Making compression algorithms for Unicode text

Adam Gleave
University of Cambridge
`adam.gleave@cantab.net`

Christian Steinruecken
University of Cambridge
`tcs27@cam.ac.uk`

Abstract

The majority of online content is written in languages other than English, and is most commonly encoded in UTF-8, the world's dominant Unicode character encoding. Traditional compression algorithms typically operate on individual bytes. While this approach works well for the single-byte ASCII encoding, it works poorly for UTF-8, where characters often span multiple bytes. Previous research has focused on developing Unicode compressors from scratch, which often failed to outperform established algorithms such as bzip2.

We develop a technique to modify byte-based compressors to operate directly on Unicode characters, and implement variants of LZW and PPM that apply this technique. We find that our method substantially improves compression effectiveness on a UTF-8 corpus, with our PPM variant outperforming the state-of-the-art PPMII compressor. On ASCII and binary files, our variants perform similarly to the original unmodified compressors.

1 Introduction

The Unicode encoding scheme UTF-8 is the most commonly used character set on the Internet, adopted by 87.0% of websites [1]. Whereas the legacy ASCII encoding represents each character by a single byte, UTF-8 maps non-ASCII characters to sequences of two to four bytes. The majority of websites are now in languages other than English [2, table 8], and contain multi-byte UTF-8 characters. Text compression has yet to adapt to this shift, as most compressors still operate on individual bytes.

To address this deficit, The Unicode Consortium defined the Standard Compression Scheme for Unicode (SCSU) and Binary Ordered Compression for Unicode (BOCU-1) [3, 4]. Both methods exploit the property that adjacent characters in a text tend to have nearby code points.[1] Although these Unicode-aware compressors are useful for compactly representing short strings, Atkin and Stansifer [5] find that on longer texts they are outperformed by gzip and bzip2. Using SCSU as a preprocessing step improves gzip's results slightly, but unmodified bzip2 is still the winner.

In this paper, we investigate an alternative approach. Like Atkin and Stansifer, we make use of existing compression algorithms: in our case, PPM and LZW. However, instead of preprocessing the input, we modify these algorithms to operate directly over Unicode characters rather than bytes. This approach is challenging: there are far more Unicode characters than bytes, making it harder to learn the source distribution. We propose an adaptive character model based on *Pólya trees* that is well-suited for learning Unicode character distributions.

Our approach yields a 12.2% average improvement in compression effectiveness for LZW over a corpus of UTF-8 files. This result compares favourably with the

---

[1]The Unicode code space is divided into *blocks* of characters that belong to the same script, and most texts use characters from only a small number of blocks.

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 1: The mapping between code points and byte sequences for UTF-8 [9, table 3.6].

Unicode-aware LZ77 variant from Fenwick and Brierley that achieved only a 2% improvement [6]. Unfortunately their corpus is not available to us, precluding a direct comparison.

For PPM, our approach gives a 6.1% improvement on our UTF-8 corpus, and outperforms Shkarin's best-in-class PPMII implementation [7]. Moreover, compression effectiveness of our LZW and PPM variants over the Canterbury corpus [8] is no worse than for the byte-based versions of LZW and PPM.

2 A token representation for UTF-8

2.1 Background on UTF-8

The Unicode Standard [9] defines how to encode multilingual text. The latest version at the time of writing, Unicode 9.0.0, contains 128 172 characters from all the world's major writing systems, including contemporary and historic scripts. Each character is identified by a unique integer, called the character's *code point*.

Unicode defines three *encoding schemes* that map code points to byte sequences: UTF-8, UTF-16 and UTF-32. For the interchange of Unicode text, UTF-8 is the *de facto* standard. Fewer than 0.1% of websites are encoded in UTF-16 or UTF-32 [1], although these encoding schemes are often used as an internal representation in programming language APIs.

