Text Compression Via Alphabet Re-Representation [1] [2] [3]

PHILIP M. LONG

Department of Information Systems and Computer Science

National University of Singapore, Singapore 119260, Republic of Singapore.

APOSTOL I. NATSEV, JEFFREY S. VITTER

Department of Computer Science, Duke University, Durham, NC 27708.

[1]An extended abstract of this work appears in *Proceedings of IEEE Data Compression Conference (DCC '97)*, March 1997.

[2]Acknowledgments: The authors would like to thank J. Schmidhuber and S. Heil for providing the newspaper training data, and the reviewers for their helpful comments and suggestions about the paper. P. Long is supported by National University of Singapore Academic Research Fund Grant RP960625. Some of the work reported in this paper was done while this author was at Duke University supported by ONR grant N00014–94–1–0938 and AFOSR grant F49620–92–J0515. A. Natsev and J. Vitter are supported in part by Air Force Office of Scientific Research, Air Force Material Command, USAF, under grants F49620–92–J–0515 and F49620–94–1–0217, and by an associate membership of J. Vitter in CESDIS.

[3]Requests for reprints should be sent to Jeffrey S. Vitter, Department of Computer Science, Duke University, Durham, NC 27708. Email: jsv@cs.duke.edu, tel.: (919) 660-6548, fax: (919) 660-6502.

Text Compression Via Alphabet Re-Representation

Abstract—*This paper introduces the notion of alphabet re-representation in the context of text compression. We consider re-representing the alphabet so that a representation of a character reflects its properties as a predictor of future text. This enables us to use an estimator from a restricted class to map contexts to predictions of upcoming characters. We describe an algorithm that uses this idea in conjunction with neural networks. The performance of our implementation is compared to other compression methods, such as UNIX compress, gzip, PPMC, and an alternative neural network approach.*

Keywords—Text compression, alphabet re-representation, neural networks, machine learning, over-fitting.

1 Introduction

Data compression is an important field of Computer Science mainly because of the reduced data communication and storage costs it achieves. Given the continued increase in the amount of data that needs to be transferred and/or archived nowadays, the importance of data compression is unlikely to diminish in the foreseeable future. On the contrary, the great variety of data that allows for compression leads to the discovery of many new techniques specifically tailored to one type of data or another. The goal of this paper is to concentrate on text compression, in particular, by pursuing an innovative and promising avenue for improving the compressibility of text. The main idea is that the standard ASCII representation is not necessarily the optimal way to order characters, and changing the representation of the alphabet may prove beneficial to various text processing tasks, including compression.

Current methods do not consider geometric information for prediction purposes. For instance, both letters p and s tend to predict the letter h (there are many words containing the sequences ph and sh). The letter q, however, tends to precede the letter u rather than h, and yet in the English alphabet (and in the ASCII code tables) p is closer to q than it is to s. Intuitively, any given character is endowed with a number of "features" that affect what might be coming next. Examples of features include whether the character is alphabetic, small or capital, and whether it is a consonant. This leads us to build a (multidimensional) re-representation of the ASCII characters. One property that is intuitively desirable of such a re-representation is that characters that tend to precede the same characters are close under the new representation. That property would ensure that small changes in the contexts lead to small changes in the probability distributions, and we can restrict ourselves to considering only the class of such smooth transitions. Since neural networks are known to be good at learning and generalization of smooth data, they may be able to make predictions with higher confidence than the traditional methods. The algorithm that we propose, named *Prediction by Smooth Mapping* or simply *PSM,* is briefly summarized as follows: we first build the above-mentioned alphabet re-representation and use it in conjunction with a particular neural network model. At each step, the input context is re-represented and passed to the neural network which outputs the probability distribution of the next character given the particular context. The next character is then arithmetic-coded using the predicted probability distribution.

The rest of the paper is organized as follows: we begin by providing some minimal background information on text compression methods, and in Section 3 we describe the motivation behind this project, and interpret it as a pattern recognition problem. In Section 4 we combine the notion of re-representation with neural networks, and outline our proposed algorithm, while Section 5 contains the implementation details of the algorithm. We then proceed by examining the particular neural network update rule that we use, and form the theoretical basis of the investigated text compression scheme. In Sections 7–9 we go on to discuss parameter estimation issues, alternatives, some possible applications, and experimental results. We finally conclude with a summary and suggestions for future work and further improvements.

2 Compression background

The majority of text compression techniques fall into two main categories: Lempel-Ziv (LZ) methods (Ziv & Lempel, 1977, 1978; Welch, 1984) and Prediction by Partial Matching (PPM) methods (Bell, Cleary, & Witten, 1990; Cleary & Witten, 1984). The PPM methods are the more successful methods in terms of compression efficiency. They generally gather (perhaps dynamically) some statistical information about the probability distribution source (the file to be encoded). This information is then used to estimate the probability distribution of the next character given what the previous n characters were, and finally the probability estimate is used in conjunction with an entropy coder, usually arithmetic coding (Moffat, Neal, & Witten, 1995), to encode the next character. Typically, the probability of each character is approximated by the fraction of times this character occured after the particular context of length n (the theoretical basis for this is given in Section 3). The PPM methods are also called context methods and character prediction methods because they basically predict the probability distribution of the next character in any given context. For a detailed description of PPM methods consult (Bell et al., 1990; Cleary & Witten, 1984).

In general, it is an information-theoretical result that the minimum number of bits required to encode a character c with a probability p of occuring next, is equal to $\log 1/p$ (Gallagher, 1968). It is also known that arithmetic coding is asymptotically optimal in this respect so the main reason for PPM methods not to achieve maximum compression is the prediction error in the modeling of the probability distribution. In light of that, we focus our attention on the issue of modeling the distribution so that the prediction error is minimal.

3 Motivation and problem statement

The compression problem of coding a character α_i in a context p can be expressed as a Pattern Recognition (PR) problem, where the contexts are treated as patterns that form classes relative to the letters of a given alphabet. This affinity between Pattern Recognition and text compression suggests that native PR approaches can be used for text compression purposes. For example, similarity to syntactic (or grammar) PR approaches can be found in Lempel-Ziv methods and language tag-based compression methods that try to exploit lexicographic redundancies due to the underlying higher-order grammatical structure in natural languages. The PPM methods, on the other hand, are a clear example of statistical PR techniques and use the well-known Bayesian estimate for the relevant character probabilities:

$$Pr(\alpha_i \mid p) \quad = \quad \frac{Pr(\alpha_i, p)}{Pr(p)} = \frac{Pr(p \mid \alpha_i)Pr(\alpha_i)}{Pr(p)} = \frac{N_p^{\alpha_i}}{N_p},$$

where $N_p^{\alpha_i}$ is the number of times the context p was followed by the letter α_i in the string, and N_p is the total number of occurrences of pattern p.

Alternative, and not-so-conventional, PR approaches (such as neural networks and fuzzy logic) have already been applied to image compression (Cottrell, Munro, & Zipser, 1989), and recently to text compression as well (Schmidhuber & Heil, 1996). The ability of neural networks, in particular, to learn smooth data suggests that they may be able to do a better job in modeling the character probabilities by employing some form of "intelligent" learning. One of the main challenges of neural network applications, however, is the problem of over-fitting, or the use of too many degrees of freedom in the underlying model. If the model size is unnecessarily large, it eventually leads to memorizing all of the details in the training data, while being unable to extract common patterns from it and generalize well outside of the sample set. In terms of text compression, this relates to the smoothness of the mapping from contexts to their character probability distributions for coding purposes. If this mapping is smooth, it would be easy to model with a few model parameters, and a neural network would be able to make good character predictions for compression purposes. We therefore propose to compute an alphabet re-representation as described previously, that would ensure smooth mapping between contexts and probability distributions. Thus, by imposing structure through re-representation, the problem of over-fitting could be greatly reduced or eliminated, even if contexts of longer length are used. For more information on modern Pattern Recognition approaches, the reader is referred to (Schalcoff, 1992), and for further discussion on its relation to text compression, see (Natsev, 1997). The problem of over-fitting is briefly addressed in Section 6, and also in (Natsev, 1997; Weigend, Rumelhart, & Huberman, 1991; Hochreiter & Schmidhuber, 1997).

