An Overhead Reduction Technique For Mega-State Compression Schemes[*]

Abraham Bookstein
University of Chicago
1010 E. 59 St.
Chicago, IL 60637
tel: (773) 702-8268
fax: (773) 702-9861
a-bookstein@uchicago.edu

Shmuel T. Klein
Dept. of Math. & Comp. Sc.
Bar-Ilan University
Ramat-Gan 52900, Israel
tomi@cs.biu.ac.il

Timo Raita
Comp. Sci. Dept.
University of Turku
20520 Turku, Finland
raita@cs.utu.fi

May 1, 1997

[*] The work of the first author (AB) was supported, in part, by NSF Grant IRI-9307895-A01. The author gratefully acknowledges this support. We also wish to acknowledge support given by the Academy of Finland to TR.

Abstract: Many of the most effective compression methods involve complicated models. Unfortunately, as model complexity increases, so does the cost of storing the model itself. This paper examines a method to reduce the amount of storage needed to represent a Markov model with an extended alphabet, by applying a clustering scheme that brings together similar states. Experiments run on a variety of large natural language texts show that much of the overhead of storing the model can be saved at the cost of a very small loss of compression efficiency.

1. Introduction

Text compression is one of the great successes of Information Theory. Simple, but effective, approaches for compressing text, for example, Huffman's classical paper (Huffman, 1952), appeared shortly after the theory itself (Shannon, 1948). Since then, compression effectiveness has come increasingly closer to what we believe to be the limiting value. However, each improvement in effectiveness has involved a growth in complexity.

Modern data compression is strongly model oriented (Rissanen, & Langdon, 1981). The approach generally taken is first to create a model giving the probability of occurrence, within a given context, of a data unit; then to use this probability to encode the unit. The increased complexity of techniques is typically due to the use of larger models. But the model must itself be part of the encoded file for the decoder to work. Unfortunately, a large model may itself be very expensive to store.

We can avoid transmitting the model by using *adaptive* methods. Some of the most efficient compression techniques are adaptive, for example many of the popular schemes, like `zip`, `arj`, `DoubleSpace`, and others. However, for certain applications, e.g. for Information Retrieval (IR) systems, adaptive methods are not always adequate, even when they yield better compression. For, in IR, it is rarely required to decompress the full database; rather, a (possibly large) number of excerpts is retrieved in response to a query. Even should larger blocks of text be required, their sizes will mostly be only a tiny fraction of the full text available. Thus short pieces of the text or concordance are decompressed on demand, and these must be accessed at random by means of pointers to their exact locations. This limits the value of adaptive methods based on tables that systematically change from the beginning to the end of the file.

There are also efficiency reasons that may favor a non-adaptive approach. For example, compression and decompression are not symmetrical tasks in an IR environment. Compression is generally done only once, while building the retrieval system. Decompression, on the other hand, occurs very often. Thus it is realistic to use a technique that is costly in time at the compression stage, provided decompression is fast. Non-adaptive methods, while costly at the compression stage, can be very fast for decompression. In this paper, we concentrate on non-adaptive (sometimes called *static*) compression schemes.

In Bookstein & Klein (1990), text was compressed by creating a simple, first-order Markov model of character generation. Such a model is commonly used for this purpose (see for example, Bell, Cleary, & Witten, 1990), although it is recognized to be an imperfect representation of how text is actually produced. The most common way to improve the performance of a Markov model is by increasing the number of states: for example, we can use higher, or even variable, size contexts (see for example, Cleary & Witten, 1984). In Bookstein & Klein (1990), we took a different approach, which we believed would be simpler, and which in fact turned out to be very effective. Instead of creating larger contexts, we used the Markov model itself to identify the strings for which the model worked badly, and incorporated these strings into the alphabet — to avoid confusion, below we shall use the term *symbol* to refer to elements from the extended alphabet, including both the elementary characters of the initial alphabet and the strings of these characters used to extend that alphabet. That is, instead of changing the state space to include more context, we changed the alphabet and maintained a single symbol history. Since the instances where the initial model performed badly were now in the extended alphabet, the simple Markov model was a better approximation of the text generation process.

This approach retains the simplicity of simple Markov models, but at a cost. The alphabet size can be greatly increased, which means that communicating the model is itself a serious cost. This is most obvious if every codeword is stored explicitly for each context. But for Huffman codes it is not necessary to give the set of codewords explicitly: a *canonical* code can be easily created if one stores only the sequence of elements to be encoded, ordered by frequency, and the optimal lengths of the codewords (see Bookstein & Klein, 1993; Witten, Moffat & Bell, 1994). However, the amount of storage needed for a first order Markov model on an alphabet of size n is still[1] $\Omega(n^2)$, since n codes of up to n elements each have to be kept.

The large size of the model can have serious repercussions. For example, it may make the processing impractical: during compression and decompression, it is convenient to keep the model in RAM. Even a moderately large model may be too large to store in RAM. Such considerations make it useful to search for techniques that reduce model size, and which might make the difference in whether the model can be used.

[1] When expressing asymptotic behavior, we use the standard notations O, Ω and θ to denote, respectively, upper bounds, lower bounds and exact order of magnitude. The formal definitions of these notations may be found in most textbooks on algorithms (for example, Corman, Leiserson & Rivest, 1990).

In this paper, we show that we can reduce much of the model cost with little loss in compression by *grouping* states. Initially, each scanned symbol defines a state, and the following symbol is assigned a codeword dependent on that state. But suppose we now assign a number of symbols to a single state, and create a shared set of codewords for each such state. When we scan a symbol, we enter the state associated with that symbol, and encode the following symbol accordingly. This strategy inverts that of Cormack (1985), who begins with a single state, and breaks this into clusters to improve compression. Cormack, however, chooses a small number of states, which he is able to do very effectively manually. We anticipate having a very large number of states and require an approach that can be automated.

Given an effective model reducing technique, it may well be useful first to expand the alphabet to substantially improve overall compression, then introduce a model reduction method that allows the model to be stored in RAM. Such an approach can be attractive even if it surrenders some of the compression benefit of expanding the alphabet; here we consider the size of the combined file and model, as stored in a header.

It is intriguing to consider the possibility, however, that the reduction in model size might more than compensate for the loss in compression efficiency, resulting in an overall compression advantage. To illustrate this point, consider the impact on a hypothetical 100 megabyte predominantly textual database. Suppose, instead of the standard alphabet of 128 ASCII characters, our methods achieve their best compression if we introduce strings (including words and phrases) into the alphabet, thereby creating an extended alphabet of size 1500.

It is not unreasonable to expect this to reduce the file to, say, 30 megabytes. However, instead of storing the 16,384 elements for a simple Markov model with 128 characters, we would, for an alphabet of size 1500, have to store information for 2,250,000 elements; if each of these requires 4 bytes of data, this would involve a header of nine megabytes. Thus the header would increase the compressed file size by about 30 percent. Such an increase in the space-complexity of the model could well eliminate a good part of the storage saved by the new technique.

Suppose we now reduce the model size by clustering the alphabet. We still need n codewords for each state, but if we have relatively few states as compared to symbols, then we greatly save on the cost of storing the model. But we then deteriorate compression effectiveness, since we are not using optimal codewords. We would like to group states into clusters in a manner that minimizes this cost. For the example cited above, if we reduce the number states from 1500 to 150, we reduce the header size to about one megabyte, saving about 25% of the combined file-header size; if this can be done with a minimal decrease in compression efficiency, then optimal clustering offers an interesting strategy for improving compression by radically increasing the alphabet size. Here, overall compression is improved so long as the compressed file increases by less than 9 megabytes—that is, provided the compression ratio, initially 30%, remains better than 39%, a criterion we believe easily obtainable.

In summary, we believe the clustering of states has at least two advantages in compression. As discussed at length above, it may make it practical to use huge alphabets as a strategy for compressing very large databases. But also, reducing the model size may be useful even for smaller alphabets, by allowing the model to be stored in limited RAM during processing. This would be true even if the overall size of the file/header combination is increased!

Although we shall outline below an overall strategy for choosing an optimal size of alphabet, which, in conjunction with efficient clustering, gives the best overall compression, this is not the main objective of this paper. Rather, we focus on the optimal clustering itself, and test the cost on such clustering on a variety of moderate sized files. We explain the philosophy underlying our clustering procedure in the following section, which also presents the algorithm used for grouping symbols. Section 3 deals with some implementations details. Finally, experimental results of applying the method to textual databases of different sizes and in several natural languages are presented in Section 4.

2. Clustering Probability Vectors

Clustering has often been understood as an *ad hoc* method for grouping items that are in some sense similar. The grouping tends to be rough; in fact, the coarseness of the method, and its freedom from explicit distributional models, is seen as one of its strengths. This has allowed the development of generic clustering algorithms that apply as well to animal species as to documents.

An example is the widespread use of the cosine function as the measure of similarity that drives the clustering algorithm. Having a universal measure is certainly a great convenience, but it has its drawbacks. One immediate loss is that of conceptual precision. The idea of two objects being close to one another is an important one for understanding the nature of the objects. Creating models requires effort, but even when not fully successful, the process forces us to think seriously about the objects we are studying, and often yields real insight.