UTF-8 is a variable-length encoding scheme, mapping code points to *code words* of one to four bytes, as specified in table 1. Code points between U+00 and U+7F are mapped to the ASCII character of the same value. This property is called "ASCII transparency", and is a key reason for the popularity of UTF-8. Another desirable property is that UTF-8 is a self-synchronising code, since the range of valid values for the first byte in a sequence is disjoint from those of the trailing bytes.

Any byte sequence that does not match one of the rows in table 1 is ill-formed. Some other ill-formed sequences also exist (e.g. code points above U+10FFFF and code points from U+D800 to U+DFFF), as described in the RFC specification of UTF-8 [10].

2.2 A token representation

We developed an invertible UTF-8 decoder and encoder that maps between sequences of bytes and sequences of *tokens*. There are three types of tokens: UnicodeChar(c) represents a Unicode character with code point c, ErrorByte(b) represents a byte b in an ill-formed sequence, and EOF is an end-of-file marker.

UTF-8 text is transformed to a sequence of UnicodeChar(c) tokens, terminated by EOF. If decoding fails on byte b then ErrorByte(b) is emitted.[2] Decoding con-

tinues without interruption on the next byte, because UTF-8 is a self-synchronising code. The transform is reversible: any sequence of bytes (not just valid UTF-8) can be transformed into a sequence of tokens and back. This paper describes several techniques for compressing the resulting token sequences.

For computational convenience, our implementation represents tokens as integers. Characters are represented by their code point, with bytes following sequentially afterwards, and `EOF` at the end.

3   Contextless models over tokens

A model may be *adaptive* (learning from past input symbols), or *non-adaptive* (a fixed probability distribution). This section introduces three contextless models over *tokens* as defined in section 2.2. Two of these models are non-adaptive and one is adaptive. All three models are used as components in our Unicode variants of LZW and PPM (described in section 4), where they serve as base models for unseen symbols.

3.1   Non-adaptive: the uniform and UTF-8 distributions

The uniform distribution is a simple example of a non-adaptive model, assigning equal probability mass to each symbol in the alphabet. However, a uniform distribution over the Unicode alphabet is a poor choice if we believe that some scripts (such as the Latin alphabet) are consistently more likely than others (such as Egyptian hieroglyphs).

When compressing UTF-8 input, a natural base model to use is the probability distribution implicitly defined by the UTF-8 encoding itself. This distribution assigns each Unicode character c the probability mass $\Pr(c) = k/256^l$, where l is the length (in bytes) of the code word that c is mapped to and k is a normalising constant (needed because UTF-8 is not a compact representation). Characters with lower code points tend to occur more frequently, so are allocated to shorter code words by UTF-8.

A non-adaptive model can never be a good fit for all texts. In particular, texts are typically written in a single script, resulting in the probability mass being *concentrated* in the contiguous region of code points assigned to that script. In the next section, we describe an adaptive character model that exploits this property when learning the source distribution.

3.2   Adaptive: the Pólya tree learner

Consider a balanced binary search tree whose leaf nodes contain the symbols of the alphabet $\mathcal{X}$. Suppose each internal node i has an associated probability θ_i of choosing the left branch over the right branch. Each symbol $x \in \mathcal{X}$ can be uniquely identified by the path of branching decisions from the root of the tree. The probability of symbol x is defined as the joint probability of these branching decisions:

$$\Pr(x \,|\, \boldsymbol{\theta}) \;=\; \prod_{i \in \mathrm{path}(x)} \mathrm{Bernoulli}(b_i \,|\, \theta_i) \;=\; \prod_{i \in \mathrm{path}(x)} \theta_i^{\,b_i} \,(1 - \theta_i)^{1-b_i}\,. \qquad (1)$$