4 Algorithm PSM

The proposed algorithm, which we have named PSM (Prediction by Smooth Mapping), is summarized below:

1. *Compute an alphabet re-representation (a pre-processing step done off-line).* This step doesn't have a match in PPM-like methods, and the purpose of its introduction here is to facilitate Step 2.

2. *Model the probability distribution of the next character given a certain context.* This step is similar to the probability modeling step of PPM methods, the only difference being that it is done by a neural network in an attempt to efficiently capture the

smoothness promised by the previous step.

3. *Use a statistical (arithmetic) coder to get the final output stream.* This step is exactly the same as in PPM. Special care is taken when the neural network's prediction is very poor and the probability of the next character is practically zero. If that happens, a special escape symbol is transmitted and the next character is encoded with a uniform distribution.

We propose to construct the re-representation while taking into account its effect on the resulting compression algorithm. We achieve this by viewing the re-representation as a neural network layer (closest to the inputs) from contexts to probability distributions. By specifying the right error-criteria to the neural network we can make sure that we have optimized everything with respect to our primary goal—minimizing the number of output bits required to encode a file. In Section 6, we derive a training update rule that performs gradient ascent on the log-likelihood.

The first two steps are therefore combined into a single feed-forward multi-layer back-propagation neural network of a particular architecture (see Section 5 for the specifics). The first layer of weights corresponds to the re-representation, while the rest of the network corresponds to the probability modeling part. The network is trained off-line over a large set of training data, and then the weights that correspond to the new re-representation are fixed. After training, we operate on the assumption that a good re-representation is already computed and we thus have a smooth mapping from the context domain to the probability distribution domain. In this sense, we can treat the first layer of the network as a re-representation layer that pre-processes the input (essentially via a table look-up) before it passes it on to the rest of the network. Once this is done, though, we can predict the probability distributions more accurately. Since Step 1 is done only once—during training—we only need to perform Steps 2 and 3 when we actually process a file on-line.

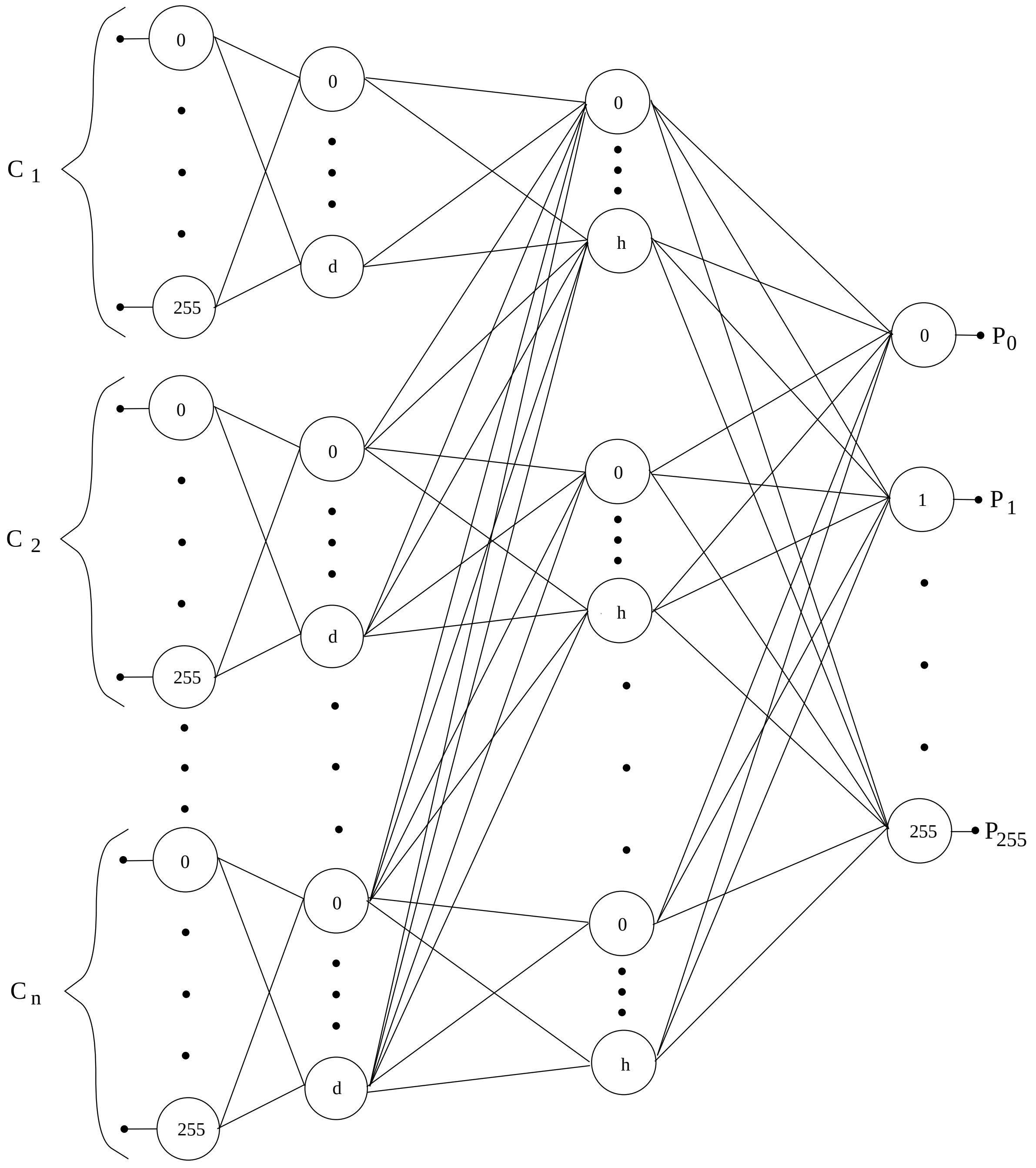


Figure 1: Architecture of the neural network demonstrating weight partitioning (between 1st and 2nd layers) and blending (between 2nd and 3rd layers). Parameters include: context size n, feature space dimensionality d, and hidden chunk factor h.

5 Algorithm implementation

Our network takes as an input the current context, and outputs a number between 0 and 1 for each letter in the alphabet. These numbers are normalized so that they add up to 1, and so the result is the probability distribution of the next character given the particular context. Then, a statistical coder, such as arithmetic coding, uses all the probabilities to encode the actual character that appears next, and the error of the network is propagated back to the lower levels (the output corresponding to the actual encoded character gets 1 as a target value to be propagated, and all other outputs receive a target of 0). Since the decoder has access to the same information the encoder uses for encoding (the context has already been decoded), it can decode the next character successfully, and update the weights of its equivalent neural net. Therefore, the network need not be transmitted.

Figure 1 illustrates the particular architecture of our neural network. The input layer consists of n groups of 256 input nodes each (the alphabet size can actually vary), where n is the context size. Each group of 256 nodes encodes exactly one character—the one that appears some k places back from the current point in the input string. For example, suppose we have the string *testing* as input, and we have just read the second t. If the context size of our model is three, the context will be est, and each of the characters e, s, and t will be encoded by a separate group of 256 input nodes. In the case of the letter e for example, all but the 101st node will be zero, the 101st node being 1 because the ASCII code of the character e is 101. The input consists of $256n$ nodes, all but n of which are 0.

