binary	kennedy.xls, ptt5, sum, genji.tar, kokoziem.tar

Table 6.2: The test files grouped by whether they are ASCII, Unicode with multi-byte UTF-8 codewords, a mixture of the two, or binary.

in table 6.1a, chosen to be representative of likely compression inputs. The majority of the files are textual: four are plain text (*.txt), two are written in a markup language (cp.html and xargs.1) and two are source code (fields.c and grammar.lsp). There are also three binary files, consisting of a spreadsheet kennedy.xls, fax data ptt5 and executable file sum.

Although the corpus has proven valuable, there is one aspect in which it is lacking: all the textual data is in English. When the Canterbury corpus was created in 1997, this may have been a reasonable decision, with Pimienta et al. (2009, table 8) estimating that 75% of online content in 1998 was in English. By 2005, their calculations place foreign language content as making up 55% of the web, overtaking English. Their estimations are unreliable after 2005 due to an increasing bias towards English in search indices, however the authors postulate the percentage of English language content dropped to below 40% in 2007.

More recently, a survey conducted by W3Techs (2016c) in May 2016 found that 53.6% of websites were in English.¹ This would seem to suggest a reversal in the trend, however this figure is likely a significant underestimate of the total foreign language content, due to a bias in the demographics of Alexa toolbar users.

Although it remains difficult to determine precisely what proportion of

¹As of 11th May, 2016. Based on a survey of the top ten million most popular websites by Alexa rankings, from <http://www.alexa.com>.

content on the web is in a foreign language, it is clearly substantial, and likely greater than 50%. Given this, any corpus claiming to be representative of today's compression inputs should aim to have around half of the text files be in a foreign language.

To this end, I have assembled a collection of non-English UTF-8 texts, enumerated in table 6.1b. The texts span a range of languages: *Beowulf* and *Ziemia obiecana* are Old English and Polish respectively, written in an Extended Latin alphabet, giving a mixture of 1-byte and 2-byte UTF-8 codewords. *Crime and punishment* is in Russian (Cyrillic alphabet), and consists of 2-byte UTF-8 codewords. *The tale of Genji* and *Kokoro* are both Japanese, with 3-byte UTF-8 codewords. However, they are separated by nine centuries, and are written in different alphabets: *The tale of Genji* uses *hiragana* almost exclusively, whereas *Kokoro* is primarily written in *kanji*.

Texts may contain multiple languages, for example the use of foreign loanwords. The file `dictionary.txt` consists of 10,000 randomly selected entries from CC-CEDICT, a Chinese-English dictionary.² `license.html` is the Ukrainian version of the Creative Commons license.³ Although the text itself is solely Ukrainian, it is mixed with HTML tags, which for the purposes of compression is similar to having multiple languages in the same file.

Finally, text is often embedded in a binary format. I include two tar archives (containing other files from the corpus) to test this case. Well-designed compressors should have similar levels of effectiveness whether compressing the original file or the tar archive.

The test data is summarised in table 6.2. Files are categorised as binaries or texts, with texts grouped by their character encoding. The table shows the corpus is well-balanced, with a similar number of ASCII texts to non-ASCII (Unicode and mixed) texts.

²<http://cc-cedict.org/>.

³<https://creativecommons.org/licenses/by/4.0/legalcode.uk>

6.2 Single-symbol models

The compression effectiveness of the single-symbol models described in chapter 3 is reported in table 6.3. The simplest model is the uniform byte distribution UB, which unsurprisingly encodes at around 8 bits/byte.⁴ Of more interest is the uniform token distribution UT, which places equal probability on all $N = 2,164,993$ possible tokens. As expected, UT is less effective than UB on binary files, encoding at around UT's entropy of $\log_2 N \approx 21.046$ (3 dp).

Intriguingly, UT is also less effective than UB on many text files. This result is because the files are encoded in UTF-8, which maps Unicode characters with lower code points to shorter codewords. Consequently, UB places more probability mass on characters that have short codewords, which tend to occur more often in texts.

Consistent with this hypothesis, UT is more effective the longer the codewords in the file. Indeed, it achieves a better compression ratio than UB in the case of Japanese text (such as `genji.txt` and `kokoro.txt`) where most characters have 3-byte codewords.

Both UB and UT are static distributions, remaining constant throughout the compression process. I also include two adaptive methods: the histogram learner outlined in section 3.1, and the Pólya tree learner described in section 3.2.

Recall that the histogram learner keeps count of the number of times each symbol has been observed. By contrast, a Pólya tree learner maintains a balanced binary search tree. Each symbol is represented by a leaf node, with internal nodes storing branching probabilities. The probability of

⁴The uniform byte distribution assigns equal probability mass to the 257 possible outcomes: one of the 256 bytes, or an end of file symbol EOF. The compression effectiveness therefore converges asymptotically to $\log_2 257 = 8.006$ bits/byte (3 dp). However, on very small files such as `fields.c` and `xargs.1`, the effectiveness is slightly worse than this.

File	Size (KiB)	Static		Adaptive			Reference		
		UB	UT	HB	HT	PT	SCSU	gzip	bzip2
alice29.txt	149	8.006	21.046	4.573	4.580	4.653	8.000	2.863	2.272
asyoulik.txt	122	8.006	21.046	4.814	4.822	4.832	8.000	3.128	2.529
cp.html	24	8.006	21.047	5.265	5.311	5.256	fail	2.598	2.479
fields.c	11	8.006	21.048	5.086	5.192	5.062	8.000	2.249	2.180
grammar.lsp	4	8.009	21.052	4.812	5.083	4.762	8.000	2.653	2.758
kennedy.xls	1006	8.006	21.045	3.576	3.580	3.614	fail	1.606	1.012
lcet10.txt	417	8.006	21.046	4.671	4.674	4.672	8.000	2.716	2.019
plrabn12.txt	471	8.006	21.046	4.533	4.536	4.538	8.000	3.241	2.417
ptt5	501	8.006	21.021	1.213	1.216	1.317	fail	0.880	0.776
sum	37	8.006	20.973	5.391	5.487	5.381	fail	2.703	2.701
xargs.1	4	8.009	21.051	5.057	5.288	5.015	8.000	3.308	3.335
beowulf.txt	156	8.006	19.319	4.623	4.080	4.073	7.547	2.974	2.221
dostoevsky.txt	1889	8.006	11.909	4.016	2.650	2.821	4.527	2.192	1.405
genji.txt	1421	8.006	7.118	4.277	2.302	2.337	3.946	2.430	1.545
genji02.txt	66	8.006	7.091	4.235	2.456	2.356	3.896	2.629	2.000
kokoro.txt	473	8.006	7.061	4.586	2.508	2.519	4.350	2.515	1.702
obiecana.txt	1202	8.006	19.607	4.891	4.441	4.577	7.751	3.150	2.276
dictionary.txt	745	8.006	17.821	5.845	5.146	5.038	7.743	4.071	2.916
license.html	36	8.006	12.960	4.807	3.532	3.538	4.940	1.899	1.502
genji.tar	1450	8.006	7.400	4.350	2.384	2.418	4.028	2.404	1.520
koziem.tar	1680	8.006	16.078	5.604	4.319	4.420	6.794	2.970	2.119

worse ←  → better

Table 6.3: Effectiveness of single-symbol compressors, over the test data in table 6.1. UB and UT are uniform distributions over bytes and tokens respectively. HB and HT are histogram learners (see section 3.1) with α set to 1 and a UB and UT base distribution respectively. PT is a Pólya tree learner (see section 3.2) over tokens. SCSU is the Standard Compression Scheme for Unicode described in section 2.2. gzip and bzip2 are included for comparison. All figures are given to 3 decimal places. Each cell is shaded to indicate how good the compression rate is relative to other compressors in the table. The best compressor in each row is in **bold**.

a symbol x is the product of the branching probabilities along the path from the root to the leaf node representing x .

The asymptotic behaviour of the Pólya tree and histogram learner are the same. However, the hierarchical assumption made by the Pólya tree allows it to learn the input distribution more quickly when the probabilities of neighbouring symbols are similar.

Results are given for two histogram learners, HB and HT, of the form given in section 3.1. The parameter α is set to 1, and their base distribution is uniform over bytes (for HB) and tokens (for HT). On non-Unicode files, HB slightly outperforms HT, with the difference narrowing the larger the file. But HT has a significant edge on Unicode files. This is most pronounced on files with many multi-byte codewords. For example, HT is almost twice as effective as HB on `genji.txt` (containing mostly 3-byte codewords).