---

[2]It is possible to extend this scheme to specify the exact cause of a decoding error [11, section 2.3]. However, our tests find little difference in the resulting compression effectiveness, so we favour the simpler approach.
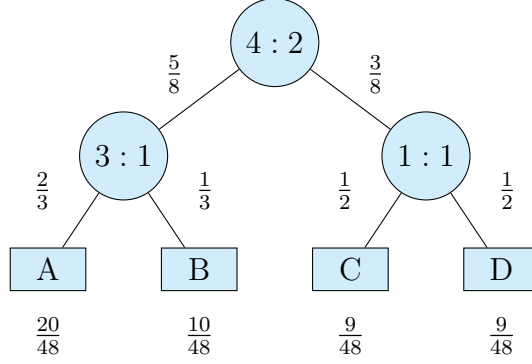
Figure 1: A finite Pólya tree over the alphabet $\mathcal{X} = \{A, B, C, D\}$ after 6 symbol observations. The parameters of the Beta prior at each internal node i were set to $\alpha_i = \beta_i = 1$. Internal nodes are labelled with $L_i : R_i$, where L_i is the number of times the left branch was taken, and R_i the number of times the right branch was taken. Edges are labelled with the posterior branching probability. Each leaf node $x \in \mathcal{X}$ is labelled with its predictive probability $\Pr(x_7 = x \mid x_{1:6}, \boldsymbol{\alpha}, \boldsymbol{\beta})$ from (2).

Here $\mathrm{path}(x)$ is the set of internal nodes from the root to symbol x, $b_i \in \{0, 1\}$ is the branching decision taken at node i, and $\boldsymbol{\theta}$ is the collection of all θ_i.

The node branching biases $\boldsymbol{\theta}$ can be learned adaptively. To derive such a method, we place conjugate priors $\theta_i \sim \mathrm{Beta}(\alpha_i, \beta_i)$ on each θ_i, and maintain two counts L_i and R_i of how often the left and right branches were taken at node i. The posterior over θ_i (given α_i, β_i, L_i, and R_i) is a Beta distribution with parameters $\alpha_i + L_i$ and $\beta_i + R_i$. The θ_i can then be integrated out, resulting in the following closed form expression for the posterior symbol probability:

$$\Pr(x_{k+1} \mid x_{1:k}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \prod_{i \in \mathrm{path}(x_{k+1})} \mathrm{Bernoulli}\left(b_i \,\middle|\, \frac{\alpha_i + L_i}{\alpha_i + \beta_i + L_i + R_i} \right), \qquad (2)$$

where $x_{1:k}$ is the collection of the first k symbol observations, and $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are the collections of all α_i and β_i. The initial predictive distribution before any symbols are observed, $\Pr(x_1 \mid \boldsymbol{\alpha}, \boldsymbol{\beta})$, can be made equal to any chosen distribution by an appropriate choice of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. Learning a new symbol x' simply requires incrementing the matching L_i or R_i counters at each node $i \in \mathrm{path}(x')$. An example Pólya tree is depicted in fig. 1.

Pólya trees have the interesting property that learning a symbol x not only boosts the predictive probability of x (as conventional histogram learning methods do), but also other symbols that are close to x in the tree. This property is apparent in fig. 1: 'B' is assigned a higher probability than 'C' and 'D' (despite all three symbols occurring once each), because 'B' is next to the frequently occurring symbol 'A'.

A Pólya tree can learn any symbol distribution given enough observations, but they are especially fast at learning distributions where symbols in contiguous regions have similar occurrence probabilities. Such distributions often occur in Unicode texts. The Pólya tree character model was first introduced in [12, section 4.3.1].

4 Compressing UTF-8 text

We extend two widely used compressors – PPM and LZW – to operate over tokens instead of bytes, and escape to a base model the first time each symbol is observed.

4.1 PPM

The Prediction by Partial Matching (PPM) algorithm [13] encodes symbols using context-dependent symbol distributions via arithmetic coding [14]. It 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 encoded (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 before, and x is then encoded with the histogram learned for this context. However, when x occurs in the input data for the first time, x will be absent even in the empty context of length 0. In this case, after encoding a final ESC, x is transmitted using a uniform distribution. Our implementation generalises this construction by allowing base models other than the uniform distribution. In particular, we recommend using the Pólya tree as a base model, which can reduce the cost of unseen symbols.