Similarly to the input layer, the first hidden layer consists of n groups as well but each one has d nodes, where the parameter d denotes the dimensionality of the feature space. The weights between the first two layers correspond to the re-representation that we mentioned in previous sections. Note that only a single node will be non-zero in each group of 256 input nodes (namely the one corresponding to the ASCII code of the character encoded by that group). Therefore, the activation values in the kth group of nodes in the first hidden layer actually corresponds to the d-dimensional embedding of the character that appears k places in the text before the character that we are trying to predict. Every character in the context is thus mapped to a d-dimensional vector, where each coefficient reflects the extent to which that character has a certain hidden feature. Note also that the characters have different representations according to how far away they are from the character to be predicted. Thus, this architecture allows us to capture the different statistical properties of characters as predictors of characters different distances further in the file. After the network is trained, all weights (i.e., character embeddings) between the first two layers are fixed so that retrieving a character's re-representation is essentially done via a look-up table.

The rest of the network is similar to traditional 3-layer networks. The presence of an additional hidden layer is warranted by the necessity to be able to learn non-linearly-separable functions, and the lack of further layers is motivated by our desire to restrict the network size so that it can generalize better. The second hidden layer, then, consists again of n groups, each containing h nodes and corresponding to a letter in the context. This time, however, the separate groups are not mutually independent but are rather combined in a blending fashion so that the ith group is fully interconnected with all but the first $i-1$ groups in the previous layer. This architecture is motivated by the ability to separate the predictions that are generated by a higher order context model from those generated by a lower order context model. This way, the nodes in the ith group of nodes in the second hidden layer have the full information of an order-i model that uses contexts of size up to i. The final probabilities are combined in the output layer so as to allow the network to weigh the different models in a different way. The parameter h simply reflects the complexity required to capture the important characteristics of an order-k model. This "blending" architecture resembles other statistical approaches such as PPM with blending, where the different models give separate estimates, which are then combined for the final prediction in a weighted blending fashion. Finally, the output layer, as we mentioned before, consists of 256 nodes, each corresponding to the probability that its character will be the next one in the string. The size of the output layer is fixed to 256 nodes.

The weight partitioning in the first layer and blending in the second layer reduces the time and space complexity of the network significantly and allows for better generalization.

Although the assumption of complete or partial independence within the initial weight layers restricts the network's architecture and thus may hinder the learning process, it generally does not hurt, and in some cases even helps, the generalization capabilities of the network at a greatly reduced computational cost in terms of both space and time requirements.

6 Neural network update

In this section we consider the design of our network and the derivation of the neural network update rule for our application. We use a 4-layer feed-forward back-propagation neural network with a sigmoidal activation function for all hidden nodes, and a normalized exponential activation function for the outputs. The cost function is selected to maximize the log-likelihood of the data given the network, and we argue that our specific choice of a cost function is optimal for compression purposes.

The classic references for back-propagation neural networks are (Werbos, 1974; Rumelhart, Hinton, & Williams, 1986), but for a more thorough treatment of design issues and factors for model selection consult (Chauvin & Rumelhart, 1995). Let $\mathcal{D} = \langle \vec{x}_i, \vec{t}_i \rangle$ denote the observed data where $\vec{x}_i$ is the ith input vector (i.e., the context), and $\vec{t}_i$ is the target vector (i.e., the character which appeared next). Let $\mathcal{N}$ be the neural network that is designed to learn $\mathcal{D}$. The usual Bayesian motivation leads us to maximize

$$\ln P(\mathcal{D} \mid \mathcal{N}) \;=\; \ln\left(\prod_i P(\langle \vec{x}_i, \vec{t}_i \rangle \mid \mathcal{N})\right) = \sum_i \ln P(\vec{t}_i \mid \vec{x}_i \wedge \mathcal{N}) + \sum_i \ln P(\vec{x}_i). \qquad (1)$$

Each $P(\vec{x}_i)$ does not depend on the network choice and can therefore be disregarded for our purposes. At this point, in order to get an expression for an optimal cost function we need to make some assumption about the type of the probability distribution $P(\vec{t}_i \mid \vec{x}_i \wedge \mathcal{N})$. In (Chauvin & Rumelhart, 1995) Rumelhart, Durbin, Golden, and Chauvin consider the general family of distributions

$$P(\vec{t} \mid \vec{x} \wedge \mathcal{N}) = \exp\left(\sum_i \frac{(t_i\theta - B(\theta)) + C(\vec{t}\phi)}{A(\phi)}\right), \qquad (2)$$

where θ is related to the mean of the distribution, ϕ is the overall variance, and the functions $A(\,)$, $B(\,)$, and $C(\,)$ are specified individually for each member of the family of distributions.

As it turns out, for any such distribution, we can derive a cost function $\mathcal{E}$ that maximizes the log-likelihood term given by equation (1) so that its gradient with respect to the net input at output node j is given by $\frac{\partial \mathcal{E}}{\partial \text{net}_j} \propto \frac{t_j - a_j}{\text{var}(a_j)}$. We can further choose an activation function for the output nodes that cancels the variance term in the above expression so that the gradient is always proportional only to the difference between the target values and the actual output values of the network. The multinomial distribution is a special case of the above family that is a generalization of the binomial distribution. The corresponding energy function is given by $\mathcal{E}_{multinomial} = \sum_i \sum_j t_{ij} \ln a_{ij}$, where index i ranges over the observations, and index j ranges over the output nodes. The appropriate activation function that cancels the variance term in the gradient is then the normalized exponential function. In (Chauvin & Rumelhart, 1995), the authors argued that the multinomial case is most suitable when the network is supposed to make 1-out-of-n classification. In that case the output is treated as a probability distribution, and the ith output node corresponds to the probability that the pattern goes to the ith class. Since this is exactly the case in data compression applications (we are trying to predict one letter from an alphabet of fixed size), for our purposes we use the energy function given by $\mathcal{E}_{multinomial}$ along with the normalized exponential activation function for the output nodes: $a_i = e^{\text{net}_i} / \sum_j e^{\text{net}_j}$, where the index j ranges over the output nodes.

If we interpret the desired and the actual outputs as probability distributions over the fixed alphabet, then the multinomial energy term is negatively proportional to the number of bits we would spend to encode the input string. In other words, if character α_j has a probability t_{ij} of occurring at the ith position in the text, and the neural network's estimate of that true probability is a_{ij}, then the expected number of bits that need to be transmitted to uniquely encode the ith character is equal to $\sum_j t_{ij} \log \frac{1}{a_{ij}} = -\sum_j t_{ij} \log a_{ij}$. When we sum over the position i in the text, we obtain the total number of bits needed to encode the input string. As we can see, the expression for the energy function we selected describes exactly that quantity, up to a constant factor of $-\frac{1}{\ln 2}$. Therefore, by maximizing the chosen energy function, we are not only maximizing the log-likelihood of the data given the network but we are also directly minimizing the total number of bits required to encode the data.

Having chosen the architecture, the cost function, and the activation functions we move on to the particular formulas used for updating the weights of the network during training. We use gradient descent, setting

$$w_{ij}^{new} = w_{ij} + \eta \frac{\partial \mathcal{E}}{\partial w_{ij}} = w_{ij} + \eta a_i \delta_j, \tag{3}$$

where w_{ij} is the weight between nodes i and j, η is the learning rate, and δ_j denotes the gradient with respect to the net input at node j. When j is an output node, we use $\delta_j = t_j - a_j$, and when j is a hidden node, we set $\delta_j = a_j(1-a_j)\sum_k \delta_k w_{jk}$, where k ranges over nodes in the next layer, and t_j and a_j denote the target and the actual output values at node j. For space considerations, we have omitted the derivation of the above formulas because it resembles the derivation of standard backpropagation update rules (Rumelhart et al., 1986), and can be found in its entirety in (Natsev, 1997).