The Pólya tree learner over tokens, PT, uses $\alpha_i = \beta_i = 1/2$ at all nodes. It is more effective than HT on small files such as `fields.c`, `xargs.1` and `genji02.txt`. But on larger files PT is typically less effective than HT. I suspect this is because the hierarchical assumption of PT holds at a macro scale, but not a micro scale.

The Unicode code space is divided into *blocks*: contiguous ranges of related characters, such as the Cyrillic alphabet or mathematical operators. Only a small number of blocks are used in typical Unicode files, so at a macro scale the hierarchical assumption holds.⁵ This enables the Pólya tree to quickly learn which blocks are used in a text, so PT achieves greater compression effectiveness on small files than HT.

But the hierarchical assumption breaks down at a micro scale. Neighbouring characters in a block do not necessarily have similar

⁵A simple monolingual text might use only a single block. More will be used in multilingual texts, or in languages which make use of multiple alphabets, such as Japanese. Texts that make use of Unicode symbols, such as mathematical operators, will also require more blocks. However, it is difficult to imagine a non-pathological example that would use more than a small fraction of the defined blocks.

probabilities. For example, ‘a’ and ‘\’ are adjacent in the Basic Latin block, yet ‘a’ occurs far more frequently than ‘\’. Accordingly, the Pólya tree learner is surprised more often *within* a block than the simpler histogram learner. This explains why PT is less effective on large files than HT.

A Pólya tree learner can operate over bytes as well as tokens. However, since there are only 256 bytes, in practice the results are indistinguishable from the histogram learner of section 3.1. I therefore omit this case for brevity.

For comparison, I include results from the Standard Compression Scheme for Unicode (SCSU) described in section 2.2. It performs poorly, with even the simplistic HB compressor being more effective on the majority of files in the corpus. Furthermore, it fails on binary files and texts not encoded in UTF-8.

I also give results for the general-purpose compression methods gzip and bzip2. Unsurprisingly, these algorithms are usually more effective than the simplistic single-symbol models. However, the power of the token-based approach is demonstrated by the fact that even *without* any contextual learning both HT and PT are more effective than gzip in the case of Japanese text, such as `genji.txt`.

6.3 LZW

Compression algorithms traditionally encode individual bytes, but can often be modified to operate over tokens. I extended LZW, a simple dictionary coder described in section 5.1, to encode symbols from arbitrary alphabets. My method *escapes* to a base distribution when a symbol is first seen, and is described in section 5.2. I report the compression effectiveness of LZW-family compressors in table 6.4, and summarise the results in table 6.5.

File	Size (KiB)	Original		Escaped			Reference	
		LZC	LZA	LUB	LUT	LPT	gzip	bzip2
alice29.txt	149	3.274	3.164	3.154	3.161	3.159	2.863	2.272
asyoulik.txt	122	3.514	3.400	3.392	3.399	3.399	3.128	2.529
cp.html	24	3.680	3.537	3.512	3.559	3.549	2.598	2.479
fields.c	11	3.562	3.410	3.394	3.502	3.499	2.249	2.180
grammar.lsp	4	3.898	3.696	3.666	3.941	3.859	2.653	2.758
kennedy.xls	1006	2.412	2.414	2.413	2.418	2.416	1.606	1.012
lcet10.txt	417	3.058	2.955	2.951	2.953	2.953	2.716	2.019
plravn12.txt	471	3.270	3.186	3.184	3.186	3.186	3.241	2.417
ptt5	501	0.970	0.937	0.936	0.947	0.945	0.880	0.776
sum	37	4.205	4.035	4.051	4.171	4.127	2.703	2.701
xargs.1	4	4.427	4.238	4.171	4.408	4.400	3.308	3.335
beowulf.txt	156	3.190	3.081	3.081	2.981	2.981	2.974	2.221
dostoevsky.txt	1889	2.282	2.080	2.078	1.790	1.790	2.192	1.405
genji.txt	1421	2.501	2.245	2.246	1.756	1.753	2.430	1.545
genji02.txt	66	2.996	2.891	2.878	2.312	2.272	2.629	2.000
kokoro.txt	473	2.679	2.601	2.599	2.002	1.991	2.515	1.702
obiecana.txt	1202	3.278	3.036	3.034	2.965	2.965	3.150	2.276
dictionary.txt	745	4.248	4.055	4.054	3.935	3.902	4.071	2.916
license.html	36	2.889	2.775	2.778	2.351	2.340	1.899	1.502
genji.tar	1450	2.435	2.212	2.213	1.734	1.731	2.404	1.520
kokoziem.tar	1680	3.191	3.057	3.056	2.814	2.811	2.970	2.119

worse ←  → better

Table 6.4: Effectiveness of LZW-family compressors, over the test data in table 6.1. LZC (Frysinger et al. 2015) is a public domain implementation of LZW, compatible with the Unix `compress` program (The Open Group 2013). LZA is an alternative implementation using arithmetic coding (Steinruecken 2014). LUB, LUT and LPT are my escaped variants of LZA, using a uniform byte, uniform token and Pólya token base distribution respectively. `gzip` and `bzip2` are not variants of LZW, but are included for comparison. All figures are given to 3 decimal places. Each cell is shaded to indicate how good the compression rate is relative to other compressors in the table. The best compressor in each row is in **bold**.

Group	LZA	LUB	LPT	gzip	bzip2
ASCII	3.448	3.428	3.501	2.844	2.499
Unicode	2.454	2.450	1.951	2.442	1.663
Mixed	2.964	2.964	2.762	2.674	2.000
Binary	2.531	2.534	2.406	2.113	1.625

Table 6.5: Mean compression effectiveness of LZW variants over the groups in table 6.2. The best compressor in each row is in **bold**.

The original description of LZW suggests encoding dictionary indices as a 12-bit number (Welch 1984). Greater compression effectiveness can be achieved with variable-length codes. The Unix `compress` command, denoted LZC, scales the number of bits b as the dictionary size N grows, such that $2^{b-1} < N \leq 2^b$. However, this is only optimal when b is a power of two, wasting up to one bit otherwise. Compression effectiveness can be substantially improved with arithmetic coding, used in LZA, which encodes each index in $\log_2 N$ bits.

In light of this advantage, I chose LZA as the basis for my escaped implementation of LZW. Surprisingly, LUB – escaping to a uniform byte base distribution – is usually more effective than LZA. I suspect this is because the dictionary in LUB is shorter, as it does not contain unseen bytes. This makes the common case – encoding a byte or sequence of bytes that has already been observed – cheaper, since the range of indices is smaller. However, encoding a byte for the first time becomes more expensive.

As a result, LUB is typically more effective than LZA on files which use a limited range of bytes, such as ASCII texts. On binary files, the methods are comparable, and LZA is sometimes more effective than LUB such as with the file `sum`.

However, the real benefit of this technique comes from the ability to use a token alphabet (rather than bytes) and to choose a base distribution. I test two compressors, LUT and LPT, developed by modifying LZA to

escape to a (static) uniform token distribution and an (adaptive) Pólya tree learner respectively.

On Unicode files, both LUT and LPT substantially outperform byte-based compressors LUB and LZA. In the case of Japanese texts `genji.txt`, `genji02.txt` and `kokoro.txt`, the compression effectiveness is around 0.5 bits/byte greater for the token-based approach.

Of course, this comes at a cost on non-Unicode files. The uniform token distribution places less probability mass on ASCII characters than does the uniform byte distribution.⁶ This makes escaping ASCII characters more expensive. However, each symbol is escaped at most once, so the overhead is only significant on small files such as `grammar.lsp` and `xargs.1`.

LPT achieves greater compression effectiveness than LUT on every file. This strongly supports the use of an (adaptive) Pólya tree base distribution rather than a (static) uniform base distribution over tokens. The Pólya tree is able to reduce the cost of escaping by more rapidly learning which blocks of characters are present in a file.

Although LZW is still widely used today – most notably in the Unix `compress` utility and the GIF file format – more modern approaches such as `gzip` and `bzip2` are typically more effective. Despite this handicap, LPT is more effective than `gzip` on all but one of the single-language Unicode text files.⁷ In the next section, I evaluate the use of a token-based approach with a more sophisticated compression algorithm, PPM.

⁶Note that the Pólya tree learner's initial predictive distribution is uniform over tokens, although it will over time adapt based on the frequency counts.