This generic description of PPM does not specify how the histogram learned in each context is used to assign probabilities to symbols. Cleary and Witten [13] state that "in the absence of *a priori* knowledge, there seems to be no theoretical basis for choosing one solution [method] over another". Different methods have been proposed, such as PPMA and PPMB [13], PPMC [15] and PPMD [16]. These methods were designed for PPM compressors operating over byte alphabets, and they might not perform as well over the much larger space of tokens.

We decided to base our experiments on PPMG, a method with two continuous parameters that generalises PPMA and PPMD [12, section 6.4.2]. PPMG has a discount parameter $\beta \in [0, 1]$ and a concentration parameter $\alpha \in [-\beta, \infty)$, and assigns the following symbol and escape probabilities:

$$\Pr(x) = \frac{n_x - \beta}{N + \alpha} \cdot \mathbb{I}\,[n_x > 0] \qquad \text{and} \qquad \Pr(\texttt{ESC}) = \frac{U\beta + \alpha}{N + \alpha}. \tag{3}$$

Here 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.[3]

The α and β parameters have a similar interpretation to those in a Pitman–Yor process [12, section 6.6.3]. Informally, the parameter α can be viewed as a pseudo-count for the escape symbol ESC. The discount parameter β influences the tail behaviour of the symbol distribution, allowing the escape probability to vary based on U, the number of unique symbols observed in the context.

---

[3]During learning, the symbol counts n_x are updated using the standard exclusion method, as pioneered by Moffat [15] in his "second modification" to PPM.

Although we agree with Cleary and Witten that there is no way to design a method without making some *a priori* assumptions, the optimal parameters for PPMG can be determined *a posteriori* from training data. These optimal parameter settings may be useful for compressing files that are similar to the training corpus. We expected PPMG's optimal parameters for byte-based and token-based alphabets to be very different. Surprisingly, we found this not to be the case, as can be seen in table 3.

4.2   LZW

The Lempel–Ziv–Welch algorithm (LZW) is a dictionary compressor [17]. While PPM typically compresses more effectively than LZW, especially on text, LZW and other dictionary compressors are often more computationally efficient. LZW makes for an interesting case study as a representative of the wider class of dictionary coding algorithms.

Unconcerned by memory usage or execution speed, our experimental implementation of LZW places no upper limit on the dictionary size N, and encodes each index in $\log_2 N$ bits using arithmetic coding [14]. Traditional implementations initialise the dictionary to contain every byte. This practice could be extended naïvely to tokens, but at the cost of considerably inflating the size of the dictionary, and thus increasing the number of bits needed to encode each index. This approach would be wasteful, as most tokens will never occur in most files.

We resolve this problem by escaping to a base model over tokens 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 first encodes ε, and then 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. An approach similar to escaping was envisaged by Welch [17, page 11]:

> "The compression string table could be initialised 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 elaborate any further on this method. Our tests in section 5 find that escaping improves compression effectiveness even when using a byte alphabet.

5   Evaluation

5.1   Corpora

We assembled the corpus of UTF-8 texts listed in table 2. First, we collected public-domain texts for all 22 languages with at least 50 million native speakers, excluding English, according to Ethnologue [18]. Next, we sampled ten languages (without replacement) with probability proportional to their number of speakers. For each sampled language we then selected the text with the lowest MD5 hash. We also included a multilingual text, `mix-sake.txt`. Our compressors were tested both on this corpus, and on the English-language Canterbury corpus [8].