7 Setting parameters

Next we shall discuss some of the issues involving parameter estimation and the techniques we have employed to automate that process as much as possible. The proposed method has an increased number of parameters compared to the conventional neural network approaches because we have tried to improve learning and generalization by imposing a particular structure in the model. As a result, the task of estimating the optimal values of the parameters becomes increasingly important. First of all, the learning rate used in the gradient descent weight-update is crucial during the training phase. Furthermore, since in data compression applications the amount of training data for the network is relatively large, the main factor in determining the learning rate was not the optimal rate of convergence. This decision is motivated by the fact that, even with a small learning rate, the network will eventually learn its function provided that it is trained sufficiently long. Our main concern then was to make sure that the learning rate is not too big so as to result in oscillation on the gradient curve after extensive training. We have therefore used a decaying learning rate which was inversely proportional to the square root of the iteration number. This is a somewhat standard practice that semi-automates the process of learning rate adjustment, and we have adopted it for our application as well.

The other crucial parameters that affect the performance of the network are the ones that control its size, namely the context length, n, the feature space dimensionality, d, and the "hidden chunk factor", h. For the context size we have used the values 5 and 10 because studies have shown that contexts of size between those numbers are the ones most heavily used for prediction purposes. The other two parameters, d and h, were estimated adaptively by monitoring the performance of the network on the training set and increasing the values of the two parameters whenever it seemed that the network converged to some local minimum (i.e., the improvement over the past several iterations was not significant). On the one hand, too small values will result in a network that is not big enough to learn the function we are modeling. On the other hand, too large values will prompt too many degrees of freedom which will probably improve the learning step during training but will certainly harm the generalization capabilities of the network (i.e., there will be over-fitting). Therefore, estimating the right network size is of paramount importance for such applications. The neural net community has proposed several strategies for dealing with the over-fitting problem, and they are mainly of two types:

- learn conservatively, and expand the model only if you have to.
- expand freely until you get the best fit on the training data, and then prune the network to improve generalization.

We have adopted the first approach for dealing with over-fitting: first, by imposing *a priori* structure on the network through blending and partitioning; and second, by doing adaptive size adjustments through manipulating the d and h parameters during training. Variations of the second approach include monitoring the usage of hidden nodes (i.e., how many times a hidden node has had a high activation vs. a low activation), and adding or deleting nodes based on the load of each hidden node (i.e., split when heavy load, prune when light load) (Thodberg, 1991). Other techniques in the same class include weight decay/elimination (Weigend et al., 1991), and the more general Flat Minimum Search (Hochreiter & Schmidhuber, 1997).

8 Alternatives and applications

One alternative to our implementation is to have a separate pre-processing step for training the network on the file to be encoded. Instead of having an adaptive neural net training and coding at the same time, we would have an additional pass (or maybe several passes) over the input file, and then the network would be transmitted to the decoder before any encoding is begun. Both encoder and decoder will use the fully trained network without modifying the weights after coding has begun. This semi-adaptive approach can perhaps significantly improve the prediction performance but will also impose some compression overhead because of the need to transmit the final weights. This variant will be feasible only for very large files so that the network size is relatively small compared to the rest of the output. Also, the fully-adaptive version will exhibit better flexibility in terms of capturing local statistical properties of the file.

A slightly modified, fully-static, approach would be to use the same network for all files without re-training it for each file either off- or on-line. This will probably decrease performance slightly but will greatly reduce both the computational overhead of updating all weights during compression/decompression (in the fully-adaptive version), and the compression overhead of transmitting the network for each file (in the semi-adaptive version).

In either case, though, a static or a semi-adaptive decoder will not have to do any training, and will therefore be much faster compared to the adaptive version. From a practical point of view, this could be very useful in applications where only fast decoding is required (i.e., data is stored once and accessed many times). There are many data bases and information retrieval systems (such as the World Wide Web) for which compressed data is kept in a centralized data bank, and each time the data is referenced, it is decompressed by the system or the user. The approach we have undertaken is a compromise between these alternatives in that it combines different methods for the two main parts of the neural network (the re-representation one and the probability modeling one). While the re-representation part uses a fully static off-line training approach, the rest of the network is trained adaptively during the on-line processing of each file. This is a reasonable compromise because the re-representation part makes up the bulk of the whole network and so it makes sense to train it only once and fix it for future use. On the other hand, the prediction part is not that big, and it is better to train it dynamically so that we could make full use of the local properties of the text. In general, there is a trade-off between speed and compression performance, and the final selection of a neural network model depends greatly on the purpose of its application.

Another factor that may influence the network architecture choice is the potential need for specifying *a priori* constraints about the neural network. For example, weight decay is a technique that forces the final weights of the network to be small and centered around a mean of zero. Another constraint that we might want to add is to have as simple a network as possible. This idea is the basis of weight elimination approaches, and is inspired by the notion of Occam's razor, which states that the complexity of the function beforehand should be as simple as possible. A third way of imposing some kind of desirable *a priori* structure on the neural network is to have weight symmetries. All these constraints might be useful, for example, in static training applications, where knowing some of the network's properties may allow us to save bits when transmitting the network to the decoder. For examples on specifying *a priori* constraints when designing a neural network see (Chauvin & Rumelhart, 1995).

Neural networks can be parallelized in hardware, or even methods other than neural networks can certainly be considered, in order to speed up computation while utilizing the benefits a re-representation can offer. We have previously investigated several different algorithms for computing a re-representation, as well as its use in conjunction with other compression methods such as block-sorting (Wheeler & Burrows, 1994). Re-representation approaches we have considered include simulated annealing, gradient descent, multi-dimensional scaling, spring optimization via a force-directed approach, and Traveling Salesman Problem approximation heuristics (such as the greedy approach and Lin-Kernighan's 2-OPT heuristic) (Faloutsos & Lin, ; Murtagh, 1983; Lawler, Lenstra, Rinnooy, & Shmoys, 1985).

9 Results and discussion

We have run two sets of experiments to test our approach. The first set is identical to the experiments reported in (Schmidhuber & Heil, 1996), where the authors propose a similar neural network that consists of the same input/output structure but uses only one hidden layer. As reported in (Schmidhuber & Heil, 1996), the number of hidden nodes used in their single hidden layer is 440, the context size is 5, and the alphabet size is 80. The matching configuration that we used had an alphabet size of 256 so that it can handle even binary files but all characters that were predicted with essentially a zero probability were combined together and treated as a single escape character. This way, the enlarged alphabet size does not hurt prediction but allows generality at the cost of a somewhat higher complexity. We used the same context size of five symbols, and the other parameters were eventually set to $d = 50$, and $h = 30$ (they were estimated adaptively during the course of training). Thus, the parameter choice resulted in a network with 1280 input nodes, 250 nodes in the first hidden layer, 150 nodes in the second, and finally 256 output nodes. However, since the weights between the first two layers correspond to the re-representation and are fixed during on-line file processing (i.e., they are retrieved as a table look up), the network used for actual compression was essentially a three-layer network with 250 input nodes, 150 hidden nodes, and 256 output nodes. The training time for PSM was very slow, though the complexity of the network is lower than that of the alternative neural network approach (Schmidhuber & Heil, 1996). For detailed complexity analysis and comparisons with that approach, the reader is referred to (Natsev, 1997). The learning rate used in the experiments reported in (Schmidhuber & Heil, 1996) was fixed to 0.2 but for our approach we used a decaying learning rate which started off at 0.2. The training set consisted of 40 articles from the