⁷The single-language Unicode files are `beowulf.txt`, `dostoevsky.txt`, `genji.txt`, `genji02.txt`, `kokoro.txt` and `obiecana.txt`. The compression effectiveness of LPT is comparable to `gzip` on `beowulf.txt`, and is substantially better than `gzip` on the other files. The key determinant is the proportion of multi-byte codewords in the file. This explains why LPT fares comparatively worse on the mixed-language texts `dictionary.txt` and `license.html`, which have a high proportion of ASCII characters.

6.4 PPM

The PPM algorithm makes predictions from context-sensitive frequency counts. When a symbol is first seen in a context, the algorithm backs-off to a shorter context. In the special case that a symbol has never before occurred in the file, the algorithm will back-off to a base distribution over the symbol alphabet. In this section, I investigate the usage of a token alphabet with a uniform or Pólya tree base distribution. See chapter 4 for more information on PPM.

PPM has a number of parameters that can affect compression effectiveness, which I selected by optimising over training data. I start by describing my methodology in section 6.4.1. Next, I report the optimal parameters found in section 6.4.2. Following this, in section 6.4.3 I explore the degree to which the results are sensitive to parameter choice. Finally, in section 6.4.4 I report the compression effectiveness on unseen test data using the previously selected parameters, including a comparison to current state-of-the-art compressors.

6.4.1 Selecting parameters

In my tests I used the PPMG variant of PPM, which is parametrised by a maximal context depth d , a discount parameter $\beta \in [0,1]$ and a concentration parameter $\alpha \in [-\beta, \infty)$.⁸ Although in principle it is possible to optimise these parameters for each file compressed, this is computationally expensive and in practice they are usually set to be constants. To select the parameters, I ran an optimisation procedure over the training corpus given in table 6.6.

The training data consists of texts in a variety of languages. Although the algorithm may sometimes need to compress binary files, it is acceptable

⁸See section 4.3 for a more detailed description of PPMG and my reasons for choosing this method.

File	Size (bytes)	Description
aristotle.txt	437 499	<i>The constitution of the Athenians</i> , Greek
austen.txt	697 801	<i>Pride and prejudice</i> , English
confucius.txt	68 762	<i>Lunyu</i> , Chinese
doyle.txt	574 997	<i>The adventures of Sherlock Holmes</i> , English
forsberg.txt	217 247	<i>Svensk litteraturhistoria</i> , Swedish
gogol.txt	703 077	<i>Evenings on a farm near Dikanka</i> , Russian
jushi.txt	265 117	<i>Tou peng hsien hua</i> , Chinese
rizal.txt	832 432	<i>Ang "filibusterismo"</i> , Tagalog
russel.html	645 570	<i>The foundations of geometry</i> , English
shimazaki.txt	723 482	<i>The broken commandment</i> , Japanese

Table 6.6: Training data used to select parameters for PPM. Available for download at <https://github.com/AdamGleave/MPhilProject/tree/master/src/test/resources/corpora/training>.

for the compression effectiveness to be worse in such cases, so I chose not to include any binary files in the training corpus. This is in contrast to the *test* data in section 6.1 where having a variety of file types was an explicit goal. Otherwise, the selection criteria for the two corpora are similar.

To find the parameters, I minimised the mean of the ratio of compressed output size to original input size over the training corpus. Note that this objective function gives equal weight to every file, regardless of its size. The optimisation was conducted independently for each $d \in \{1, \dots, 9\}$. A 10×10 grid search over $\alpha \in [-1, 3]$ and $\beta \in [0, 1]$ was performed to find an initial guess. Nelder–Mead was used to refine this to an optimal α and β (Nelder and Mead 1965). Finally, the (d, α, β) triple with the lowest mean was selected as the optimum.

Alphabet	Base	d	α_{opt}	β_{opt}	Effectiveness (bits/byte)
Byte	Uniform	5	0.176	0.392	2.036
Byte	Uniform	6*	0.095	0.409	2.016
Token	Uniform	5*	0.001	0.513	1.951
Token	Uniform	6	-0.050	0.520	1.969
Token	Pólya	5*	0.001	0.513	1.938
Token	Pólya	6	-0.050	0.521	1.956

Table 6.7: The optimal (α, β) over the training data, at a given depth, for each alphabet and base distribution tested. Optimal depths are marked with *. Tests were conducted for $d \in \{1, \dots, 9\}$ but for brevity I only report on depths which were optimal for at least one alphabet-base pair. Effectiveness is the mean bits of compressed output per byte of input over the training data. All figures are given to 3 decimal places.

6.4.2 Optimal parameters

Table 6.7 reports the parameters I found to maximise compression effectiveness over the training corpus. The parameters are sensitive to the alphabet used, but not the choice of base distribution, with the optimal parameters for a uniform and Pólya base distribution over tokens equal to 3 decimal places.

When the variability in the number of unique symbols U between files is greater, the optimal value of β tends to be larger, as explained in section 4.3.2. With a token alphabet, U is equal to the number of characters used in the file. For a typical English text, this might be less than a hundred characters. But in a language such as Chinese, many thousands of characters could be used.

By contrast, the number of unique *bytes* in a file is of course at most 256. Consequently, U varies substantially more between files when using a token alphabet rather than a byte alphabet. This property explains why β_{opt} is around 0.1 higher for the compressors using token alphabets.

I found the optimal maximum context depth d to be six bytes and five

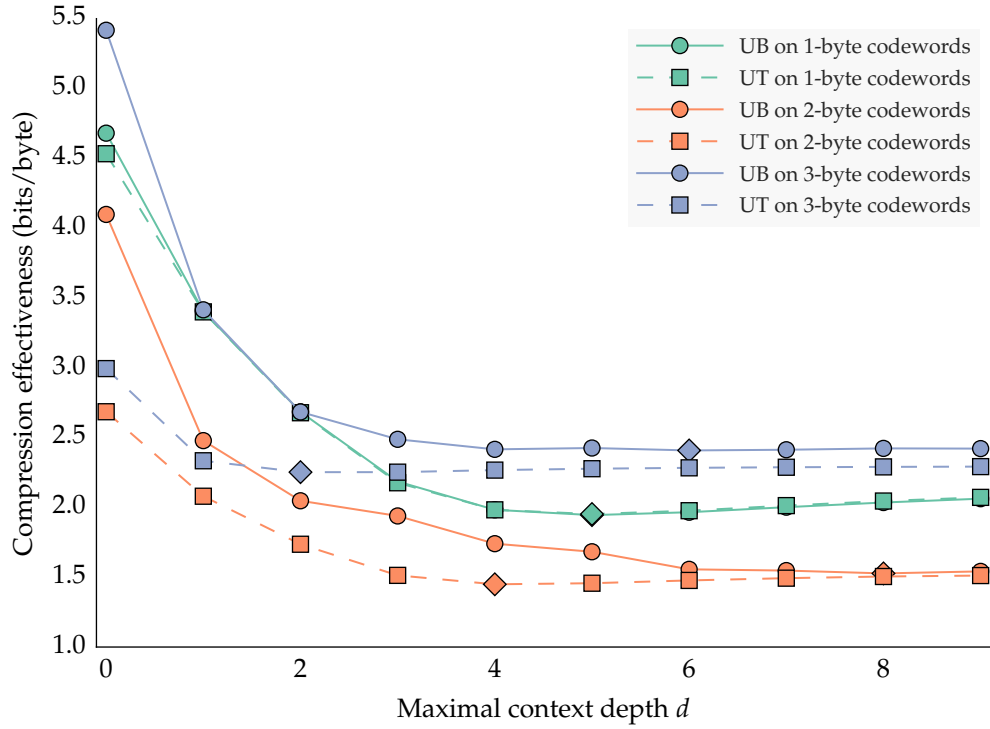


Figure 6.1: The effect of maximal context depth d on the compression effectiveness of PPM. UB and UT denote the use of a uniform base distribution over bytes and tokens respectively. The training data is partitioned into groups, given in table 6.8, based on the length of the UTF-8 codewords in each file. The mean effectiveness is reported for each group. Diamond markers ◆ denote the optimal depth for that compressor-group pair.

Group	$d_{\text{opt}}^{\text{UB}}$ (bytes)	$d_{\text{opt}}^{\text{UT}}$ (tokens)	$d_{\text{opt}}^{\text{UT}}$ (bytes)	Training files
1-byte	5	5	$5 \times 1 = 5$	austen.txt, doyle.txt, forsberg.txt, rizal.txt, russel.html
2-byte	8	4	$4 \times 2 = 8$	aristotle.txt, gogol.txt
3-byte	6	2	$2 \times 3 = 6$	confucius.txt, jushi.txt, shimazaki.txt

Table 6.8: Membership of the groups in figure 6.1, with the optimal depth for the uniform byte and token base distributions UB and UT.

tokens respectively. A Unicode character is represented by a single token, but takes between one to four bytes to encode in UTF-8. Five tokens will therefore correspond to substantially more than six bytes on files with a high proportion of multi-byte codewords.