Using a generic clustering algorithm also has a practical cost. If the clusters formed are sensitive to the associational measure used, we won't get optimal clusters. There is presumably a reason, generally unstated and imprecisely understood, for carrying out the clustering in the first place. To accomplish our ends most effectively, our measure of similarity should reflect our intentions.

Crude clustering techniques will often be adequate, and the benefit of improving the clustering methodology may not justify its cost. Unfortunately, the ubiquity of standard clustering methods causes us to overlook situations where the benefits of an improved technique are substantial. In this paper, we are taking a Goal Oriented Clustering approach (Bookstein A, 1995): the criterion we use to create the clusters are intimately tied to the reason the clusters are being formed.

2.1 Grouping Probability Distributions

The basic idea behind our approach is that if the probabilities associated with two states are similar, the set of codewords they define should likewise be similar. We want to measure the similarity of pairs of vectors of probabilities, and to use this measure to create clusters of states, constructed to minimize the loss in compression.

The conventional clustering approach (see, for example, Jain & Dubes, 1988), would be, given two probability vectors P_1 and P_2, to take their cosine as a measure of closeness, and initiate a clustering algorithm on the basis of this measure. If P_1 and P_2 are similar, their cosine should be close to one, and we would expect to see little deterioration in compression if we: create a centroid vector $P = (P_1 + P_2)/2$; produce a code based on P; and use this code to encode a symbol, if one of the symbols associated with either P_1 or P_2 is scanned. The use of a centroid vector is conventional in clustering, though a sophisticated user might suggest a weighted average, similar to that derived below. (The Kullback-Leibler measure, another measure often used to measure the discrepancy between two probability-distributions, is described below.)

In keeping with the philosophy of Goal Oriented Clustering, we took a different approach. Since our goal was to group states to minimize the compression loss, we used as a measure of distance the *actual* loss, assuming optimal encoding.

2.2 Optimal Cluster Probability-Vector

We first ask: Suppose we partition an alphabet into an arbitrary set $\mathcal{P}$ of symbol-clusters, and require that all the symbols in a given cluster be represented by a single probability-vector — then how should this probability-vector be chosen?

If we have just scanned a symbol s_i of the alphabet, then the probability that the symbol s_k be generated next is p_{ik}. For a given symbol s_k, this probability depends on the preceding symbol. Suppose s_k belongs to the cluster $\mathcal{C}_r$. We now wish to replace each probability p_{ik}, for i such that $s_i \in \mathcal{C}_r$, by a single probability value, $p_{\mathcal{C}_r k}$; $p_{\mathcal{C}_r k}$ generalizes our earlier notation, if p_{ik} is interpreted as shorthand for $p_{\{s_i\}\,k}$. These probabilities determine the codewords of any symbol following a symbol in $\mathcal{C}_r$.

It is easy to calculate the expected size of codewords derived from these approximate probabilities. As noted in Bookstein & Klein (1990), when we create a code based on the (possibly incorrect) assumption that the occurrence probabilities are $\{\hat{p}_k\}$, then the length of the codeword for s_k will be approximately $-\log \hat{p}_k$. We now apply this general result to the cluster probabilities. Suppose that, having just scanned a character in cluster $\mathcal{C}_r$, we use an assumed probability distribution $\hat{p}_{\mathcal{C}_r k}$ to create the codeword for each s_k. Then the above general result asserts that the length of the codeword for s_k will be approximately $-\log \hat{p}_{\mathcal{C}_r k}$, since now $\hat{p}_k = \hat{p}_{\mathcal{C}_r k}$. If θ_i is the true unconditional probability of occurrence of symbol s_i, then the expected length $\widehat{EL}$ of such a code will be approximately

$$\widehat{EL} = \sum_{\mathcal{C}_r \in \mathcal{P}} \sum_{s_i \in \mathcal{C}_r} \theta_i \Big(-\sum_k p_{ik} \log \hat{p}_{\mathcal{C}_r k}\Big). \tag{1}$$

Here the sum over the symbols in the alphabet is broken down first into the sum of symbols in each member $\mathcal{C}_r$ of $\mathcal{P}$, then the sum over the members of $\mathcal{P}$. If no clustering occurs, and the individual symbol probability-vectors are used, then the best compression consistent with our generation model is attained. The expected length in this case, EL_0, is given by:

$$EL_0 = \sum_{\mathcal{C}_r \in \mathcal{P}} \sum_{s_i \in \mathcal{C}_r} \theta_i \Big(-\sum_k p_{ik} \log p_{ik}\Big). \tag{2}$$

As shown in Bookstein & Klein (1990), and proved more directly here:

Theorem 1 *The best summarizing distribution we can use, in the sense of minimizing the expected length of a codeword, is a weighted average. Specifically, the optimal cluster probability-vector for cluster $\mathcal{C}$ is $P_{\mathcal{C}} = (p_{\mathcal{C}1}, \cdots, p_{\mathcal{C}n})$, where:*

$$p_{\mathcal{C}k} \;=\; \sum_{s_i \in \mathcal{C}} \frac{\theta_i}{\theta_{\mathcal{C}}}\, p_{ik}, \tag{3}$$

and,

$$\theta_{\mathcal{C}} \;\equiv\; \sum_{s_i \in \mathcal{C}} \theta_i. \tag{4}$$

In terms of the optimal probability vector, the expected length, EL, may be rewritten as

$$EL = \sum_{\mathcal{C}_r \in \mathcal{P}} \theta_{\mathcal{C}_r} (-\sum_k p_{\mathcal{C}_r k} \log p_{\mathcal{C}_r k}). \tag{5}$$

Proof: Let $\hat{p}_{\mathcal{C}_r k}$ be an arbitrary distribution over the k-index. Then the increase Δ (which we will show to be positive) in expected code-size using $\hat{p}_{\mathcal{C}_r k}$ rather than $p_{\mathcal{C}_r k}$ is

$$\begin{aligned} \Delta &= \sum_{\mathcal{C}_r \in \mathcal{P}} \sum_{s_i \in \mathcal{C}_r} \theta_i \left(- \sum_k p_{ik} \log \frac{\hat{p}_{\mathcal{C}_r k}}{p_{\mathcal{C}_r k}} \right) \\ &= \sum_{\mathcal{C}_r \in \mathcal{P}} \theta_{\mathcal{C}_r} \sum_{s_i \in \mathcal{C}_r} \frac{\theta_i}{\theta_{\mathcal{C}_r}} \left(- \sum_k p_{ik} \log \frac{\hat{p}_{\mathcal{C}_r k}}{p_{\mathcal{C}_r k}} \right) \\ &= \sum_{\mathcal{C}_r \in \mathcal{P}} \theta_{\mathcal{C}_r} \left(- \sum_k p_{\mathcal{C}_r k} \log \frac{\hat{p}_{\mathcal{C}_r k}}{p_{\mathcal{C}_r k}} \right). \end{aligned} \tag{6}$$

Since both the p's and $\hat{p}$'s are true probabilities, it is well known that

$$- \sum_k p_{\mathcal{C}_r k} \log \frac{\hat{p}_{\mathcal{C}_r k}}{p_{\mathcal{C}_r k}} \geq 0,$$

and thus Δ must be non-negative as well.

Equation 5 follows immediately from the definition of $p_{\mathcal{C}_r}$. □

It is interesting to compare eqn's (2) and (5). These are formally the same, except that EL is defined over a set of clusters, whereas EL_0 is defined over the initial alphabet of symbols. In eqn (5), $\theta_{\mathcal{C}_r}$ is the probability that some $s_i \in \mathcal{C}_r$ occurs; it generalizes the notation for the unconditional probability, θ_i. Similarly, the cluster probability-vector $P_{\mathcal{C}_r}$ generalizes the individual symbol probability-vectors. That is, if we interpret an individual symbol, s_i, as a cluster, $\{s_i\}$, of one element, then the same equation describes all levels of clustering[2]. This observation will be useful below.

Theorem 1 shows that the conventional representation of a cluster by its centroid is not generally valid; unless the symbols in a cluster have equal probabilities of occurrence, the vectors associated with more probable symbols should be weighted more heavily, and in precisely the manner indicated by this theorem. This type of weighting would likely have been anticipated by a more sophisticated user, and, in a sense, the theorem can be seen as a rigorous verification such intuition. It is also interesting to note that the probabilities enter only via the ratio of probabilities θ_i and θ_C; thus, if convenient, we could have used any set of values proportional to the θ's instead of the θ's themselves. This observation is true throughout this paper.

2.3 Loss Function

Eqn (5) for the expected length of codewords after clustering allows us to compute the compression cost of clustering: this cost is the increase in the expected number of bits needed to encode a symbol if we partition the alphabet into a set of clusters and use the optimal cluster probability-vectors to derive the codewords. The compression cost allows us to assess the benefit of using one partition rather than another. This increase can be realized with arithmetic coding, and usually well approximated by Huffman coding (Bookstein & Klein, 1993).