German newspaper *Münchner Merkur*. Test set 1 consisted of additional 20 articles from the same newspaper, and test set 2 consisted of 10 articles from a different newspaper, *Frankenpost*, on which the networks were not trained in advance. All of the files in both the training and the test sets were smaller than 20 kilobytes. The results of the experiment are shown in Table 1. *Pack* is the UNIX command that implements Huffman coding only. *Compress* and *gzip* are also UNIX commands that use Ziv-Lempel algorithms (Ziv & Lempel, 1977, 1978; Welch, 1984), and are often used in practice due in part to their computational efficiency. *PPMC* is the public domain implementation of that method with fixed context size 3. The $PPMC^{tr}$ method is just *trained PPMC* that attempts to use the statistics from the training set when compressing files from the test set. We have tried to simulate training for the *PPMC* method by concatenating the whole training set and prepending it to each test file to be compressed. The total compressed size of the training set is then subtracted from the compressed size of the modified test file in order to obtain the true compressed size of the test file when statistics from the training set are used by *PPMC*. To put it in another way, if T is the concatenated training set, and the function $size(f)$ returns the *PPMC*-compressed size of a given file f, then the results reported for the plain *PPMC* method are $avg_f(size(f))$, while the results reported for $PPMC^{tr}$ are $avg_f(size(Tf) - size(T))$.

As seen from the table, our approach outperforms all other competitors (including plain *PPMC*), except $PPMC^{tr}$. The compression improvement over the other methods ranges anywhere from about 15% (for *PPMC* and the other neural network approach) to more than 50% (for *pack* and *compress*). $PPMC^{tr}$ performs roughly the same as *PSM* on one test set and about 6% better on the other test set. We should note, however, that the purpose of this experiment was to compare *PSM* against the other neural network approach. We have therefore used parameter settings that are as close as possible to the ones reported in (Schmidhuber & Heil, 1996), and have tried to match the alternative neural network configuration. To that purpose, for example, we have only used a context size 5 in this experiment, even though a larger context size may have resulted in a better performance (as seen in the second set of experiments).

Method's Name	Average Compression Ratio (Variance)		
	Münchner Merkur	*Frankenpost*	*Jack London*
pack	1.74 (.0002)	1.67 (.0003)	1.78 (.0001)
compress	1.99 (.0014)	1.71 (.0036)	2.45 (.0060)
gzip -9	2.30 (.0033)	2.05 (.0097)	2.64 (.0049)
Other *NN*	2.72 (.0234)	2.20 (.0112)	—
PPMC	2.70 (.0069)	2.27 (.0131)	3.54 (.0984)
$PPMC^{tr}$	**3.27** (.0633)	2.60 (.0281)	3.33 (.0075)
PSM	3.09 (.0142)	**2.61** (.0047)	**3.56** (.0083)

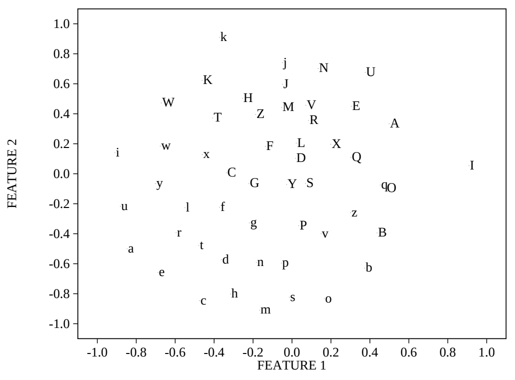
Table 1: Compression performance of various methods on 3 test sets consisting of newspaper articles from *Münchner Merkur* and *Frankenpost*, and of books by *Jack London*.

Since all the files in the previous experiment were relatively short (i.e., < 20 kilobytes), trained methods such as $PPMC^{tr}$, *PSM*, and *NN* (Schmidhuber & Heil, 1996) have a natural advantage over other methods that do not use any *a priori* information about the text source. That explains in part why *PSM* and $PPMC^{tr}$ outperform all other methods by a large margin. We therefore designed a second experiment that uses longer files. While the purpose of the first experiment was to compare our approach with the alternative neural net approach proposed in (Schmidhuber & Heil, 1996), in the second experiment we wanted to primarily test our method against *PPMC* at more reasonable file sizes. The training set consisted of three books by Jack London totaling a little over 1 megabyte (*Sea Wolf, White Fang,* and *Call Of The Wild*), and the test set consisted of three other books (*Son Of The Wolf, Iron Heel,* and *People Of The Abyss*) by the same same author of approximately the same size. As we expected, the plain *PPMC* method performed better than before on the longer files, apparently because it had enough data to compute more accurate statistics for prediction. Surprisingly, though, the $PPMC^{tr}$ method actually performed worse than the plain one—indicating perhaps that the complexity of the *PPMC* order-3 model is limited and using more training data for gathering of statistics eventually results in worse predictions.

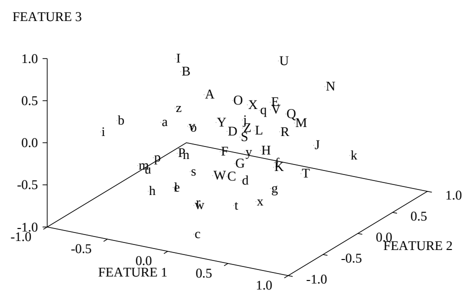
The performance degradation can be explained in part by the fact that the static "pre-training" of *PPMC* essentially prevented it from being more locally adaptive, which made it lose its compression edge on some of the files. This can be confirmed by the fact that the variance of the compression ratios was significantly lower for the $PPMC^{tr}$ method, which indicates that pre-training in general leads to more consistent and uniform results over the test files. In any case, our experiments with a context size 5 failed to outperform *PPMC* on these particular data sets, yielding compression ratios just short of 3.5. However, when we used a context length 10 and allowed the network to grow quite large (the size was adaptively adjusted to the final parameter estimates of $d = 45$ and $h = 40$), the performance of the network after 100 training iterations (with an initial learning rate of 0.3) actually surpassed that of both plain *PPMC* and Trained PPMC. The results from the second experiment are also given in Table 1 and they reveal that the performance of the *PSM* method is fairly consistent and very competitive in terms of compression efficiency.

In order to interpret the results in terms of the alphabet re-representation we have provided several plots of the re-representation obtained from training on the Jack London book set. We have used the weights in between the first two layers that correspond to the re-representation as described in Section 5, and we have shown only the 52 lower case and upper case characters from the English alphabet. The original dimensionality was 25 (that is, each character was mapped to a vector with 25 features). For visualization purposes, however, we have shown 2-dimensional and 3-dimensional embeddings which preserve the original "distances" between pairs of characters as close as possible. The algorithm used to compute the embeddings is a gradient descent iterative optimization technique that balances imaginary spring forces between character nodes so that the original distances are matched as well as possible.