Figure 6.1 explores how compression effectiveness varies with depth d . The results reported are the mean compression effectiveness over sets of texts, grouped in table 6.8 by the typical UTF-8 codeword length present in the files. The tests were performed on PPM with a uniform base distribution over bytes (UB) and tokens (UT).⁹ Each compressor was run with the (α, β) pair that maximised compression effectiveness over the training data at that depth.

The 1-byte group consists of files containing predominantly ASCII characters.¹⁰ The results for UB and UT are almost identical, since each ASCII character is both a single token and single byte. In the 2-byte and 3-byte groups, UB benefits from a larger value of d than UT. However, the optimal depth d is the same number of *bytes* in both cases.

Intriguingly, the optimal context depth d is larger for the 2-byte group than for the 3-byte group, over both UB and UT. This apparent paradox hints at a more subtle invariant: depth in terms of *information content*.

The 2-byte group consists of Russian and Greek text, which has a similar information content per character to English. By contrast, the 3-byte group is formed of Chinese and Japanese text, where individual characters can represent entire words, and so may therefore have a much greater information content. Accordingly, the optimal depth (in tokens) is much less than for the other two groups.

⁹Using the Pólya tree learner as a base distribution was also tested, but I omit the results as they were extremely similar to UT.

¹⁰forsberg.txt, rizal.txt and russel.html contain a small number of 2-byte codewords.

6.4.3 Robustness of parameters

Previously, I reported the parameters that are optimal for the training data, and discussed why these parameters are optimal. In the next section, I will investigate the compression effectiveness of PPM using these parameters on test data. Before this, it is worth ensuring the parameters selected are reasonable. I will start by testing how close the parameters are to optimality, and then investigate the degree to which PPM is sensitive to parameter choice.

Table 6.9 shows the parameters optimal for the training data, to the left, and the test data, to the right. There is a slight variation, but overall they agree fairly closely. d is always the same, α differs by at most 0.052 and β by at most 0.015.

The impact of these parameters on compression effectiveness is shown in table 6.10. The column $\Delta\bar{e}$ gives the reduction in mean effectiveness over the textual test data due to the use of sub-optimal parameters. This decrease in effectiveness is at most 0.01 bits/byte, small compared to the variation between different compression algorithms.

In light of this, the parameters chosen from the training data are clearly a good fit for the test data. As the training data was chosen independently of the test data, and differs in many important characteristics such as the languages present, this suggests the method of choosing parameters is reasonably robust. However, it is possible – although unlikely – that this is simply good fortune. It is therefore worth investigating how sensitive the compression effectiveness of PPM is to parameter choice.

The relationship between effectiveness and depth d is shown in figure 6.1 from the previous section. Note the curve is relatively flat in the region surrounding the optimum. Picking a depth one too high or too low would therefore incur a relatively small penalty. This result is confirmed by table 6.11, with the penalty never exceeding 0.07 bits/byte, and frequently much lower.

Alphabet	Prior	$d_{\text{TR}}^{\text{opt}}$	$\alpha_{\text{TR}}^{\text{opt}}$	$\beta_{\text{TR}}^{\text{opt}}$	$d_{\text{TE}}^{\text{opt}}$	$\alpha_{\text{TE}}^{\text{opt}}$	$\beta_{\text{TE}}^{\text{opt}}$
Byte	Uniform	6	-0.105	0.478	6	-0.100	0.476
Token	Uniform	4	-0.059	0.498	4	-0.0124	0.484
Token	Pólya	4	-0.065	0.499	4	-0.0130	0.484

Table 6.9: The optimal (d, α, β) , over the training and textual test data, denoted by superscript TR and TE respectively. The training data is listed in table 6.6, and the textual test data is the ASCII, Unicode and mixed groups of table 6.2. All figures are given to 3 decimal places.

Alphabet	Prior	\bar{e}^{TE}	\bar{e}^{TR}	$\Delta\bar{e}$	due to d	due to α, β
Byte	Uniform	2.072	2.075	0.00290	0.00000	0.00290
Token	Uniform	2.080	2.087	0.00611	0.00296	0.00315
Token	Pólya	2.069	2.075	0.00603	0.00286	0.00316

Table 6.10: The effect of parameter choice on compression effectiveness. \bar{e}^D is the mean compression effectiveness, in bits/byte, over the textual test data. If $D = \text{TR}$, the parameters were optimised over the training data; if $D = \text{TE}$, the textual test data was used. The optimal parameters in each case are given in table 6.9. $\Delta\bar{e}$ is the amount by which \bar{e}^{TE} outperforms \bar{e}^{TR} . The following two columns break this down into the amount lost from using sub-optimal depth, and from a sub-optimal (α, β) pair. This is computed by running the compressor with depth optimal on the test data but using the (α, β) optimal at that depth on the training data.

Alphabet	Prior	$d = d_{\text{opt}}^{\text{TE}}$	$\bar{e}_{d-1}^{\text{TE}}$	\bar{e}_d^{TE}	$\bar{e}_{d+1}^{\text{TE}}$
Byte	Uniform	6	2.095 (+0.023)	2.072	2.086 (+0.014)
Token	Uniform	4	2.147 (+0.066)	2.080	2.084 (+0.004)
Token	Pólya	4	2.135 (+0.066)	2.069	2.072 (+0.004)

Table 6.11: PPM compression effectiveness against depth. $d = d_{\text{opt}}^{\text{TE}}$ is the optimal depth on the textual test data. \bar{e}_n^{TE} is the mean effectiveness on the textual test data, with PPM set to depth n using the optimal (α, β) at that depth.

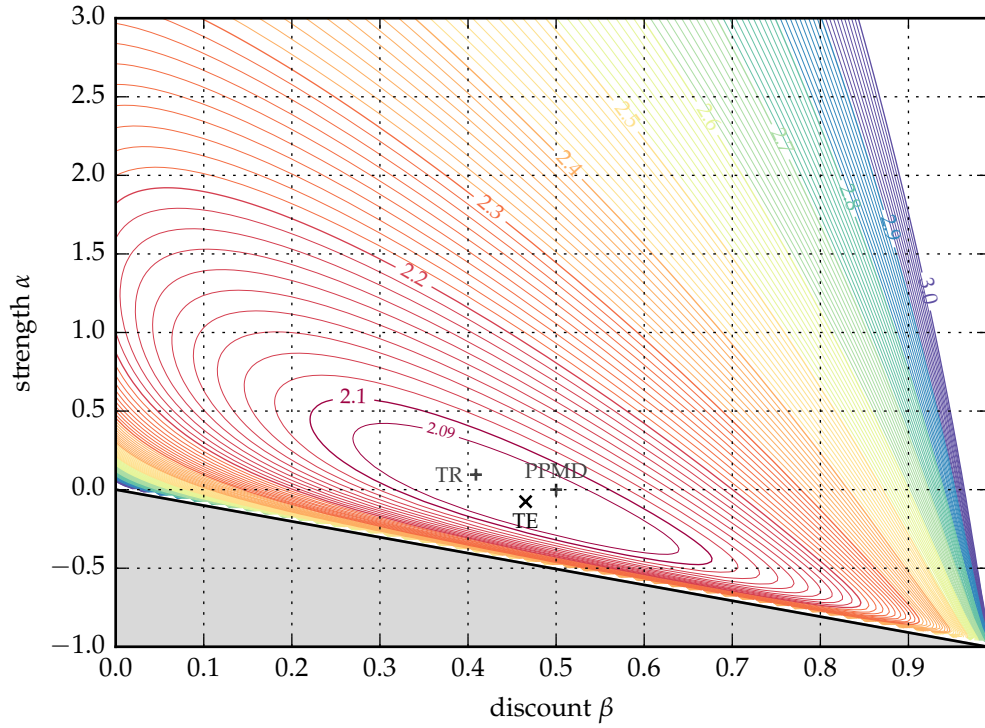


Figure 6.2: Contour plot of compression effectiveness in bits/byte against α and β . The compressor is PPM over a byte alphabet with a uniform base distribution and $d = 6$. Contour heights represent the mean compression effectiveness over the textual test data. TR and TE mark the parameters optimal over the training data and textual test data respectively. PPMD marks the parameters recommended by Howard (1993). The region of invalid parameters, $\alpha + \beta < 0$, is shaded in grey.