2.3.1 Definition of loss-function

The efficiency loss $\mathcal{L}(\mathcal{P}) \equiv EL(\mathcal{P}) - EL(\mathcal{P}_0)$, for $\mathcal{P}_0$ the initial alphabet, can be defined in two steps. The result, while straightforward, is very important and will be stated in the form of a theorem (we retain the symbol $\mathcal{L}$ in each step for simplicity of notation):

Theorem 2 *If we use the optimal cluster probability-vectors, then the loss in compression efficiency, $\mathcal{L}$, can be approximated as follows. First:*

[2] In both cases, EL is a state based entropy function; the distinction is in the state space

- *For a given set, or cluster, of symbols, $\mathcal{C}$, we define the loss by*

$$\mathcal{L}(\mathcal{C}) \;=\; -\sum_{s_i \in \mathcal{C}} \theta_i \sum_k p_{ik} \log \frac{p_{\mathcal{C}k}}{p_{ik}}. \tag{7}$$

  In particular, $\mathcal{L}(\mathcal{C}) = 0$ if $\mathcal{C}$ is a singleton cluster.

- *Given an arbitrary partition $\mathcal{P} = \{\mathcal{C}_r\}$ of the alphabet, then*

$$\mathcal{L}(\mathcal{P}) = \sum_{\mathcal{C}_r \in \mathcal{P}} \mathcal{L}(\mathcal{C}_r). \tag{8}$$

- *An important special case deserves separate mention: If $\mathcal{P}$ is made up of a number of singleton clusters and one multi-symbol cluster $\mathcal{C}$, then $\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{C})$. Thus the loss is determined solely by the items being merged.*

We have derived the cluster probability-vectors and loss function of a partition in terms of the symbols of the underlying alphabet. Our results depend on only two parameters describing each symbol: a probability-vector and the probability of occurrence of the symbol. However, after we define a partition, we formally have a situation similar to the one we began with. Now the clusters that make up a partition appear as *pseudo-symbols*, parallel to the symbols in the initial alphabet; and each has a θ-value and a probability-vector associated with it.

It is now possible, and below will be convenient, to combine clusters in a partition to create a *coarser* partition. We can consider the coarser partition, then, either as made up of clusters of pseudo-symbols, or as being made up of clusters of primary symbols. It is interesting, then, to relate the probability-vectors, θ-weights and loss function of the resulting coarser partition, when defined in terms of the pseudo-symbols, to that obtained by using the parameters of the primary symbols.

2.3.2 Self-consistency property

Suppose then that the clusters are formed by a multi-stage process: starting with $\mathcal{P}_0$, the initial alphabet, in each stage a number of clusters are combined to form a coarser set of clusters. The relationship between the parameters in two stages obeys an important *self-consistency* property; this property allows us to determine the parameters in the *next* stage from their values in the *current* stage, without regard either to history or the parameters of the inital alphabet.

Suppose at the current stage we have a set of pseudo-symbols, $\mathcal{P}$, and are given a partition of this set into subsets $\{\mathcal{C}_i\}$. Then we can create θ-values and optimal probability-vectors $P_{\mathcal{C}_i}$ for each member $\mathcal{C}_i$ by using eqns 3 and 4 on the pseudo-symbols. But each $\mathcal{C}_i$ is itself a cluster of symbols taken from the initial alphabet, and this also defines a set of θ-values and cluster probability vectors. The self-consistency property relates these ways of creating the parameters:

Suppose in the current stage of cluster formation, the pseudo-symbols are $\{\mathcal{C}_1\}, \cdots, \{\mathcal{C}_n\}$, where in the 0-th stage the pseudo-symbols are identical to the initial alphabet. Let $\{\mathcal{C}_1\}, \cdots,$ $\{\mathcal{C}_s\}$ combine to form cluster $\mathcal{C}'$. Then:

- If we define $\theta_{\mathcal{C}'} = \sum_{\mathcal{C}_i \subseteq \mathcal{C}'} \theta_{\mathcal{C}_i}$, then it is obvious that $\theta_{\mathcal{C}'} = \sum_{s_i \in \mathcal{C}'} \theta_i$.
- Similarly, if we define $p_{\mathcal{C}'\,k} = \sum_{\mathcal{C}_i \subseteq \mathcal{C}'} \frac{\theta_{\mathcal{C}_i}}{\theta_{\mathcal{C}'}} p_{\mathcal{C}_i k}$, it is easy to confirm that $p_{\mathcal{C}'\,k} = \sum_{s_i \in \mathcal{C}'} \frac{\theta_i}{\theta_{\mathcal{C}'}} p_{ik}$.

Thus, given any stage of clustering, we can form $\theta_{\mathcal{C}'}$ and $p_{\mathcal{C}'\,k}$ for the next stage using only parameters defined in the preceding stage.

The self-consistency property for the loss function is a bit more complex, as expressed by the following theorem:

Theorem 3 *Suppose we have a partition $\mathcal{P}$, and a second partition $\mathcal{P}'$, all of whose members are unions of members in $\mathcal{P}$. Then we can compute the actual loss $\mathcal{L}(\mathcal{P}')$ by applying eqn (8) based on the elementary symbols. Formally, we could also have computed a value by introducing the parameters of the pseudo-symbols of $\mathcal{P}$ into eqn (8); call this value $\Delta\mathcal{L}(\mathcal{P}, \mathcal{P}')$, or, where the partitions involved are understood, $\Delta\mathcal{L}$ for short. If we do this, we find*

$$\mathcal{L}(\mathcal{P}') = \mathcal{L}(\mathcal{P}) + \Delta\mathcal{L}; \tag{9}$$

that is, $\Delta\mathcal{L}$ is the increment in loss as we combine clusters into coarser clusters.

Proof: Suppose that $\mathcal{P}'$ is a partition coarser than $\mathcal{P}$. We denote members of $\mathcal{P}'$ by $\mathcal{C}'_\ell$ and members of $\mathcal{P}$ by $\mathcal{C}_r$. Then, the formula for $\mathcal{L}(\mathcal{P}')$ becomes, after the sum over elements in $\mathcal{C}'_\ell$ is expressed first as a sum of elements in $\mathcal{C}_r$, then over the components $\mathcal{C}_r$ of $\mathcal{C}'_\ell$:

$$\begin{aligned}
\mathcal{L}(\mathcal{P}') &= -\sum_{\mathcal{C}'_\ell \in \mathcal{P}'} \sum_{s_i \in \mathcal{C}'_\ell} \theta_i \sum_k p_{ik} \log \frac{p_{\mathcal{C}'_\ell k}}{p_{ik}} \\
&= -\sum_{\mathcal{C}'_\ell \in \mathcal{P}'} \sum_{\mathcal{C}_r \subseteq \mathcal{C}'_\ell} \sum_{s_i \in \mathcal{C}_r} \theta_i \sum_k p_{ik} \log \frac{p_{\mathcal{C}'_\ell k}}{p_{\mathcal{C}_r k}} - \sum_{\mathcal{C}'_\ell \in \mathcal{P}'} \sum_{\mathcal{C}_r \subseteq \mathcal{C}'_\ell} \sum_{s_i \in \mathcal{C}_r} \theta_i \sum_k p_{ik} \log \frac{p_{\mathcal{C}_r k}}{p_{ik}},
\end{aligned}$$

where we added and subtracted the term $\theta_i p_{ik} \log p_{\mathcal{C}_r k}$ to every term of the first sum to get the final equation. In effect, we have rewritten $\mathcal{L}(\mathcal{P}') = EL(\mathcal{P}') - EL(\mathcal{P}_0)$ as $(EL(\mathcal{P}') - EL(\mathcal{P})) - (EL(\mathcal{P}) - EL(\mathcal{P}_0))$.

By combining the outer sums, the right-hand summation is immediately seen to be $\mathcal{L}(\mathcal{P})$. But in the left hand term, we can invert the order of the two innermost summations and, recalling eqn (3), carry out the sum over s_i first:

$$\sum_{s_i \in \mathcal{C}_r} \theta_i p_{ik} = \theta_{\mathcal{C}_r} \sum_{s_i \in \mathcal{C}_r} \frac{\theta_i}{\theta_{\mathcal{C}_r}} \, p_{ik} = \theta_{\mathcal{C}_r} \, p_{\mathcal{C}_r k}.$$

Substituting this result, we find that the left-hand summation is:

$$\Delta\mathcal{L} = \sum_{\mathcal{C}'_\ell \in \mathcal{P}'} \Delta\mathcal{L}(\mathcal{C}'_\ell),$$

where,

$$\Delta\mathcal{L}(\mathcal{C}'_\ell) = -\sum_{\mathcal{C}_r \subseteq \mathcal{C}'_\ell} \theta_{\mathcal{C}_r} \sum_k p_{\mathcal{C}_r k} \log \frac{p_{\mathcal{C}'_\ell k}}{p_{\mathcal{C}_r k}}.$$

Bringing together the two sums demonstrates our assertion. □

If we construct a partition $\mathcal{P}'$ by a sequence of r mergings, then, by an obvious inductive argument, we can conclude:

$$\mathcal{L}(\mathcal{P}') = \Delta\mathcal{L}_1 + \Delta\mathcal{L}_2 + \cdots + \Delta\mathcal{L}_r, \tag{10}$$

where $\Delta\mathcal{L}_i$ is the loss function for the i-th merger, computed with the parameters of the pseudo-symbols at the preceding stage. (Of course, the initial $\mathcal{L}(\mathcal{P})$, for $\mathcal{P}$ the original alphabet, is zero.)