File	Size (bytes)	Description
`ara-tabula.txt`	2 330 891	*Tabula Rogeriana*, Arabic geographical text
`ben-kobita.txt`	430 218	*Shesher Kobita*, seminal Bengali novel
`hin-baital.txt`	214 533	*Baital Pachisi*, Indian legend in Hindi
`jav-tuban.txt`	141 694	*Rangsang Tuban*, early Javanese novel
`jpn-yujo.txt`	203 373	*Yujo*, Japanese novel
`lah-wiki.txt`	124 083	*Wikipedia* article, in Punjabi
`por-noites.txt`	115 877	*Noites de Insomnia*, Portuguese book
`rus-mosco.txt`	82 039	*Moscovia v Predstavlenii Inostrantsev*, Russian work
`spa-trans.txt`	311 333	*Transfusión*, Spanish novel
`zho-you.txt`	64 526	*You Xue Qiong Lin*, classical Chinese text
`mix-sake.txt`	257 061	Japanese article, with parallel English translation

Table 2: Our UTF-8 corpus, containing a sample of the world's most common languages.

Label	Base	Alphabet	$d_{\mathbf{opt}}$	$\alpha_{\mathbf{opt}}$	$\beta_{\mathbf{opt}}$
PUB	Uniform	Byte	6	0.095	0.409
PUT	Uniform	Token	5	-0.001	0.514
PPT	Pólya	Token	5	0.000	0.513

Table 3: Optimal settings of the PPMG parameters (d, α, β) on training data. These values are curiously close to $\alpha = 0$ and $\beta = \frac{1}{2}$, the parameter settings that correspond to PPMD [16]. The training data and optimisation procedure are described by Gleave [11, section 6.4].

5.2 Parameter selection

We set the PPMG parameters to values that maximised the compression effectiveness on training data. These values are shown in table 3. The optimal depth is 6 for bytes, but only 5 for tokens, since a token can substitute for multiple bytes. The optimal β is higher under the token alphabet, because the number of unique tokens varies more among files than the number of unique bytes, which is always bounded by 256. These differences highlight the importance of a principled method of parameter selection.

Our experiments found these parameters were almost the same as those that would be optimal on the test corpus described in section 5.1. Furthermore, a sensitivity analysis showed that PPM is fairly robust to parameter choice. The training corpus, our optimisation procedure, and details of these additional experiments are described by Gleave [11, section 6.4].

The Pólya tree's $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ parameters were set such that the initial predictive distribution is equal to the UTF-8 implicit probability distribution.

5.3 Experimental results

We test the hypothesis that a token alphabet improves the compression effectiveness of LZW and PPM on UTF-8 text.

| | File | Size (KiB) | LZW | | | PPM | | | |
			LUB	LUT	LPT	PUB	PUT	PPT	PPMII
ASCII	alice29.txt	149	3.154	3.160	3.153	2.203	2.188	2.181	**2.101**
	asyoulik.txt	122	3.392	3.398	3.391	2.502	2.468	2.461	**2.340**
	cp.html	24	3.512	3.556	3.506	2.312	2.335	2.285	**2.174**
	fields.c	11	3.394	3.494	3.379	2.073	2.191	2.076	**1.963**
	grammar.lsp	4	3.666	3.922	3.633	2.408	2.657	2.371	**2.307**
	lcet10.txt	417	2.951	2.953	2.950	1.946	1.934	1.931	**1.897**
	plrabn12.txt	471	3.184	3.186	3.184	2.364	2.317	2.314	**2.238**
	xargs.1	4	4.171	4.391	4.147	2.992	3.185	2.941	**2.869**
UTF-8	ara-tabula.txt	2276	1.692	1.343	1.342	1.104	0.929	**0.928**	1.268
	ben-kobita.txt	420	1.804	1.427	1.423	1.222	1.097	**1.093**	1.380
	hin-baital.txt	210	1.692	1.246	1.241	0.995	0.815	**0.810**	1.221
	jav-tuban.txt	138	2.962	2.935	2.926	2.187	2.177	2.168	**2.140**
	jpn-yujo.txt	199	2.658	2.038	2.002	1.684	1.621	**1.585**	1.736
	lah-wiki.txt	121	2.638	2.232	2.218	1.748	1.686	**1.672**	1.847
	por-noites.txt	113	3.625	3.607	3.596	2.834	2.801	2.789	**2.663**
	rus-mosco.txt	80	2.839	2.438	2.415	1.846	1.796	**1.772**	1.895
	spa-trans.txt	304	3.116	3.083	3.079	2.266	2.236	2.232	**2.169**
	zho-you.txt	63	4.730	4.391	4.146	3.787	3.628	**3.383**	3.563
	mix-sake.txt	251	3.476	3.060	3.026	2.102	2.060	**2.026**	2.076
Binary	kennedy.xls	1006	2.413	2.418	2.414	1.586	1.474	1.470	**0.919**
	ptt5	501	0.936	0.946	0.941	0.824	0.821	0.816	**0.781**
	sum	37	4.051	4.161	4.067	2.734	2.831	2.737	**2.469**