Figure 6.2 gives a contour plot of the mean effectiveness, \bar{e} , against α and β . The compressed size in bits/byte climbs gradually away from the optimum (marked TE). This shows that a wide range of parameters can achieve high compression effectiveness. Indeed, the PPMD parameters suggested by Howard (1993) are a mere 0.008 bits/byte from the optimum, and so lie within the innermost contour.

To conclude, I have found that PPM is reasonably robust to parameter choice, achieving high compression effectiveness over a range of parameter settings. This confirms the exact choice of training data used to choose parameters is unimportant, provided it is broadly similar to the test data. Furthermore, my experiments show that the parameters selected in table 6.7 achieve close to optimal performance on the (unseen) test data.

6.4.4 Effectiveness of the token compressor

In this section, I test the hypothesis that a token-based alphabet improves PPM's compression effectiveness. My control is PUB, PPM over a byte alphabet with a uniform base distribution. The experimental algorithms are PUT and PPT, which use the same algorithm as PUB but over a token alphabet with a uniform and Pólya tree base distribution respectively. I set the parameters (d, α, β) to the values in table 6.7, found by optimising over the training data. The results for these and other algorithms (discussed in the following section) are reported in table 6.12.

It is immediately apparent that a Pólya base distribution is superior to a uniform base distribution over tokens, with PPT achieving greater compression effectiveness than PUT on every test file. I will therefore only consider PPT in subsequent discussion.

PPT outperforms PUB on Unicode files (the second and third group in the table) in seven out of eight cases. The file where PUB wins, `beowulf.txt`,

File	Size (KiB)	PPM					Reference		
		PUB	PUT	PPT	P5B	PPMII	CMIX	PAQ	bzip2
alice29.txt	149	2.203	2.189	2.187	2.173	2.101	1.792	1.720	2.272
asyoulik.txt	122	2.502	2.469	2.468	2.457	2.340	2.062	1.964	2.529
cp.html	24	2.312	2.339	2.328	2.305	2.174	1.843	1.685	2.479
fields.c	11	2.073	2.199	2.196	2.085	1.963	1.558	1.554	2.180
grammar.lsp	4	2.408	2.677	2.595	2.402	2.307	2.002	1.881	2.758
kennedy.xls	1006	1.586	1.475	1.473	1.503	0.919	0.067	0.135	1.012
lcet10.txt	417	1.946	1.934	1.933	1.922	1.897	1.523	1.442	2.019
plrabn12.txt	471	2.364	2.317	2.316	2.302	2.238	1.974	1.925	2.417
ptt5	501	0.824	0.822	0.820	0.821	0.781	0.342	0.645	0.776
sum	37	2.734	2.842	2.797	2.743	2.469	1.535	1.764	2.701
xargs.1	4	2.992	3.202	3.195	2.977	2.869	2.604	2.343	3.335
beowulf.txt	156	2.185	2.220	2.219	2.222	2.202	1.720	1.878	2.221
dostoevsky.txt	1889	1.426	1.273	1.273	1.584	1.677	1.113	1.342	1.405
genji.txt	1421	1.452	1.399	1.396	1.601	1.656	1.284	1.433	1.545
genji02.txt	66	1.999	1.923	1.883	2.047	1.983	1.770	1.845	2.000
kokoro.txt	473	1.658	1.600	1.588	1.754	1.747	1.446	1.559	1.702
obiecana.txt	1202	2.128	2.111	2.110	2.110	2.146	1.781	1.839	2.276
dictionary.txt	745	2.864	2.831	2.798	2.822	2.745	2.040	2.066	2.916
license.html	36	1.475	1.448	1.437	1.549	1.559	1.167	1.261	1.502
genji.tar	1450	1.427	1.375	1.372	1.572	1.626	1.259	1.406	1.520
kokoziem.tar	1680	1.993	1.967	1.964	2.007	2.029	1.681	1.754	2.119


worse ←  → better

Table 6.12: Effectiveness of PPM-family compressors, over the test data in table 6.1. PUB, PUT, PPT, P5B use a uniform byte, uniform token, Pólya token and uniform byte base distribution respectively. The parameters (d, α, β) were set to the values in table 6.7, found by optimising over the training data. For P5B, d was forced to 5, with (α, β) taken from the table. As a within-family comparison, I include PPMII, a state-of-the-art PPM-based compressor.

For comparative evaluation, I give results for CMIX and PAQ (short for *paq8hp12any*), two highly effective (though comparatively slow) ensemble compressors. Finally, I include bzip2, for ease of comparison with previous tables.

All figures are given to 3 decimal places. Each cell is shaded to indicate how good the compression rate is relative to other compressors in the table. The best compressor in each row is in **bold**.

is unusual as it is predominantly ASCII, with only 8.9% of its characters coming from other Unicode blocks.

Surprisingly, the token-based compressors were more effective than the control PUB even on the Canterbury corpus (the first group in the table), winning in six out of eleven cases. I suspect this is because PUB uses a depth of six, whereas PPT use a depth of five, which figure 6.1 shows is optimal on ASCII files. PUB needs the longer depth to accommodate files with multi-byte codewords. PPT faces less pressure, as the use of a token alphabet reduces the variation in optimal context depth.

Indeed, I find that P5B – like PUB, but with a depth of five – beats PPT compressors on ten out of eleven Canterbury files. However, its performance over Unicode files is substantially worse than both PUB and PPT.

PPM is a highly successful compression algorithm, and many variants have been developed. PPMII is a particularly sophisticated modification, with state-of-the-art performance on text. It outperforms all of my compressors on every file of the Canterbury corpus. But on Unicode, PPT outperforms PPMII in six out of eight cases, and often by a very substantial margin. Interestingly, PUB also outperforms PPMII in most cases on Unicode text (although by a slimmer margin), suggesting that PPMII has been tuned for English-language texts at the expense of other languages.

Overall, I would conclude that the use of a token-based alphabet improves the compression effectiveness of PPM over a mixture of ASCII and Unicode texts. PPT outperforms PUB on almost all Unicode texts, and most ASCII texts. Variants of PPM which have been tuned for ASCII texts, such as P5B and PPMII, can outperform PPT on ASCII but pay a significant penalty on Unicode texts.

Group	PUB	PPT	PPMII	CMIX	PAQ
ASCII	2.350	2.402	2.236	1.920	1.814
Unicode	1.634	1.535	1.766	1.403	1.545
Mixed	1.930	1.922	1.969	1.556	1.659
Binary	1.713	1.685	1.565	0.977	1.141

Table 6.13: Mean compression effectiveness of PPM variants over the groups in table 6.2. The best compressor in each row is in **bold**.

Comparative evaluation

Previously, I compared variants of PPM to each other, finding PPT to be best. In this section, I test PPT against state-of-the-art compressors CMIX and PAQ. I find PPT to be somewhat less *effective* than these compressors, but substantially more *efficient*, completing in a fraction of the time with a considerably smaller memory footprint.

`cmix v9` (abbreviated CMIX) and `paq8hp12any` (abbreviated PAQ) come first and second place in a comprehensive text benchmark by Mahoney (2016a).¹¹ CMIX is also ranked first in tests over the Silesia corpus, containing a mixture of text and binary files (Deorowicz 2003; Mahoney 2016b).

Both algorithms make single-bit predictions by combining the output from an ensemble of independent models, a technique sometimes called *context mixing*. PAQ has been tuned for English-language text, and is the best compressor in table 6.12 on all English-language files. CMIX is more versatile, using a total of 1740 independent models, including many of those from PAQ. It is the best compressor in table 6.12 for binary and Unicode files.

I summarise the results in table 6.13. On ASCII files, PAQ achieves a mean compression effectiveness of 1.814, 0.422 bits/byte better than the

¹¹By the compressed size of `enwiki8`, the first 10 MiB of English-language Wikipedia. Results last updated May 5th, 2016. I exclude `durilca'kingsize`, which is specialised for the benchmark and is unable to compress files other than `enwiki`.

most effective PPM variant, PPMII. On Unicode texts, my PPM variant PPT just beats PAQ by a 0.010 bits/byte margin. But CMIX is the overall winner, outperforming PPT by 0.132 bits/byte. This gap is substantially smaller than the performance difference on ASCII, confirming the benefit from using a token-based alphabet. Furthermore, this suggests that CMIX and PAQ might benefit from being modified to incorporate knowledge of UTF-8.