Thus we can assert that $\mathcal{L}$ is an *increasing* function in the following sense: if $\mathcal{P}$ and $\mathcal{P}'$ are two partitions of the alphabet, and $\mathcal{P}'$ is coarser than $\mathcal{P}$, then $\mathcal{L}(\mathcal{P}') \geq \mathcal{L}(\mathcal{P})$; this follows from the fact that $\Delta\mathcal{L}$ has the form given in eqn (6), which is non-negative.

Below, a stage will involve the merging of two clusters in $\mathcal{P}$ to form a single cluster in $\mathcal{P}'$. With this restriction, the loss function has properties reminding us of a distance function. To emphasize this, below we shall denote $\Delta\mathcal{L}$ by d_{rs} when $\mathcal{P}'$ is generated from $\mathcal{P}$ by combining two of its elements, $\mathcal{C}_r$ and $\mathcal{C}_s$.

Now that we know, given an arbitrary partition of the alphabet into, say, N clusters, how best to assign a probability distribution to each cluster, we can give a criterion for an *optimal* partition into those N clusters: select that set of N clusters that, with optimal probability-distribution assignment, minimizes the overall loss. If we wish, we could then choose that value of N that minimizes the overall size of the file, including a header that contains information necessary for decoding (see Section 3.3 below).

Finally, recall that the alphabet is itself variable: we could increase the size of the alphabet by adding more, and perhaps longer, strings. Given an alphabet extension algorithm that optimally allows us to choose new strings to append to the alphabet, we can now add another stage of cluster definition: once we know for each size alphabet the optimal cluster set, and the consequent file size, we can then determine the size of the alphabet for which the overall file size is minimized. Thus, potentially, our program permits us through the iterative procedure indicated above, to determine the optimal extended alphabet, the optimal number of clusters, and their definition.

In this paper we consider only the cluster formation problem. We believe finding an optimal partition to be a hard problem. However, a greedy algorithm for approximating this partition is suggested in the next section. A similar process was suggested in the area of image compression in Equitz (1989).

2.4 Description of Algorithm

The self-consistency property noted above strongly suggests a greedy iterative procedure to cluster the symbols. We start with each isolated symbol constituting its own cluster. Then, at each stage, we combine two clusters, such that the loss in compression efficiency caused by this merging action is minimal over all possible cluster pairs. Details on how this loss can be evaluated are presented below. This process is continued until one or more of the following halting conditions is reached:

- the number of clusters is less than some threshold N;
- the overall internal memory requirements (clusters, tables, probabilities or Huffman trees for the successors of each cluster, etc.) are below some threshold M;
- the cumulative loss in compression efficiency relative to the full Markov model is above some threshold ϵ;

- the total size of the compressed file plus model description is minimized.

At first sight, it would seem that the last of these criteria would be preferred; that is, we might in principal successively reduce the number of clusters, keeping track of the total size of the file plus model, for each number of clusters N. We then choose that value of N for which this total sum is minimized. However, there are other considerations that may dominate that of external file storage efficiency. For example, it may be required that during processing the entire model be held in core. In that case, it may be necessary to sacrifice external storage efficiency in order to permit effective processing of the file.

In any case, we proceed iteratively. Let K be the number of iterations performed until the halting condition is reached, where $0 \leq K \leq n-1$. As K varies, we range from high compression, but huge model storage requirements (full Markov model, $K=0$), to lower compression, with low model storage requirements (independence model, $K=n-1$).

It is instructive to examine separately the special case $K=n-1$, which corresponds to the merging of all the elements into a single cluster. If $\mathcal{A}$ denotes the cluster that contains all the symbols of the alphabet, we get from the above definition of the cluster probabilities:

$$p_{\mathcal{A}k} \;=\; \sum_{s_i \in \mathcal{A}} \frac{\theta_i}{\theta_{\mathcal{A}}}\, p_{ik} \;=\; \sum_{s_i \in \mathcal{A}} \theta_i\, p_{ik},$$

since $\theta_{\mathcal{A}} \;=\; \sum_{s_j \in \mathcal{A}} \theta_j \;=\; 1$. Let $P(s_i s_k)$ denote the probability of occurrence of the sequence $s_i s_k$. We have then

$$p_{\mathcal{A}k} \;=\; \sum_{s_i \in \mathcal{A}} \theta_i\, p_{ik} \;=\; \sum_{s_i \in \mathcal{A}} \theta_i\, \frac{P(s_i s_k)}{\theta_i} \;=\; \sum_{s_i \in \mathcal{A}} P(s_i s_k) \;\equiv\; \theta_k.$$

The last equality follows from the fact that when we sum the probabilities of the symbol pairs $s_i s_k$ over all possible first symbols s_i in $\mathcal{A}$, we get θ_k, the unconditional probability of occurrence of s_k. This argument verifies that combining all the vectors into a single cluster is equivalent to having the simple unconditional model.

We begin with $\mathcal{P}_0$ made up of singleton clusters $\mathcal{C}_i = \{s_i\}$, each with its associated symbol probability, θ_i, and symbol probability-vector. After several stages, we have a partition $\mathcal{P}$ comprised of clusters of characters. We now turn to the problem of choosing the pair of clusters belonging to $\mathcal{P}$ that should be merged.

Suppose that at any stage, we wish to to form cluster $\mathcal{C}_{\overline{rs}}$ by combining clusters $\mathcal{C}_r$ and $\mathcal{C}_s$; to simplify our notation, we now denote their overall probabilities by θ_r and θ_s, respectively, and their probability vectors by P_r and P_s. We can now take advantage of eqn (9) to compute the loss in compression efficiency of combining these clusters. We first form the new combined cluster probabilities using equations (3) and (4), as described in Thm 1:

$$P_{\overline{rs}} = \frac{\theta_r}{\theta_{\overline{rs}}} P_r + \frac{\theta_s}{\theta_{\overline{rs}}} P_s, \tag{11}$$

where $\theta_{\overline{rs}} = \theta_r + \theta_s$. Equation (11) is justified by the self consistency property.

The loss of compression efficiency resulting from merging $\mathcal{C}_r$ and $\mathcal{C}_s$ is then given by d_{rs}, where:

$$d_{rs} = -\theta_r \sum_k p_{rk} \log \frac{p_{\overline{rs}\,k}}{p_{rk}} - \theta_s \sum_k p_{sk} \log \frac{p_{\overline{rs}\,k}}{p_{sk}}. \tag{12}$$

The expression for d_{rs} is complex, and we address the technical details of using it iteratively to construct clusters in the next section. We should, however, note how different this is from the cosine measure; and that this measure was not chosen from a standard repertoire, but carefully manufactured while guided by the goal we were attempting.

The form taken by eqn (12) also deserves comment. The commonly used Kullback-Leibler (KL) measure (Kullback, 1959) of the distance between two arbitrary probability distributions $\{p_i\}$ and $\{q_i\}$ is given by

$$d^{\text{KL}} = -\sum_i p_i \log_2 \frac{q_i}{p_i}.$$

Our measure is an alternative to this that 1) is symmetric, and 2) can easily be generalized to measure the extent of disagreement among a *set* of probability distributions. Given two probability distributions, we don't use the KL measure directly. Instead we compute a summary distribution, compute the KL measure between each of the individual distributions and the summary distribution, and take an average of these measures, weighted by the probabilities of the elements that are being combined.

Note also that both the KL and our measure are based on the assumption that the information content of an element with probability p is $-\log_2 p$. This suggests the use of arithmetic codes for compression. We shall however prefer Huffman codes in our implementation, which are much faster and for which the necessary information (lengths of codewords) can be stored more economically than the information required for arithmetic codes (probabilities); the loss incurred by substituting arithmetic codes by Huffman codes is usually small and often negligible (Bookstein & Klein, 1993).

Unfortunately, our measure, restricted to two vectors, is not a true metric: though always positive, symmetric and equal to zero only if the two probability distributions are identical, the triangle inequality is not valid. For example, consider the case of three vectors P_1, P_2, and P_3, with equal probability of occurrence. Thus we have $\theta_i/\theta_{\overline{ij}} = .5$ for each each vector in a pair, regardless of the pair taken. Suppose $P_1 = (.8, .2)$, $P_2 = (.6, .4)$, and $P_3 = (.2, .8)$. Then using the θ-weights as given above, we find $P_{\overline{12}} = (.7, .3)$, $P_{\overline{23}} = (.4, .6)$, and $P_{\overline{13}} = (.5, .5)$. This results in (taking logarithms to base 2) $d_{12} = .023$, $d_{23} = .083$, and $d_{13} = .185$. Clearly, $d_{12} + d_{23} = .106 < .185 = d_{13}$, in conflict with the demands of the triangle inequality.

Our greedy technique, which makes locally optimal decisions at each step, may not necessarily yield a globally optimum partition. Suppose that at some step of the greedy algorithm, clusters $\mathcal{C}_1$ and $\mathcal{C}_2$ are combined, and at the next step $\mathcal{C}_3$ and $\mathcal{C}_4$. The cumulative loss is $d_{12} + d_{34}$.