Table 4: Compression effectiveness in bits/byte. Column key: first character indicates whether compressor is based on LZW (L) or PPM (P); remaining characters specify whether base model is uniform byte (UB), uniform token (UT) or Pólya token (PT). For comparison, we also include PPMII, a PPM variant by Shkarin [7]. 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**.

worse ← ■■■■■■■■■■■■■■■■■■■■ → better

Group	LZA	LUB	LUT	LPT	PUB	PUT	PPT	PPMII
ASCII	3.448	3.428	3.508	3.418	2.350	2.410	2.320	**2.236**
UTF-8	2.841	2.839	2.527	2.492	1.980	1.895	**1.860**	1.996
Binary	2.462	2.467	2.508	2.474	1.715	1.709	1.675	**1.389**
All	3.010	3.002	2.881	2.826	2.078	2.057	2.002	**2.001**

Table 5: Mean compression effectiveness, in bits/byte, over the groups and compressors in table 4. We also include LZA, a LZW compressor equivalent to LUB without escaping. All figures are given to 3 decimal places. The best compressor in each row is in **bold**.

We made three variants of each algorithm: one with a uniform base distribution over the byte alphabet (UB), one with a uniform base distribution over the token alphabet (UT), and one with the Pólya tree base model over the token alphabet (PT). The resulting six algorithms are called LUB, LUT, LPT, and PUB, PUT, PPT.

The results in table 4 show that using a Pólya tree base model improves compression effectiveness: LPT and PPT outperform LUT and PUT on every file. We therefore focus on LPT and PPT in the remainder of this section.

We first implemented a traditional version of LZW, denoted LZA, and then extended it to support escaping. Table 5 shows that on average the escaped version, LUB, is a little better on text files than the original, LZA. This is probably because LUB's dictionary can avoid allocating entries for byte values that do not occur. On binary files, where all byte values may occur, LZA performs slightly better than LUB.

However, the real benefit of using escaping in LZW comes from operating over tokens. On UTF-8 texts, LPT outperforms byte-based LUB on every file, by an average of 0.347 bits/byte. On ASCII and binary files, their performance is the same to within 0.010 bits/byte, with LPT winning on ASCII texts but losing on binary files.

The results are broadly similar for PPM. PPT outperforms PUB on all UTF-8 texts, by an average of 0.120 bits/byte. Surprisingly, PPT also wins on all but one of the ASCII and binary files, with an average improvement of 0.030 bits/byte and 0.040 bits/byte respectively.

To enable a direct comparison between the byte and token alphabets, our PPM implementation intentionally omits many refinements to the original PPM algorithm. Nonetheless, on UTF-8 texts we found it to outperform PPMII, with PPT winning on nine out of twelve files and giving an average improvement of 0.136 bits/byte. However, PPMII wins on all ASCII texts and binary files. Interestingly, PUB often outperforms PPMII on UTF-8 texts (although by a slimmer margin than PPT), which suggests that PPMII may have been tuned for texts encoded in 8-bit alphabets, rather than multi-byte encodings such as UTF-8.