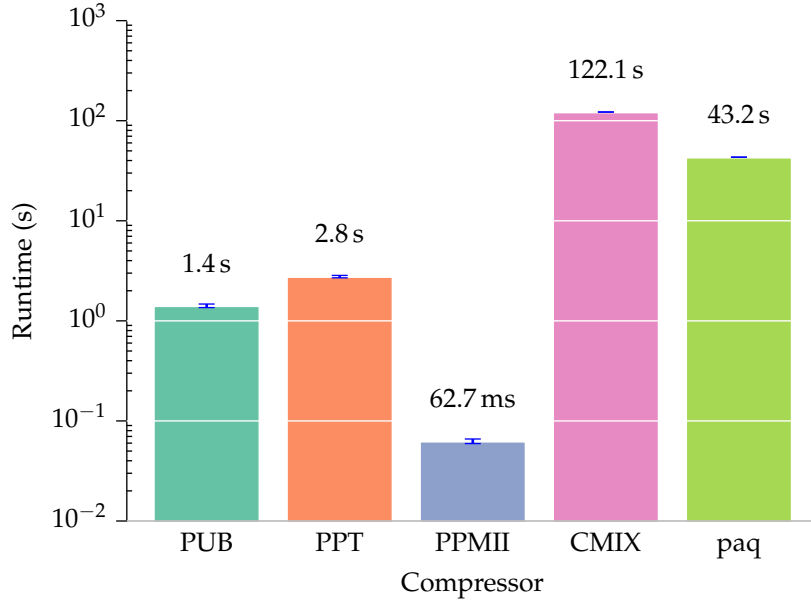
So far, the focus of this evaluation has been on compressor *effectiveness*: the degree to which the input file size can be shrunk. But *efficiency*, how economical the algorithm is in time and space, is of considerable practical importance. To some extent, it is always possible to improve effectiveness at the cost of a longer runtime and larger memory footprint.

Figure 6.3 shows the time and memory taken to compress and decompress the `genji02.txt` test file.¹² The tests were conducted on a machine with specification given in appendix A.2. `genji02.txt` was read once before running any compressors, to ensure it was present in cache. The test was repeated 20 times, to reduce the effect of random variation. Compressors were run round-robin (i.e. each replication ran every compressor) so that if system performance varied over time all compressors would be equally affected.

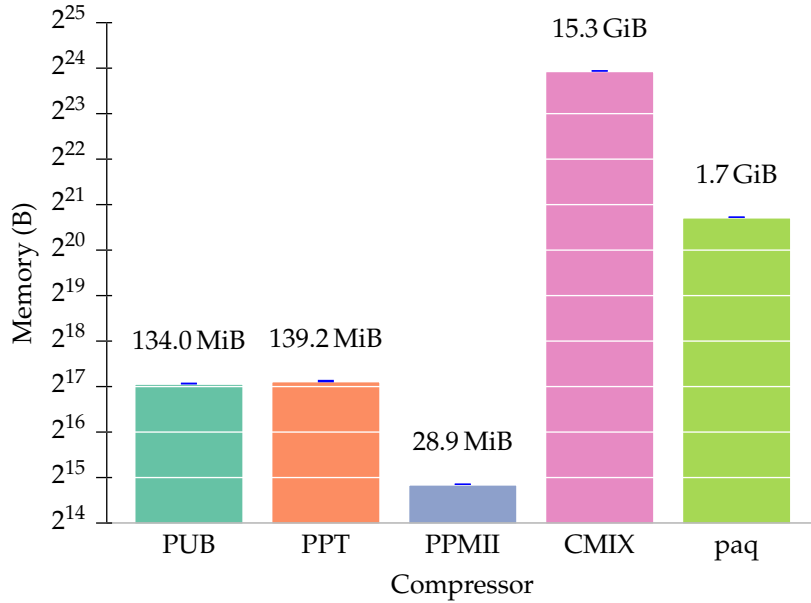
The runtime reported is *wall-clock* time, a measure of the real time taken for the process to complete. This is as opposed to CPU time, the duration the process actually spends executing. I selected wall-clock time as it is the least biased metric. However, it is higher variance than CPU time, being affected by concurrently running processes. To mitigate this, I performed the experiments on a machine that was otherwise idle, and only ran one compressor at a time.

Memory consumption is measured by the *maximum* resident set size of

¹²I conducted tests on other files, finding runtime to be approximately proportional to file size, and memory consumption to be mostly unchanged. The rankings of the algorithms remained the same in all cases. For brevity, I therefore report on just one file.



(a) Time to completion (wall-clock time), base-10 log scale.



(b) Maximum memory usage (resident set size), base-2 log scale.

Figure 6.3: Resource consumption compressing then decompressing *genji02.txt*. The heights of the bars represent the mean result over 20 replications, and the blue whiskers the standard error. Note the log scale necessary due to the large variation in compressor performance. The tests were conducted on a machine with specification described in appendix A.2.

the process.¹³ This is the largest amount of main memory (RAM) used by the process throughout its execution. Equivalently, it is the total amount of available RAM needed for the program to complete successfully. Note the program's virtual address space can be larger,¹⁴ but any pages not resident in memory must have never been accessed by the program, since swap was disabled on the test machine.

It is immediately apparent that the increased effectiveness of CMIX and PAQ comes at a heavy cost in efficiency. CMIX takes $45\times$ longer than PPT and uses $113\times$ as much memory. PAQ is slightly more economical, taking just $16\times$ as long and $12\times$ as much memory. These results are particularly striking since PPT is an unoptimised implementation of PPM intended only for research.

PPMII shows what is possible with some intensive code profiling, running $21.8\times$ faster than PPT and over $1900\times$ faster than CMIX. The use of a token-based alphabet hurts PPM's performance slightly, with PPT around $2\times$ as slow as PUB. Yet even if an optimised implementation of PPT were only within $4\times$ PPMII's speed, it would still be almost $500\times$ faster than CMIX.

To conclude, PPT offers state-of-the-art performance on Unicode text amongst algorithms with comparable resource consumption. My implementation was intended for research only and is likely too slow for many practical purposes, but optimised implementations such as PPMII show that PPM can have excellent performance. Ensemble compressors using orders of magnitude more CPU and memory outperform PPT, although there is reason to believe these compressors would also benefit from incorporating a token-based approach, the key contribution of this dissertation.

¹³As reported in the field `ru_maxrss` by `getrusage` on the Linux kernel.

¹⁴Indeed, CMIX consumes around 24 GiB of virtual memory, but only around 16 GiB physical.

Chapter 7

Conclusions

7.1 Summary

The majority of text online is currently encoded in UTF-8, and this situation is unlikely to change in the foreseeable future. There is therefore a clear utility in the ability to compress such data effectively. Existing methods, such as SCSU and BOCU-1, fall short on two counts: they can *only* compress Unicode data, and are *less* effective than general-purpose compressors such as bzip2.

In this dissertation, I have shown a method for modifying general-purpose compressors to operate over Unicode characters and error tokens. I applied this technique to LZW and PPM (two very different compressors) and found it improves their effectiveness on multilingual UTF-8 files substantially, as shown in tables 6.5 and 6.13. Furthermore, PPT (my version of PPM) is more effective than PPMII (a state-of-the-art variant of PPM) on selected files. PPT is slightly less effective than the ensemble compressor CMIX, but runs $40\times$ faster.

My source code and datasets are open-source and available at <https://github.com/AdamGleave/MPhilProject>. In the next section, I discuss directions for further work.

7.2 Future work

7.2.1 Optimised implementation

The focus of this dissertation has been on compression *effectiveness*: how much the file can be shrunk. But the *efficiency* of an implementation – how economical it is with computational resources – is of considerable practical importance.

Optimised implementations of PPM, such as PPMII, are almost as efficient as widely-used compressors like bzip2.¹ Porting my approach to an implementation such as PPMII seems likely to yield a compressor that is both highly efficient and effective over UTF-8 text.

7.2.2 Applications to other algorithms

I have demonstrated that modifying LZW and PPM to operate over Unicode characters and error tokens substantially improves their compression effectiveness on UTF-8 text. LZW and PPM have radically different designs, so it seems likely that a broad range of compressors might benefit from this technique. However, further work is required to verify this.

7.2.3 Compression of other data formats

My implementations of LZW and PPM can operate over *any* countable alphabet of tokens given a corresponding base distribution. In this dissertation, I used an alphabet of Unicode tokens, and a Pólya tree base distribution (taking advantage of Unicode’s block-based layout).