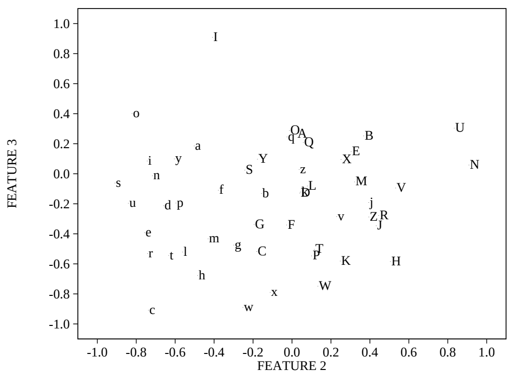
Figure 2(a) shows the two-dimensional embedding of an order-1 re-representation (that is, only one character is used as a context for predicting the next character). Similarly, Figure 2(b) shows the three-dimensional embedding of the same re-representation. Since the perspective view of the three-dimensional embedding does not provide sufficient detail



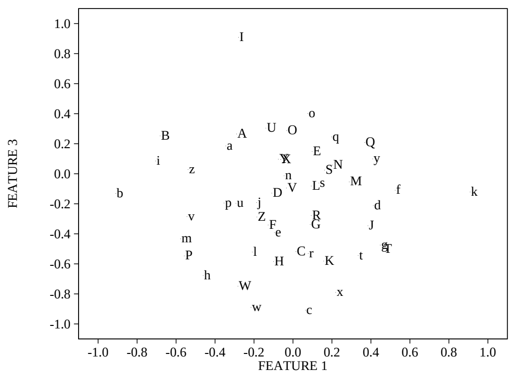
(a) 2-D embedding.



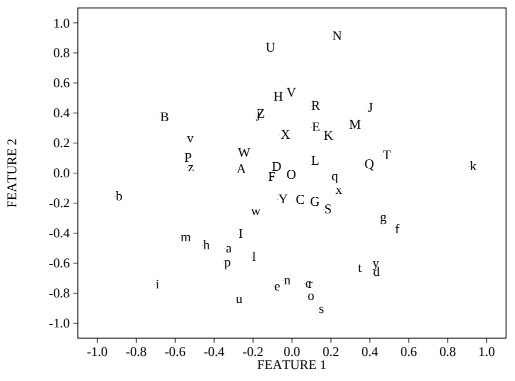
(b) 3-D embedding.



(c) X-axis projection.



(d) Y-axis projection.



(e) Z-axis projection.

Figure 2: Two- and three-dimensional embeddings of a sample 25-dimensional re-representation, along with 3 parameter plane projections of the 3-D embedding.

for interpretation purposes, we have also given the two-dimensional projections of that embedding onto the three parameter planes (Figure 2(c) shows the projection in the direction of the X-axis, Figure 2(d) represents the projection along the Y-axis, and Figure 2(e) gives the projection onto the XY-plane). We have also considered the embeddings of higher order re-representations but we found that the low-order ones are more expressive in terms of distinguishing between different characters and being able to identify implicit character features. For example, an order-5 re-representation looks clustered in a relatively small area, whereas the order-1 re-representation from Figure 2(a) is somewhat uniformly distributed in a larger region, and is thus more informative because it provides more detail about the character clusters.

Looking at Figures 2(a) and 2(c), for instance, we can clearly observe that the capital letters are all clustered towards the right hand side, while the small letters tend to be on the left hand side. A similar separation between small letters and capitals, but in the horizontal direction, occurs in Figure 2(e), and it can be seen even in the 3-dimensional graph on Figure 2(b). All this means that the neural network is able to capture the dissimilarities between small and capital letters due to the different statistical properties of characters at the beginning of a word and in the middle of a word. Another intuitive feature that the network was able to extract was the difference between vowels and consonants in their predictive properties. All of the above graphs have most of the vowels positioned on the outside of the clusters, while the consonants usually appear in the middle. Other features that may not be so intuitive to humans should also be present but are harder to identify and explain. In general, though, we can observe the formation of various clusters, which means that groups of characters are being treated in different ways within and between themselves. We should also note that the original re-representation has 25 features, not just 2 or 3, and therefore it captures much more of the statistical properties than we can see from the two-dimensional and three-dimensional visualizations.

Another way to measure the "goodness" of the computed re-representation is to compare the probability distributions of the next character under two contexts that are mapped

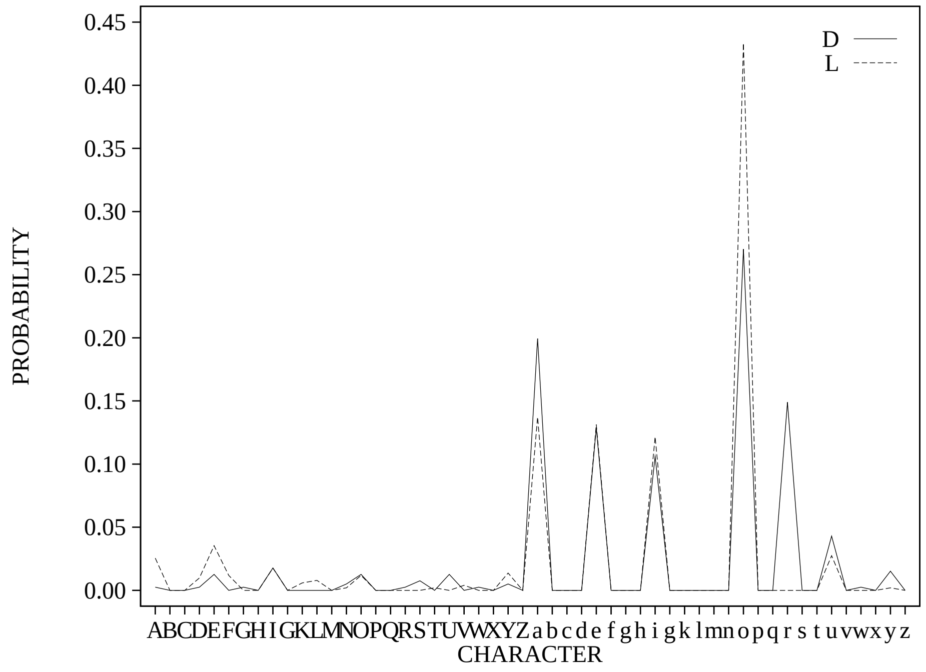


Figure 3: Probability distribution of the next character in contexts "D" and "L".

very close to each other by the new re-representation. To achieve that, we computed the distances between all pairs of characters under the new re-representation, and for each character we considered only its closest character. In other words, we picked the closest pairs of characters, and looked at the top few such pairs that corresponded to the smallest distances (with respect to the 25-dimensional Euclidean metric). The top two pairs turned out to be "D" vs. "L" (with Euclidean distance of 0.8512), and "F" vs. "G" (with Euclidean distance of 0.8793). The next-character distributions corresponding to each of those four letters as a context are shown in Figures 3 and 4, where the pairs of distributions are superimposed so that they can be compared more easily. When we compare the individual distributions in contexts "D" and "L" (Figure 3), we can easily see that the two distributions are similar in that they share the same peaks. Although some of the peaks correspond to somewhat different probabilities, the fact that they occur at the same places means that the two contexts tend to isolate and predict the same few characters with high probabilities.

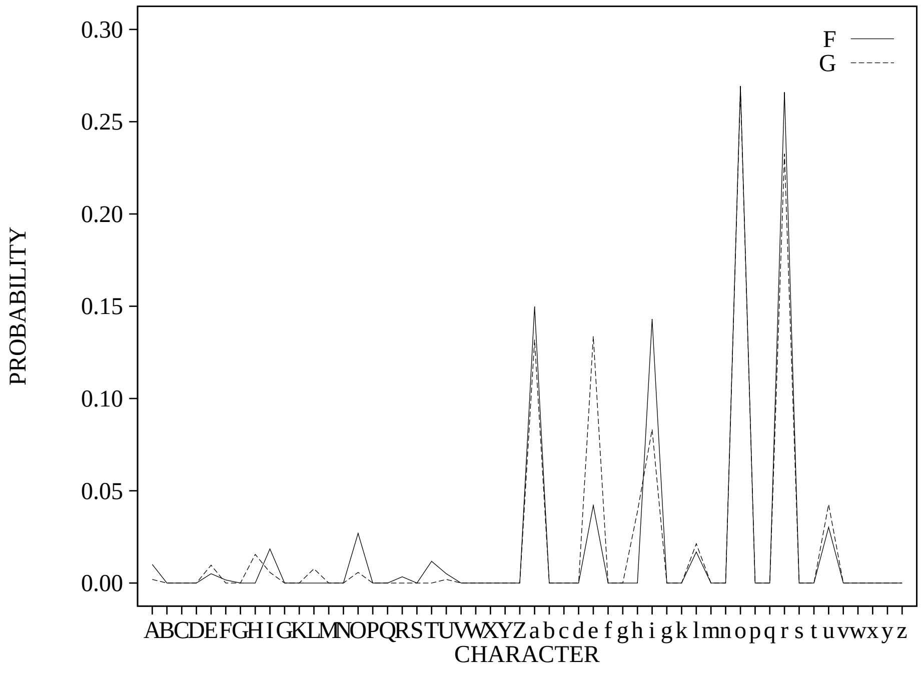


Figure 4: Probability distribution of the next character in contexts "F" and "G".