6   Conclusions

We extended the byte-based compressors LZW and PPM to operate over *tokens*: combinations of Unicode characters and byte values for error conditions. Our variants substantially outperform the original algorithms on a UTF-8 corpus that we hope is somewhat representative of the world's current use of languages. Furthermore, PPT (our version of PPM) is more effective than PPMII (an extensively tuned variant of PPM) on most texts in this corpus. In contrast to the existing Unicode compressors SCSU and BOCU-1, our compressors can handle all inputs, including UTF-8 text embedded inside binary formats.

Despite the radically different designs of LZW and PPM, both compressors enjoy a similar improvement in compression effectiveness from operating over tokens. This result suggests that other compressors might also benefit from our technique.

Our source code and data sets are open source and available to download from https://github.com/AdamGleave/UnicodeCompressor.

Acknowledgements

References

[1] W3Techs. (2016, May) Usage of character encodings for websites. `https://w3techs.com/technologies/overview/character_encoding/all`

[2] D. Pimienta, D. Prado, and Á. Blanco, "Twelve years of measuring linguistic diversity in the Internet: balance and perspectives," UNESCO, Tech. Rep. CI.2009/WS/1, 2009.

[3] M. Wolf, K. Whistler, C. Wicksteed, M. Davis, A. Freytag, and M. W. Scherer, "A standard compression scheme for Unicode," The Unicode Consortium, Tech. Rep. UTS 6, 2005.

[4] M. W. Scherer and M. Davis, "BOCU-1: MIME-compatible Unicode compression," The Unicode Consortium, Tech. Rep. UTN 6, 2016.

[5] S. Atkin and R. Stansifer, "Unicode compression: does size really matter?" Florida Institute of Technology, Tech. Rep. CS-2002-11, 2003.

[6] P. Fenwick and S. Brierly, "Compression of Unicode files," in *Proceedings of the Data Compression Conference*, 1998, p. 547, extended version available at `https://www.cs.auckland.ac.nz/~peter-f/FTPfiles/UnicodeDCC98.pdf`.

[7] D. A. Shkarin. (2006) `ppmdj`. An implementation of PPMII, see [19, 20]. Source code. `http://www.compression.ru/ds/ppmdj.rar`

[8] R. Arnold and T. C. Bell, "A corpus for the evaluation of lossless compression algorithms," in *Proceedings of the Data Compression Conference*, 1997, pp. 201–210.

[9] The Unicode Consortium, *The Unicode Standard, Version 9.0.0*, Jun. 2016. `http://www.unicode.org/versions/Unicode9.0.0/`

[10] F. Yergeau, "UTF-8, a transformation format of ISO 10646," RFC 3629 in *Request For Comments*, Internet Engineering Task Force, Nov. 2003. `http://www.ietf.org/rfc/rfc3629`

[11] A. Gleave, "A modular architecture for Unicode text compression," Master's thesis, University of Cambridge, 2016.

[12] C. Steinruecken, "Lossless data compression," Ph.D. dissertation, University of Cambridge, 2014.

[13] J. G. Cleary and I. H. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Transactions on Communications*, vol. 32, no. 4, pp. 396–402, April 1984.

[14] I. H. Witten, R. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, June 1987.

[15] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Transactions on Communications*, vol. 38, no. 11, pp. 1917–1921, Nov. 1990.

[16] P. G. Howard, "The design and analysis of efficient lossless data compression systems," Brown University, Tech. Rep. CS-93-28, 1993.

[17] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, June 1984.

[18] P. M. Lewis, G. F. Simons, and C. D. Fennig, Eds., *Ethnologue: Languages of the World*, 19th ed. Dallas, Texas: SIL International, 2016.

[19] D. A. Shkarin, "Improving the efficiency of the PPM algorithm," *Problems of Information Transmission*, vol. 37, no. 3, pp. 226–235, July 2001, translated from Russian.

[20] D. A. Shkarin, "PPM: One step to practicality," in *Proceedings of the Data Compression Conference*, 2002, pp. 202–211.