¹The *Large text compression benchmark* finds that PPMII takes 880s to compress a 100 MiB file compared to bzip2’s 379s (Mahoney 2016a).

Compressors optimised for other data formats can be built by selecting an appropriate alphabet and base distribution. For example, a compressor for executable files could use an alphabet of machine language instructions. The base distribution could be chosen based on observed frequencies of different instructions in a training corpus.

Ideally, a general-purpose compressor should be able to adapt to individual data formats, eliminating the need for such explicit models. However, the currently most effective general-purpose compressors use ensemble models, combining predictions from a large number of specialised models.

Ensemble compressors can achieve excellent compression effectiveness, but are extremely resource intensive. If my technique is as successful with other data formats as it has been for UTF-8 text, then the resulting specialised compressors may be almost as effective as ensemble compressors over files of the target data format, while being orders of magnitude more efficient.

Appendix A

Test results

A.1 Compression effectiveness

My evaluation, in chapter 6, includes tables reporting the effectiveness of particular families of compressors. This makes it straightforward to compare performance within-family, but it can be hard to see the broader perspective. As an alternative, in tables A.1 and A.2 I give results for all the compressors tested in this dissertation, listed in table A.3.

A.2 Machine specification

The resource consumption tests in section 6.4.4 were conducted on a machine with:

- 2× Intel Xeon X5560 CPU. Each has 4 physical cores, running at 2.8 GHz.
- 47 GiB of system memory.
- GNU/Linux (Ubuntu 14.04 *Trusty* operating system, 2014).

File	Size (KiB)	Static		Adaptive			LZW				
		UB	UT	HB	HT	PT	LZC	LZA	LUB	LUT	LPT
alice29.txt	149	8.006	21.046	4.573	4.580	4.653	3.274	3.164	3.154	3.161	3.159
asyoulik.txt	122	8.006	21.046	4.814	4.822	4.832	3.514	3.400	3.392	3.399	3.399
cp.html	24	8.006	21.047	5.265	5.311	5.256	3.680	3.537	3.512	3.559	3.549
fields.c	11	8.006	21.048	5.086	5.192	5.062	3.562	3.410	3.394	3.502	3.499
grammar.lsp	4	8.009	21.052	4.812	5.083	4.762	3.898	3.696	3.666	3.941	3.859
kennedy.xls	1006	8.006	21.045	3.576	3.580	3.614	2.412	2.414	2.413	2.418	2.416
lcet10.txt	417	8.006	21.046	4.671	4.674	4.672	3.058	2.955	2.951	2.953	2.953
plrabn12.txt	471	8.006	21.046	4.533	4.536	4.538	3.270	3.186	3.184	3.186	3.186
ptt5	501	8.006	21.021	1.213	1.216	1.317	0.970	0.937	0.936	0.947	0.945
sum	37	8.006	20.973	5.391	5.487	5.381	4.205	4.035	4.051	4.171	4.127
xargs.1	4	8.009	21.051	5.057	5.288	5.015	4.427	4.238	4.171	4.408	4.400
beowulf.txt	156	8.006	19.319	4.623	4.080	4.073	3.190	3.081	3.081	2.981	2.981
dostoevsky.txt	1889	8.006	11.909	4.016	2.650	2.821	2.282	2.080	2.078	1.790	1.790
genji.txt	1421	8.006	7.118	4.277	2.302	2.337	2.501	2.245	2.246	1.756	1.753
genji02.txt	66	8.006	7.091	4.235	2.456	2.356	2.996	2.891	2.878	2.312	2.272
kokoro.txt	473	8.006	7.061	4.586	2.508	2.519	2.679	2.601	2.599	2.002	1.991
obiecana.txt	1202	8.006	19.607	4.891	4.441	4.577	3.278	3.036	3.034	2.965	2.965
dictionary.txt	745	8.006	17.821	5.845	5.146	5.038	4.248	4.055	4.054	3.935	3.902
license.html	36	8.006	12.960	4.807	3.532	3.538	2.889	2.775	2.778	2.351	2.340
genji.tar	1450	8.006	7.400	4.350	2.384	2.418	2.435	2.212	2.213	1.734	1.731
koziem.tar	1680	8.006	16.078	5.604	4.319	4.420	3.191	3.057	3.056	2.814	2.811

worse ←  → better

Table A.1: Compression effectiveness in bits/byte, over the test data in table 6.1. See table A.2 on opposite page for remaining results. All figures are given to 3 decimal places. Each cell is shaded to indicate how good the compression rate is relative to other compressors in the table. The best compressor in each row is in **bold**.

PPM					Reference					Size	File
PUB	PUT	PPT	P5B	PPMII	SCSU	gzip	bzip2	CMIX	PAQ	(KiB)	
2.203	2.189	2.187	2.173	2.101	8.000	2.863	2.272	1.792	1.720	149	alice29.txt
2.502	2.469	2.468	2.457	2.340	8.000	3.128	2.529	2.062	1.964	122	asyoulik.txt
2.312	2.339	2.328	2.305	2.174	fail	2.598	2.479	1.843	1.685	24	cp.html
2.073	2.199	2.196	2.085	1.963	8.000	2.249	2.180	1.558	1.554	11	fields.c
2.408	2.677	2.595	2.402	2.307	8.000	2.653	2.758	2.002	1.881	4	grammar.lsp
1.586	1.475	1.473	1.503	0.919	fail	1.606	1.012	0.067	0.135	1006	kennedy.xls
1.946	1.934	1.933	1.922	1.897	8.000	2.716	2.019	1.523	1.442	417	lcet10.txt
2.364	2.317	2.316	2.302	2.238	8.000	3.241	2.417	1.974	1.925	471	plrabn12.txt
0.824	0.822	0.820	0.821	0.781	fail	0.880	0.776	0.342	0.645	501	ptt5
2.734	2.842	2.797	2.743	2.469	fail	2.703	2.701	1.535	1.764	37	sum
2.992	3.202	3.195	2.977	2.869	8.000	3.308	3.335	2.604	2.343	4	xargs.1
2.185	2.220	2.219	2.222	2.202	7.547	2.974	2.221	1.720	1.878	156	beowulf.txt
1.426	1.273	1.273	1.584	1.677	4.527	2.192	1.405	1.113	1.342	1889	dostoevsky.txt
1.452	1.399	1.396	1.601	1.656	3.946	2.430	1.545	1.284	1.433	1421	genji.txt
1.999	1.923	1.883	2.047	1.983	3.896	2.629	2.000	1.770	1.845	66	genji02.txt
1.658	1.600	1.588	1.754	1.747	4.350	2.515	1.702	1.446	1.559	473	kokoro.txt
2.128	2.111	2.110	2.110	2.146	7.751	3.150	2.276	1.781	1.839	1202	obiecana.txt
2.864	2.831	2.798	2.822	2.745	7.743	4.071	2.916	2.040	2.066	745	dictionary.txt
1.475	1.448	1.437	1.549	1.559	4.940	1.899	1.502	1.167	1.261	36	license.html
1.427	1.375	1.372	1.572	1.626	4.028	2.404	1.520	1.259	1.406	1450	genji.tar
1.993	1.967	1.964	2.007	2.029	6.794	2.970	2.119	1.681	1.754	1680	kokoziem.tar

Table A.2: Continued from table A.1

Compressor	Description
<i>Single-symbol distributions</i>	
UB	The uniform distribution over $\chi = \{0, 1, \dots, 255, \text{EOF}\}$, where $0, \dots, 255$ represent bytes and EOF is a special symbol indicating end of file.
UT	The uniform distribution over <i>tokens</i> (see section 2.3).
HB	Histogram learner (see section 3.1) with a UB base distribution.
HT	As above, but with a UT base distribution.
PT	Pólya tree learner (see section 3.2) over tokens.
<i>LZW and variants</i>	
LZC	The UNIX <code>compress</code> command (Frysinger et al. 2015). This is a variant of LZW (see section 5.1).
LZA	An implementation of LZW using arithmetic coding and with no memory constraints.
LUB	LZA modified to use escaping, as described in section 5.2, with a UB base distribution.
LUT	As above, but with a UT base distribution.
LPT	As above, but with a PT base distribution.
<i>PPM and variants</i>	
PUB	The implementation of PPM described in chapter 4, with a UB base distribution. Parameters were set to those in table 6.7, selected by optimising over training data.
PUT	As above, but with a UT base distribution.
PPT	As above, but with a PT base distribution.
P5B	As above, but with maximal context depth set to 5.
PPMII	ppmdj, a highly optimised variant of PPM due to Shkarin (2006). It achieves state-of-the-art compression effectiveness on text.
<i>Other compressors</i>	
SCSU	The reference implementation of SCSU (see section 2.2) due to Freytag (1998).
gzip	Version 1.6 of gzip (Deutsch 1996; Gailly 2011).
bzip2	Version 1.0.6 of bzip2 due to Seward (2010).
CMIX	Version 9 of cmix due to Knoll (2016).
PAQ	paq8hp12any due to Rhatushnyak (2007).