The same trend is even more evident in Figure 4, where the peaks are not only at the same positions but also correspond to approximately the same probability values. Given the close resemblance within the pairs of probability distributions corresponding to nearby contexts, we have a good reason to believe that the re-representation has been quite successful in accomplishing its task. This, by itself, is an achievement of independent interest that can be used in various applications for purposes other than text compression.

10 Conclusion and future work

The main contribution of this project is the introduction of a new re-representation approach to the probability modeling step in data compression systems. The proposed technique is an initial attempt to gather semantic information about the geometric structure of text and to use it intelligently through the locality principle and neural networks.

The idea of imposing structure through re-representation proves advantageous in two important counts: it reduces time and space algorithm complexity (compared to the traditional neural network approaches), and at the same time it facilitates learning and generalization, thus providing better compression performance at a smaller cost.

Neural network algorithms are well known to be sensitive to parameters governing the learning process. We have not gone to great lengths to optimize our neural network training algorithm. The results obtained from our original attempts, together with the qualitative information represented in Figures 2(a) through 4, are encouraging. Future work may include a more thorough examination of some alternatives for the network's architecture, as well as a further study of other settings of the parameters (including longer than order-10 contexts). Although our PSM method has lower complexity than the alternative neural network approach, the running time is still very slow, even when compared to the "slow" PPM methods. The problem of reducing the complexity of the algorithm even more, so that it becomes practical, is still open. One possibility is to use a hybrid of PSM and other faster methods that could exploit the advantages of a re-representation in a more efficient way. For instance, techniques other than neural networks can be considered both for computing the re-representation and for the prediction step. Examples of techniques that we have considered for re-representation computation include Traveling Salesman Problem heuristics for approximation algorithms, simulated annealing, spring-force iterative optimization, hierarchical clustering and multi-dimensional scaling methods.

In addition, the proposed approach of alphabet re-representation may prove useful for a variety of other problems. For instance, training the network on certain (known) types of text and using it to test the compressibility of other texts of unknown origin may give some insights about the texts' sources. Alternatively, computing an alphabet re-representation from two different data sets and comparing the two re-representations could provide statistical information about the predictive characteristics of the two sources. The application of the alphabet re-representation idea to other text processing domains is an interesting open problem that can be considered in future work.

References

Bell, T. C., Cleary, J. G., & Witten, I. H. (1990). *Text Compression.* Prentice Hall.

Chauvin, Y., & Rumelhart, D. E. (1995). *Backpropagation: Theory, Architectures, and Applications.* Lawrence Erlbaum Associates, Inc.

Cleary, J. G., & Witten, I. H. (1984). Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communication*, *32*, 396–402.

Cottrell, G. W., Munro, P., & Zipser, D. (1989). Image compression by backpropagation: an example of extensional programming. In Sharkey, N. E. (Ed.), *Models of cognition: a review of cognition science.* Norwood, NJ.

Faloutsos, C., & Lin, K.-I. Fastmap: a fast algorithm for indexing, data-mining, and visualization of traditional and multimedia datasets. Technical Report, Department of Computer Science, University of Maryland, College Park.

Gallagher, R. (1968). *Information Theory and Reliable Communication.* John Wiley & Sons.

Hochreiter, S., & Schmidhuber, J. (1997). Flat minima. *Neural Computation*, *9*(1), 1–43.

Lawler, E. L., Lenstra, J. K., Rinnooy, A. H. G., & Shmoys, D. B. (Eds.). (1985). *The Traveling Salesman Problem.* John Wiley & Sons.

Moffat, A., Neal, R., & Witten, I. (1995). Arithmetic coding revisited. *Proceedings of IEEE Data Compression Conference.*

Murtagh, F. (1983). A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, *26*(4), 354–359.

Natsev, A. (1997). Text compression via alphabet re-representation. Master's thesis, Duke University, Durham, NC. Available at http://www.cs.duke.edu/~natsev/thesis.

Rumelhart, D., Hinton, D., & Williams, R. (1986). *Parallel Distributed Processing, Explorations in the Microstructure of Cognition, Vol. 1: Foundations.* MIT Press.

Schalcoff, R. (1992). *Pattern Recognition.* John Wiley & Sons, Inc.

Schmidhuber, J., & Heil, S. (1996). Sequential neural text compression. *IEEE Transactions on Neural Networks*, *7*(1), 142–146.

Thodberg, H. H. (1991). Improving generalization of neural networks through pruning. *International Journal of Neural Systems*, *1*(4), 317–326.

Weigend, A. S., Rumelhart, D. E., & Huberman, B. A. (1991). Generalization by weight-elimination with application to forecasting. *Advances in Neural Information Processing*, *3*, 875–882.

Welch, T. A. (1984). A technique for high performance data compression. *Computer*, 8–19.

Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Ph.D. thesis, Harvard University.

Wheeler, D., & Burrows, M. (1994). A block-sorting lossless data compression algorithm. Tech. rep. SRC 124, DEC Corporation. Available by ftp at gatekeeper.dec.com: /pub/DEC/SRC/research-reports/SRC-124.ps.Z.

Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, *23*, 337–343.

Ziv, J., & Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, *24*, 530–536.

Figure 1: Architecture of the neural network demonstrating weight partitioning (between 1st and 2nd layers) and blending (between 2nd and 3rd layers). Parameters include: context size n, feature space dimensionality d, and hidden chunk factor h.

Figure 2: Two- and three-dimensional embeddings of a sample 25-dimensional re-representation, along with 3 parameter plane projections of the 3-D embedding.

(a) 2-D embedding.

(b) 3-D embedding.