But it may be that $d_{12} + d_{34} > d_{134}$, for d_{134} the loss associated with the cluster consisting of $\mathcal{C}_1$, $\mathcal{C}_3$, and $\mathcal{C}_4$. Using a sequential algorithm, such a cluster may be formed in two steps by, say, first combining $\mathcal{C}_1$ and $\mathcal{C}_3$, even though $d_{13} > d_{12}$; then combining this cluster with $\mathcal{C}_4$. But if we first combined $\mathcal{C}_1$ and $\mathcal{C}_2$, it would not be possible to subsequently combine $\mathcal{C}_1$ and $\mathcal{C}_3$. Here an early stage of the greedy algorithm makes a choice that prevents us making an even better choice later.

An optimal procedure would then have to inspect all the partitions of the set of symbols into a given number of clusters, and this might be computationally impossible: the number of ways to partition the n symbols into k clusters is a Stirling number of the second kind $\left\{ {n \atop k} \right\}$ (see Knuth D.E. (1973), Exercise 1.2.7–64). But $\left\{ {n \atop k} \right\}$ is asymptotically $e^k\, k^{n-k+1/2}$ (Abramowitz & Stegun, 1965: Section 24.1.4), so if we want, e.g., to reduce the number of clusters to half (where we consider the initial full Markov model to consist of n singleton-clusters), the number of partitions is $\left\{ {n \atop n/2} \right\} = \Omega(n^{n/2})$. However, our experiments (see Section 5) indicate that in practice, our clustering heuristic is so efficient, that a possible improvement by an optimal algorithm will often be negligible. The additional effort to find the optimal partition will thus generally not be worthwhile for realistic applications.

3. Implementation details

3.1 Distance between probability distributions

We first note that our distance measure d_{rs} is not well-defined unless the two probability distributions are defined on the same set of elements. This is important regarding our Markov model, since different symbols may have different sets of successors. We circumvent this by using the full alphabet for each probability distribution, assigning zeroes as the probability values for non-occurring symbols. These zeroes cause problems if the KL measure is used, since the KL distance from $\{p_i\}$ to $\{q_i\}$ is infinite if for some i, $q_i = 0$ but $p_i \neq 0$. On the other hand, our measure (eqn. (12)) is well-defined, since by eqn. (11), $p_{\overline{rs}\,k} = 0$ only if either the k-th components of both P_r and P_s are zero, or if, say, the k-th component of P_r is not zero, but the unconditional probability θ_r is zero. In both cases the k-th term of the each of the summations in eqn. (12) is also zero, under the usual convention that $0 \log 0 \equiv 0$.

3.2 Merging clusters

Our algorithm iteratively chooses the pair of clusters giving minimum d_{ij}. For an alphabet of size n, we can accomplish this straightforwardly by building a two-dimensional table $\mathcal{D}$, of size n^2. Such a table would be symmetric, and all entries on the diagonal are zero; thus only $\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$ entries are needed. The construction of the initial table requires $n(n-1)/2$ evaluations of loss values d_{ij}, each of which involves two summations over the full alphabet. The complexity of the initialization of $\mathcal{D}$ is thus $\theta(n^3)$.

We now proceed iteratively by finding the minimum value in $\mathcal{D}$. Suppose the minimal element is found at position (i, j). We would then modify the matrix so as to eliminate reference to clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ and introduce a new cluster $\mathcal{C}_{\overline{ij}}$. The complexity of the iterative part could be reduced by using a *heap* for managing the requirement of successively finding minimum distances, but the overall complexity does not change, as it is dominated by the initialization of table $\mathcal{D}$.

We also have to keep track, during the clustering process, of the current partition of the alphabet into clusters. This is not needed for the clustering itself, since we handle only cluster names, and not their individual components; at the end of the process, however, we do need to know the final partition for the encoding and decoding algorithms. A simple way to monitor the formation of the clusters is by representing each cluster by a *rooted tree*, using Union-Find algorithms (Cormen, Leiserson & Rivest, 1990). Of course, once the clustering is completed, there is no sense in keeping information about the partition in the form of a forest of rooted trees, since during the encoding and decoding phases, we need many accesses to the clusters of a given element. So before starting the encoding process, we construct in a single scan of the alphabet, a vector giving for each symbol the name of the cluster it belongs to.

3.3 Additional overhead

The main reason for using the clustering technique is to reduce the huge overhead implied by mega-state models. In fact, this overhead must be considered in two different contexts:

- at run-time: the decoding trees, the extended alphabet and the partition into clusters have to be available in RAM to permit fast decompression; the required space depends on the extent to which we are willing to sacrifice space for an increase in speed.
- in a header-file: in storage, the necessary information that allows the construction of the decoding trees must be kept along with the file.

We now concentrate on the header file. The succinct coding of this meta-information is a challenge in itself (Bookstein & Klein, 1993), and we suggest here one possible solution.

The header can be schematically represented as follows:

$\langle$`header`$\rangle$::=$\langle$`extended alphabet`$\rangle\langle$`cluster definitions`$\rangle\langle$`definition of Huffman trees`$\rangle$.

We now shall describe each in turn.

`1. Extended alphabet`: The extended alphabet consists of an ordered list of symbols. Below we shall be interested in the actual realization of the symbols as strings. We emphasize below that some of these symbols are actually strings of one or more characters taken from an initial alphabet by referring to them as *meta-characters*. To represent the extended alphabet, we concatenate these meta-characters into a single string S, proceeding by increasing length, where the length of a meta-character x is defined as the number of eg ASCII characters forming x. The string S is preceded by the sequence $R = r_2, r_3, \ldots, r_m$, where r_i is the number of meta-characters of length i, and the value r_1 is assumed known; R is preceded by m, the length of the longest meta-character. Thus the size of the extended alphabet is given by $\sum_{i=1}^{m} r_i$, and $|S| = \sum_{i=1}^{m} i\, r_i$.

2. `Cluster definitions`: We first store the number K of clusters, followed by the numbers $t_1, \ldots, t_K$, where t_i is the number of symbols constituting cluster i, followed by a list of indices to the $\sum_i t_i$ meta-characters themselves.

3. `Definition of Huffman trees`: For each of the clusters $\mathcal{C}_i$, one has to store the Huffman tree corresponding to the distribution of the symbols that are successors of $\mathcal{C}_i$. In fact, it suffices to store the sequence of the successors of $\mathcal{C}_i$, sorted by decreasing frequency, along with the sequence of numbers $k, n_1, \ldots, n_k$, where k is the depth of the Huffman tree (the length in bits of the longest codeword), and n_j is the number of codewords of length j bits. Let N_i be the number of successors of $\mathcal{C}_i$, so that $N_i = \sum_{j=1}^{k} n_j$. The string $T = \langle n_1, \ldots, n_k \rangle$ uniquely determines a canonical Huffman code as follows: first derive from T the length t_j of the j-th codeword, for $1 \le j \le N_i$; the j-th codeword itself consists then of the first t_j bits immediately to the right of the "binary point" in the infinite binary expansion of $\sum_{s=1}^{j-1} 2^{-t_s}$, for $j = 1, \ldots, N_i$ (Gilbert & Moore, 1959).

To store the indices of the the meta-characters in the header, $\lceil \log_2(n) \rceil$ bits are needed for each, but since $n - K$ trees are stored, there are many repetitions. It could thus be that one can save by using a Huffman code to encode these meta-characters; this Huffman code would be internal to the header. The numbers in the header are most easily stored in fixed length format, using $\lceil \log_2(max) \rceil$ bits for each, where max is the number of distinct symbols in the original alphabet — however, since most of the numbers are small, some variable-length universal encoding of the integers may be preferable. A large variety of such encodings can be found in Bell, Cleary, & Witten (1990). More methods for encoding the header are discussed in Bookstein & Klein (1993).

The largest component of the header is clearly the description of the Huffman trees, which is linear in the number of clusters. This is the reason why reducing the number of clusters generates savings over the complete Markov representation.

4. Experimental Data

Since our primary concern is IR applications, we decided to test our clustering algorithm on files in several natural languages. Since our main motivation is to closely examine the trade-off of clustering and compression efficiency, we restrict our experiments to moderately sized databases.

The first set of files consists of the Bible in English (King James version), Finnish, German (Elberfelder Übersetzung) and Hebrew, and a French text by Voltaire called *Dictionaire philosophique*. Each of these texts is large, so the size of the header, relative to the main text, is not significant — to simplify our experiments, we are working with a relatively small, fixed, extended alphabet, and are not considering here the possibility, noted earlier, of expanding the size of the alphabet enough to optimize overall compression. However, it is still useful to reduce the size of the header to facilitate processing of the file in RAM.

We also included a set of smaller texts, for which the header, when considered as part of the compressed file, has a dramatic impact on compression efficiency. This set includes the first 10000 words of *Gadsby*, the famous novel by E. Wright in which the letter `E` never occurs; and the files `progc`, `progl`, and `progp` from the Calgary corpus (Bell, Cleary, & Witten, 1990), which are programs in C, Lisp and Pascal, respectively. Finally, we also studied other non-textual files from the Calgary collection: `obj1`, some object code, and `pic`, a facsimile picture. These were chosen just for comparison, since the size n of the extended alphabet for `obj1` is much larger, and the compression for `pic` is much better, than for the other test files.