Table A.3: List of all compressors tested in this dissertation.

Bibliography

- Åberg, Jan, Yuri M. Shtarkov and Ben J. M. Smeets (1997). 'Estimation of escape probabilities for PPM based on universal source coding theory'. In: *Proceedings of the IEEE International Symposium on Information Theory*. Ulm, Germany (cited on page 25).
- Arnold, Ross and Timothy Clinton Bell (1997). 'A corpus for the evaluation of lossless compression algorithms'. In: *Proceedings of the Data Compression Conference*. Snowbird, UT, USA, pages 201–210 (cited on pages 32–33).
- Atkin, Steve and Ryan Stansifer (2003). *Unicode compression: does size really matter?* Technical report CS-2002-11. Florida Institute of Technology (cited on page 8).
- Cleary, John Gerald and Ian Hugh Witten (1984). 'Data compression using adaptive coding and partial string matching'. In: *IEEE Transactions on Communications* 32.4, pages 396–402 (cited on pages 21–25).
- Cover, Thomas M. and Roger C. King (1978). 'A convergent gambling estimate of the entropy of English'. In: *IEEE Transactions on Information Theory* 24.4, pages 413–421 (cited on page 1).
- Deorowicz, Sebastian (2003). *Silesia compression corpus*. URL: <http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia> (visited on 22/05/2016) (cited on page 54).
- Deutsch, Peter (1996). *GZIP file format specification version 4.3*. RFC 1952. RFC Editor, pages 1–12 (cited on page 66).
- Fenwick, Peter and Simon Brierly (1998). 'Compression of Unicode files'. In: *Proceedings of the Data Compression Conference*. Snowbird, UT, USA, pages 547– (cited on page 8).
- Freytag, Asmus (1998). *SCSU Sample Code*. URL: <http://www.unicode.org/Public/PROGRAMS/SCSU/> (visited on 06/06/2016) (cited on page 66).

- Frigyik, Bela A., Amol Kapila and Maya R. Gupta (2010). *Introduction to the Dirichlet Distribution and Related Processes*. Technical report UWEETR-2010-0006. University of Washington (cited on page 15).
- Frysjinger, Mike, Peter Jannesen, Spencer Thomas, Jim McKie, Steve Davies, Ken Turkowski et al. (2015). *ncompress: A fast, simple LZW file compressor*. URL: <https://github.com/vapier/ncompress> (visited on 18/05/2016) (cited on pages 40, 66).
- Gailly, Jean-loup (2011). *gzip source code*. URL: <http://ftp.gnu.org/gnu/gzip/gzip-1.6.tar.gz> (cited on page 66).
- Howard, Paul G (1993). *The Design and Analysis of Efficient Lossless Data Compression Systems*. Technical report. Brown University (cited on pages 25, 50–51).
- Knoll, Byron (2016). *cmix*. URL: <http://www.byronknoll.com/cmix-v9.zip> (cited on page 66).
- MacKay, David J. C. and Linda C. Bauman Peto (1995). ‘A hierarchical Dirichlet language model’. In: *Natural Language Engineering* 1 (03), pages 289–308 (cited on page 24).
- Mahoney, Matt (2016a). *Large text compression benchmark*. URL: <http://mattmahoney.net/dc/text.html> (visited on 22/05/2016) (cited on pages 54, 60).
- (2016b). *Silesia open source compression benchmark*. URL: <http://mattmahoney.net/dc/silesia.html> (visited on 22/05/2016) (cited on page 54).
- Mauldin, R. Daniel, William D. Sudderth and S. C. Williams (1992). ‘Pólya trees and random distributions’. In: *The Annals of Statistics* 20.3, pages 1203–1221 (cited on page 18).
- Moffat, Alistair (1990). ‘Implementing the PPM data compression scheme’. In: *IEEE Transactions on Communications* 38.11, pages 1917–1921 (cited on pages 24–25).
- Müller, Peter and Abel Rodriguez (2013). *Nonparametric Bayesian Inference*. Institute of Mathematical Statistics (cited on page 18).
- Nelder, John A. and Roger Mead (1965). ‘A simplex method for function minimization’. In: *The Computer Journal* 7.4, pages 308–313 (cited on page 44).
- Pimienta, Daniel, Daniel Prado and Álvaro Blanco (2009). *Twelve years of measuring linguistic diversity in the Internet: balance and perspectives*. Paris: United Nations Educational, Scientific and Cultural Organization (cited on page 34).
- Rhatushnyak, Alexander (2007). *paq8hp12any*. URL: http://mattmahoney.net/dc/paq8hp12any_src.rar (visited on 06/06/2016) (cited on page 66).

- Salomon, David (2012). *Data Compression: The Complete Reference*. Springer Berlin Heidelberg (cited on pages 23, 28).
- Scherer, Markus W. and Mark Davis (2006). *BOCU-1: MIME-compatible Unicode compression*. Technical report UTN 6. The Unicode Consortium (cited on page 7).
- Seward, Julian (2010). *bzip2*. URL: <http://www.bzip.org/1.0.6/bzip2-1.0.6.tar.gz> (cited on page 66).
- Shkarin, Dmitry A. (2006). *ppmdj*. URL: <http://www.compression.ru/ds/ppmdj.rar> (cited on page 66).
- Steinruecken, Christian (2014). 'Lossless Data Compression'. PhD thesis. University of Cambridge (cited on pages 17, 25–26, 40).
- Steinruecken, Christian, Zoubin Ghahramani and David MacKay (2015). 'Improving PPM with dynamic parameter updates'. In: *Proceedings of the Data Compression Conference*. Snowbird, UT, USA, pages 193–202 (cited on page 25).
- Teh, Whye Yee (2010). 'Dirichlet process'. In: *Encyclopedia of Machine Learning*. Springer US, pages 280–287 (cited on page 15).
- The Open Group (2013). 'compress'. In: *The Open Group Base Specifications Issue 7* (cited on page 40).
- The Unicode Consortium (2010). *FAQ - Compression*. URL: <http://www.unicode.org/faq/compression.html> (visited on 25/05/2016) (cited on page 8).
- (2015). *The Unicode Standard, Version 8.0.0*. The Unicode Consortium (cited on pages 5–6).
 - W3Techs (2016a). *Usage of character encodings for websites*. URL: https://w3techs.com/technologies/overview/character_encoding/all (visited on 24/05/2016) (cited on pages 1, 5).
 - (2016b). *Usage of compression for websites*. URL: <https://w3techs.com/technologies/details/ce-compression/all/all> (visited on 27/05/2016) (cited on page 1).
 - (2016c). *Usage statistics and market share of content languages for websites*. URL: http://w3techs.com/technologies/overview/content_language/all (visited on 11/05/2016) (cited on page 34).
- Welch, Terry A (1984). 'A technique for high-performance data compression'. In: *Computer* 17.6, pages 8–19 (cited on pages 27–29, 41).
- Witten, Ian Hugh and Timothy Clinton Bell (1991). 'The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression'. In: *IEEE Transactions on Information Theory* 37.4, pages 1085–1094 (cited on page 25).

- Witten, Ian Hugh, Radford Neal and John Gerald Cleary (1987). 'Arithmetic coding for data compression'. In: *Communications of the ACM* 30.6, pages 520–540 (cited on pages 18, 28).
- Wolf, Misha, Ken Whistler, Charles Wicksteed, Mark Davis, Asmus Freytag and Markus W. Scherer (2005). *A standard compression scheme for Unicode*. Technical report UTS 6. The Unicode Consortium (cited on pages 7–8).
- Yergeau, François (2003). *UTF-8, a transformation format of ISO 10646*. STD 63. RFC Editor, pages 1–14 (cited on page 6).
- Ziv, Jacob and Abraham Lempel (1977). 'A universal algorithm for sequential data compression'. In: *IEEE Transactions on Information Theory* 23.3, pages 337–343 (cited on page 27).
- (1978). 'Compression of individual sequences via variable-rate coding'. In: *IEEE Transactions on Information Theory* 24.5, pages 530–536 (cited on page 27).