For each of these files, we first selected the set of meta-characters that will constitute the symbols of our extended alphabet, and then applied Huffman coding on a full Markov model that uses a different Huffman tree for the successors of each of the symbols. The selection of meta-characters was done automatically by a simple procedure, and there was no attempt to optimize this selection process. Nonetheless, some of the meta-characters strongly reflect the language or the type of file they were taken from. For example, the English Bible produced strings like[3] `ith␣the␣`, `y␣shall`, `LORD.`; in Finnish `mppeli`, `Jumal`, `␣Ja␣`; in German: `icht␣der␣HERR`, `König`, `Volk`; in Hebrew (translated): House of God, King in, David; in French: `hommes`, `vertu␣`, `je␣su`; in the C-program: `fprintf(stderr,␣`, `endif`; in the Pascal program: `begin`, `writeln`. The string `Egyp` appeared both in English and Finnish,

[3] blanks are visualized by the symbol ␣

as did Ägyp in German. In the object code and picture file, the improved compression is due to a meta-character of 256 nulls.

Table 1 summarizes the main statistics describing these files. The sizes of the files are given in bytes, and the column entitled n gives the total size of the meta-alphabet (the number of ASCII characters used in the file plus the number of character strings). The column entitled *compressed* lists the size of the compressed file, excluding the header, expressed as number of bits per original character used. For example, this value is 2.5065 for the English Bible, which means that the size of the compressed file is $(2.5065*3230258)/8 = 1.012$ MB. The next column gives the size of the header file in bytes, and the column entitled *compressed with head* gives the total size of the file, including the header, expressed as the ratio of bits per character as before. We calculated the size of the header without any attempt at optimization, using one byte for each of the values r_i and n_j mentioned in Section 3.3, and 9 bits to refer to each meta-character.

File	Size (bytes)	n	compressed (bits/char)	header (bytes)	compressed with head	compress	gzip	ppmc
English Bible	3230258	243	2.51	16239	2.55	2.79	2.31	
Finnish Bible	3178050	251	2.71	19934	2.76	3.14	2.69	
German Bible	3518702	296	2.63	21175	2.68	3.14	2.70	
Hebrew Bible	1519526	182	3.13	14764	3.21	3.47	3.00	
Voltaire	554719	219	2.51	13559	2.71	3.07	2.81	
Gadsby	55840	215	2.57	7834	3.69	3.63	3.18	
progc	39611	245	2.59	6540	3.91	3.87	2.68	2.49
progl	71646	230	2.45	5441	3.05	3.03	1.80	1.90
progp	49379	242	1.99	5874	2.94	3.11	1.81	1.84
obj1	21504	430	2.72	10607	6.67	5.23	3.84	3.76
pic	513216	212	0.78	8252	0.91	0.97	0.82	1.09

Table 1: *Statistics and compression results*

The final three columns assess the compression efficiency. The values for compress were produced by the standard Unix compress utility, which is based on the LZW algorithm (Welch, 1984); gzip is based on the first Lempel-Ziv algorithm (Ziv & Lempel, 1977), and was applied with the parameter yielding maximal compression; the numbers for ppmc are taken from Bell, Cleary, & Witten (1990) for the files of the Calgary corpus. These three are adaptive methods, even though we are concerned with static methods only. We give these values to set some baselines, rather than as a measure of relative merit.

Note, nevertheless, that even though we did not at all try to find an optimal model, the compression with the full Markov model is often almost as good as with gzip or ppmc and sometimes even better. Depending on the size and the compressibility of the file at hand, the size of the header file, for the large natural language files, is 2–8% of the compressed file, which is not negligible. For small or non-textual files, the header may even be larger than the compressed file itself: see, for example, obj1, which has been compressed to 7314 bytes! There is thus good reason for trying to reduce the header, in particular for applications on smaller machines with scarce internal memory.

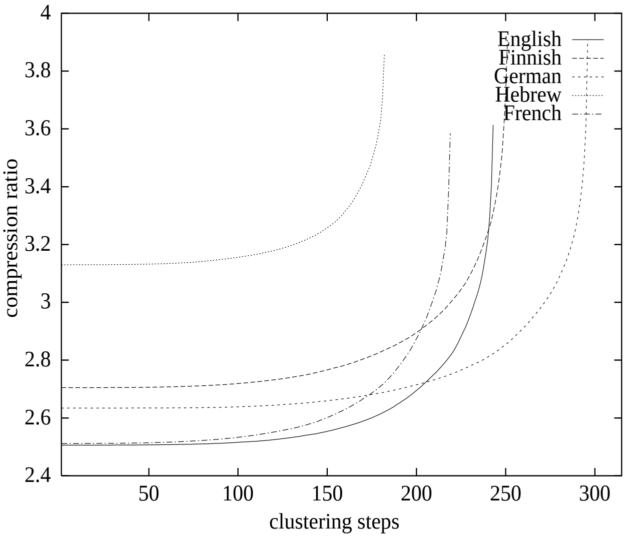


Figure 1: *Compression tradeoffs*

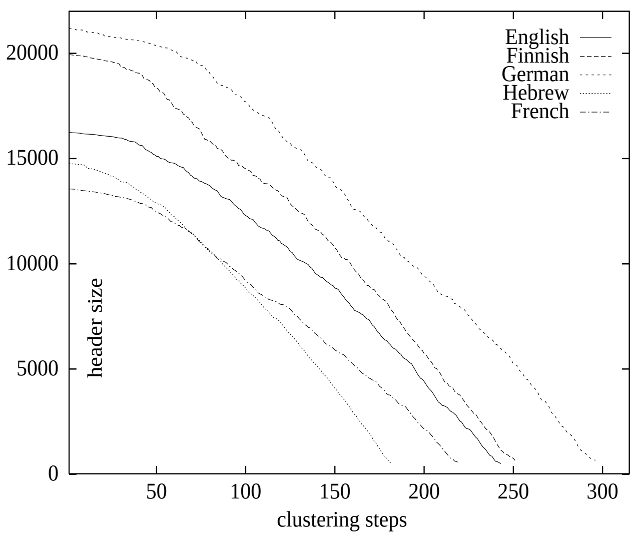


Figure 2: *Header size*

Figure 1 is a graphical representation of the compression efficiency as a function of the number of clustering steps performed, for the five large natural language files of the first set. Compression is measured, as before, in bits per original character, but omits the size of the header. As can be seen, the general form of the graphs for the different languages is similar, and this was also true for all the other files we checked. We note that the increase in size of the compressed file is almost negligible at the beginning (actually of the order of several bits (!)

even for MegaByte large files), and becomes significant only when the number of remaining clusters is reduced to a few tens.

Such behavior, at least at the beginning, might be expected: in a large alphabet, there are often several symbols that either occur only rarely, or are almost always followed by the same symbols, with similar probability distributions. In either case, such symbols can be merged into a single cluster with little cost. In the extreme case we might merge two symbols which have the same set of successors; should these successors occur with frequencies that are similar enough to generate the same Huffman code, the compressed file does not increase at all.

On the other hand, as can be seen in Figure 2, the size of the header, measured in bytes, decreases steadily and almost reaches zero, again with no striking difference in form between the languages.

File name	After 75 clustering steps		After 50% clustering steps		After reducing to single cluster	
	loss (%)	head	loss (%)	head	loss (%)	head
English Bible	0.14	13919	0.81	10862	44.2	506
Finnish Bible	0.19	16252	1.17	12755	44.4	659
German Bible	0.04	19425	0.91	14079	48.3	657
Hebrew Bible	0.30	11032	0.61	9658	23.2	503
Voltaire	0.35	11002	1.18	8454	42.7	571

Table 2: *Tradeoff details on natural language files*

Table 2 displays the exact values for selected points of the above graphs. The first pair of columns corresponds to the values after 75 merging steps have been performed. The next pair of columns gives the values after the number of clusters has been reduced to half of the size of the original extended alphabet (given in the column headed n of Table 1). Finally, the last columns correspond to a single cluster, i.e., simple, unconditional Huffman coding. The values in the columns headed *loss* give the increase, in percent, of the size of the compressed file, relative to the compression obtained by the full Markov model; this loss is of the file itself, not including the header. The columns headed *head* are the size of the header in bytes. In particular, the last column lists the size needed to store the extended alphabet and the single Huffman code. We see that even after hundreds of clustering steps, the loss of compression is still around 1% only, whereas the size of the header has decreased considerably. The values for simple Huffman coding correspond to the right upper top of the graphs in Figure 1. For example, using simple Huffman coding increases the size of the English Bible by 44.2%, to 3.62 bits per character; we note that this is still much better than using a Huffman code on English text without extending the alphabet first.

Since in successive clustering steps the size of the compressed file increases, while at the same time the size of the header decreases, we next consider the combined size. For our natural language files, the plots are almost identical to the corresponding ones in Figure 1, since the size of the header is very small as compared to the texts. But for the smaller files, the combined size clearly decreases, reaches some minimum, and then increases.

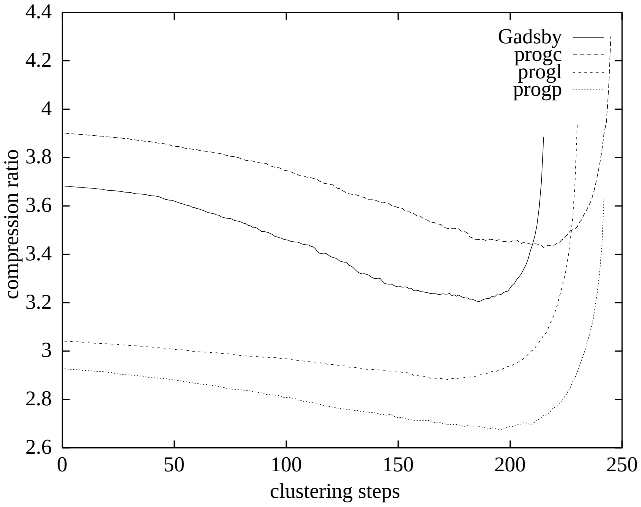


Figure 3: *Compression tradeoffs*

File name	K_{opt}	compr size	head size	total compr
Gadsby	186	20373	1997	3.205
progc	215	14923	2052	3.428
progl	172	22839	2973	2.882
progp	195	13971	2535	2.674
obj1	393	9426	3294	4.732
pic	191	51336	2818	0.844

Table 3: *Tradeoff details*

For the file Gadsby and the three program files, the total size of the compressed file plus header expressed in bits per character, as a function of the number of clustering steps, is displayed in Figure 3. Details on the exact point at which the minimum is reached for each of these files are given in Table 3. The first column gives the number K_{opt} of merging steps for which the combined size is minimized. The next two columns are the sizes of the compressed file and the header in bytes, after K_{opt} merging steps, and the last column gives the total size, again in bits per character. We see that for the smaller files, clustering not only reduces the space needed to store the model, but can actually reduce the space required to store the model and file combined. In our examples, performing K_{opt} merging steps reduced the combined size by 5–13%, and on the `obj1` file, by as much as 29%. Note also that K_{opt} is surprisingly high relative to the size of the extended alphabet.

The following are a few examples of elements that clustered. As one might expect, their Huffman trees were often quite similar. For example, for the English file, the meta-characters `␣sai` and `.␣An` were united at an early stage. Indeed, the successors of the first were: `d␣`, `d`, `th␣`, `th`, `nt`, and of the second `d␣`, `d`, `␣`, `␣h`, `s`, `␣a`. But for both, the probability of `d␣` was the highest (0.58 and 0.99 respectively). A (canonical) Huffman code can be defined by the number of elements having codeword sizes 1, 2, etc. For the two codes we are describing, these sequences were very similar ($\langle 1,1,1,2\rangle$ for the first and $\langle 1,1,1,1,2\rangle$ for the second). Other elements that were merged are the letters `a`, `i` and `e` (vowels); the words `hall␣`, `have␣`, `␣have␣`, `␣will␣`, `␣be␣`, `not␣` and `␣not␣` (all terminating in space); and the words `of`, `And`, `and`, `that`, `shall` and `␣shall` (all full words in English, and thus generally followed by space, comma, etc.).

5. Conclusion

In this paper we examined a novel application of the concept of clustering — methods for partitioning a set into subsets of like items. Our compression task involved creating large alphabets and predicting symbol occurrence by means of a Markov model. We used a greedy clustering heuristic to group states in a manner that greatly reduced the cost of storing the model. This permitted a choice from a range of compression / overhead tradeoffs. Selecting the best one for our application generally yielded significant savings in model storage, at the cost of a negligible loss in compression efficiency.

Acknowledgment: We would like to thank Nili Schwartz for doing a large part of the programming.

References

[1] Abramowitz M., & Stegun I.A. (1965). *Handbook of Mathematical Functions.* New York: Dover Publishing.

[2] Bell T.C., Cleary J.G., & Witten I.H. (1990). *Text Compression.* Englewood Cliffs, NJ: Prentice Hall.

[3] Bookstein A. (1995). New Directions in Clustering. Paper presented at the *Fourth Annual Symposium on Document Analysis and Information Retrieval*, Las Vegas, Nevada.

[4] Bookstein A., & Klein S.T. (1990). Compression, Information Theory and Grammars: A Unified Approach. *ACM Trans. on Inf. Sys, 8,* 27–49.

[5] Bookstein A., & Klein S.T. (1993). Is Huffman coding dead? *Computing*, 50, 279–296.

[6] Cleary J.G., & Witten I.H. (1984). Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Trans. on Comm,* COM–32, 396–442.

[7] Cormack, G. V. (1985). Data Compression on a Database System. *Commnications of the ACM*, 28(12), 1336–1342.

[8] Cormen, T., Leiserson, C.E., & Rivest, R (1990). *Algorithms.* Cambridge, MA: MIT Press.

[9] Equitz W.H. (1989). A new vector quantization clustering algorithm. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 37, 1568–1575.

[10] Jain A.K., & Dubes R.C. (1988). *Algorithms For Clustering Data.* Englewood Cliffs, NJ: Prentice Hall.

[11] Gilbert E.N., & Moore E.F. (1959). Variable-length Binary Encodings. *The Bell System Technical Journal*, 38, 933–968.

[12] Huffman D. (1952). A method for the Construction of Minimum Redundancy Codes. *Proc. of the IRE*, 40, 1098–1101.

[13] Knuth D.E. (1973). *The Art of Computer Programming, Vol I, Fundamental algorithms.* Reading, Mass: Addison-Wesley.

[14] Kullback S. (1959). *Information Theory and Statistics* New York: John Wiley and Sons.

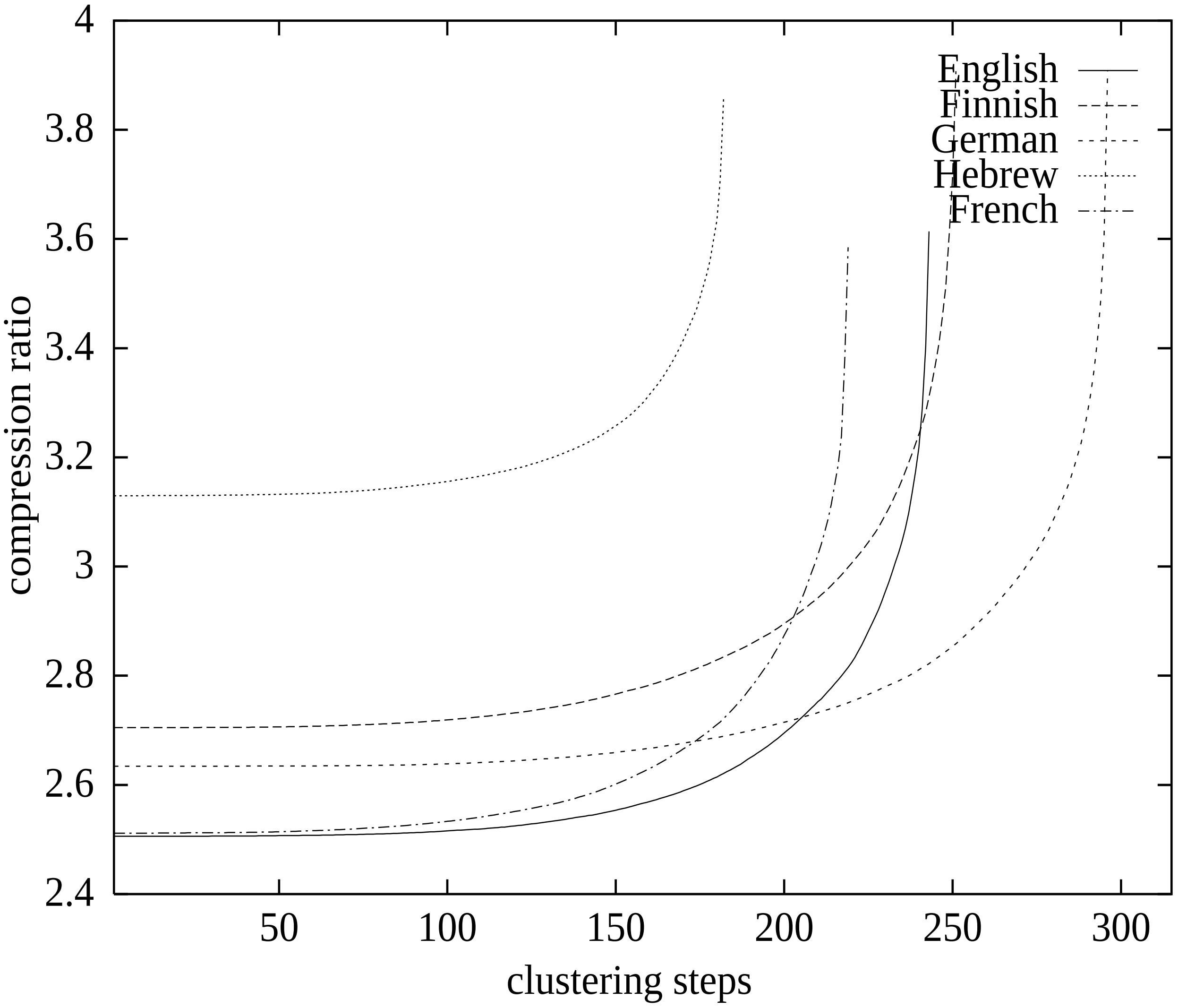
[15] Rissanen J.J., & Langdon G.G. (1981). Universal Modeling and Coding. *IEEE Trans. on Inf. Th,* IT–27, 12–23.

[16] Shannon C.E. (1948). A Mathematical Theory of Communication. *Bell System Tech. J.*, 27, pp. 379–423, 623–656.

[17] Welch T.A. (1984). A Technique for High-Performance Data Compression. *IEEE Computer*, 17, 8–19.

[18] Witten I.H., Moffat A., & Bell T.C. (1994). *Managing Gigabytes: Compressing and Indexing Documents and Images.* New York: Van Nostrand Reinhold.

[19] Ziv J., & Lempel A. (1977). A Universal Algorithm for Sequential Data Compression. *IEEE Trans. on Inf. Th,* IT–23, 337–343.



compression ratio
4
3.8
3.6
3.4
3.2
3
2.8
2.6
2.4
50
100
150
200
250
300
clustering steps
English
Finnish
German
Hebrew
French

Figure 1: Compression tradeoffs

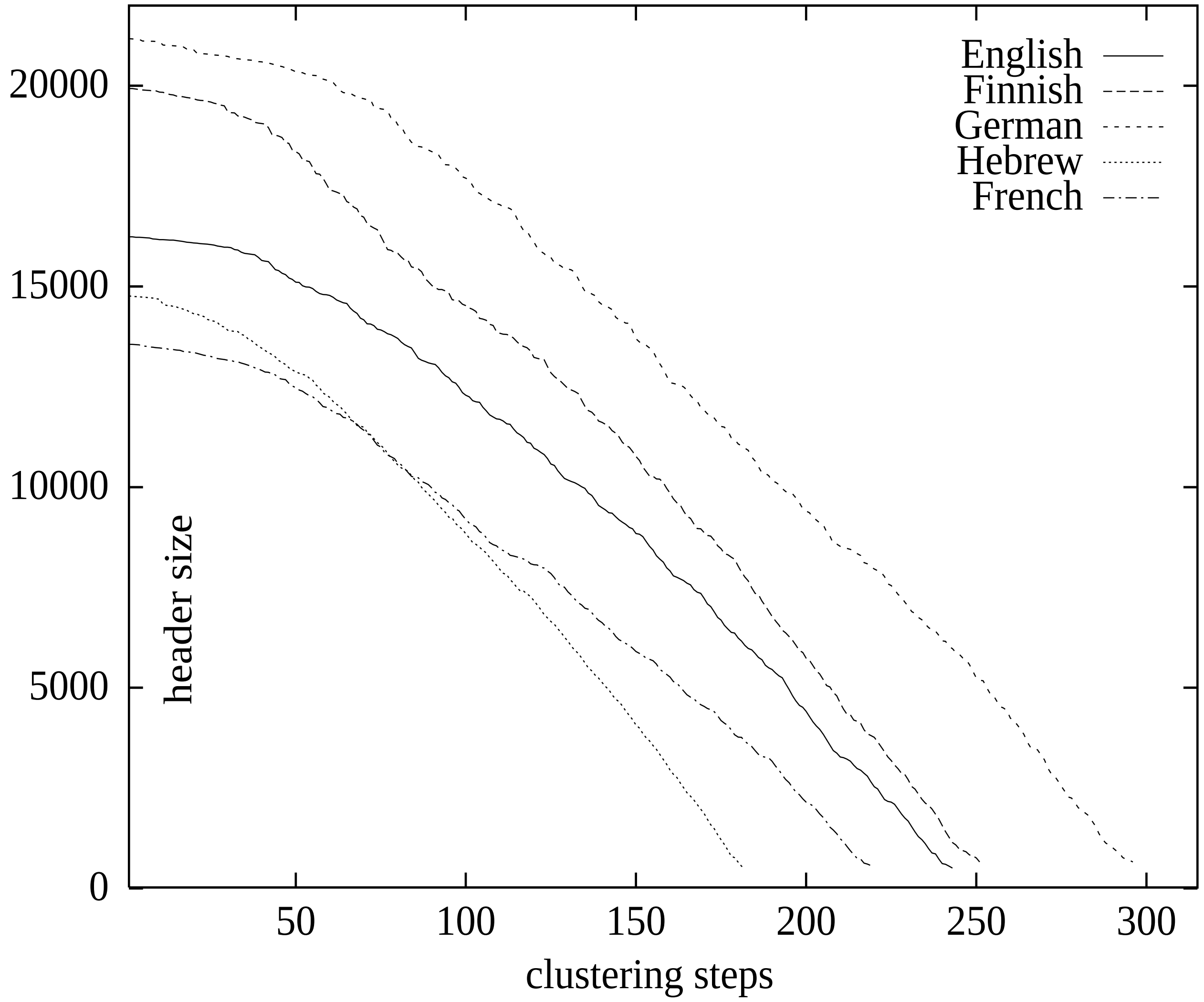

English
Finnish
German
Hebrew
French
header size
0
5000
10000
15000
20000
50
100
150
200
250
300
clustering steps

Figure 2: Header size

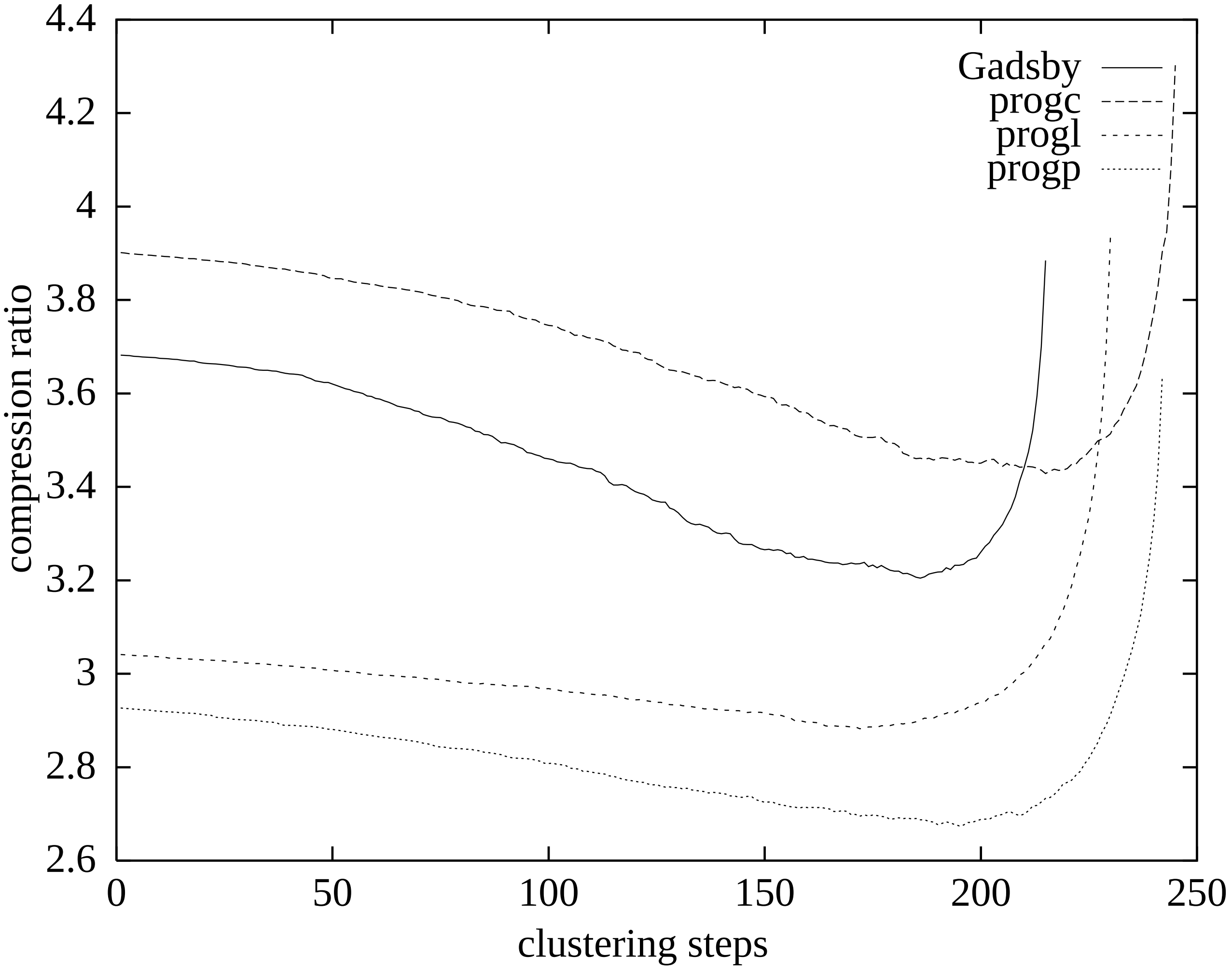

Gadsby
progc
progl
progp
compression ratio
clustering steps
4.4
4.2
4
3.8
3.6
3.4
3.2
3
2.8
2.6
0
50
100
150
200
250

Figure 3: Compression tradeoffs