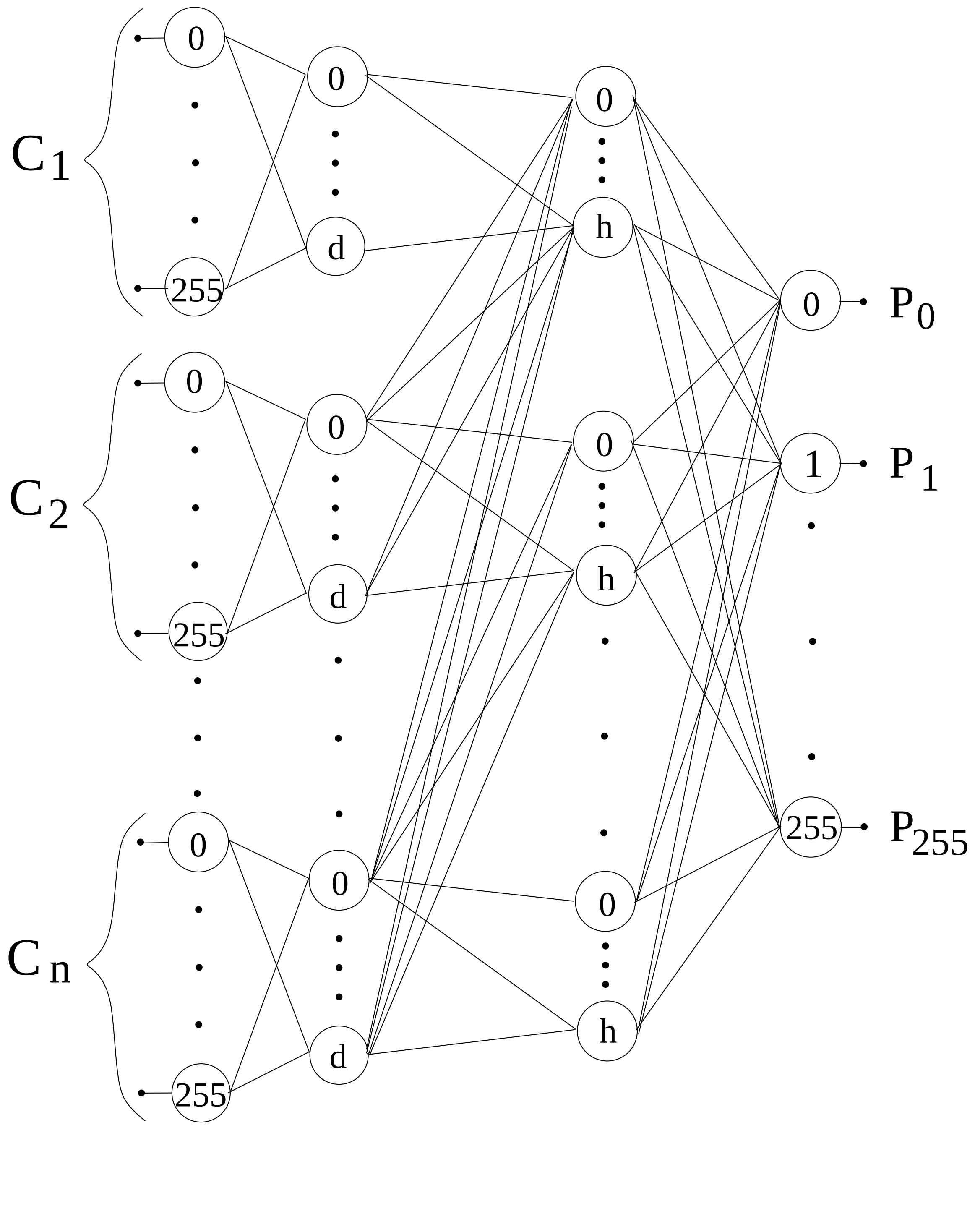
(c) X-axis projection.

(d) Y-axis projection.

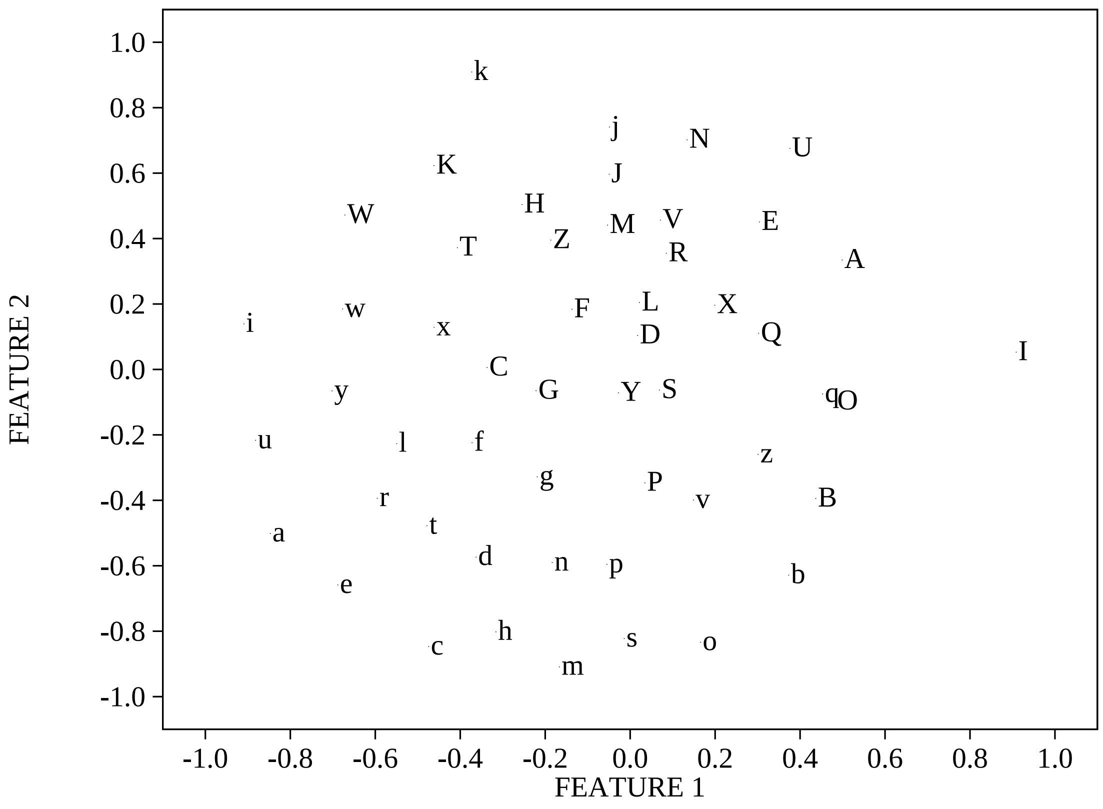
(e) Z-axis projection.

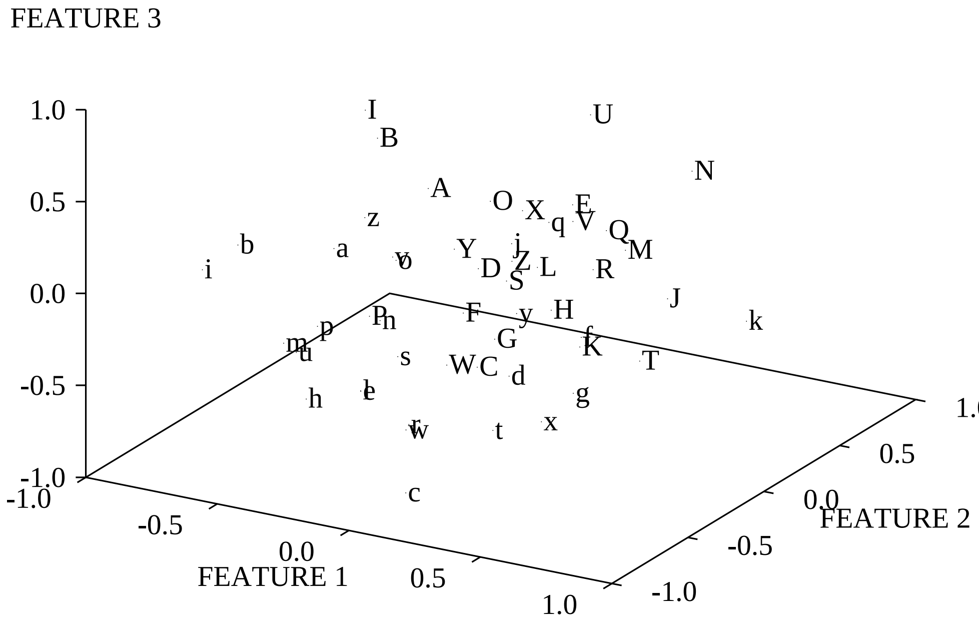
Figure 3: Probability distribution of the next character in contexts "D" and "L".

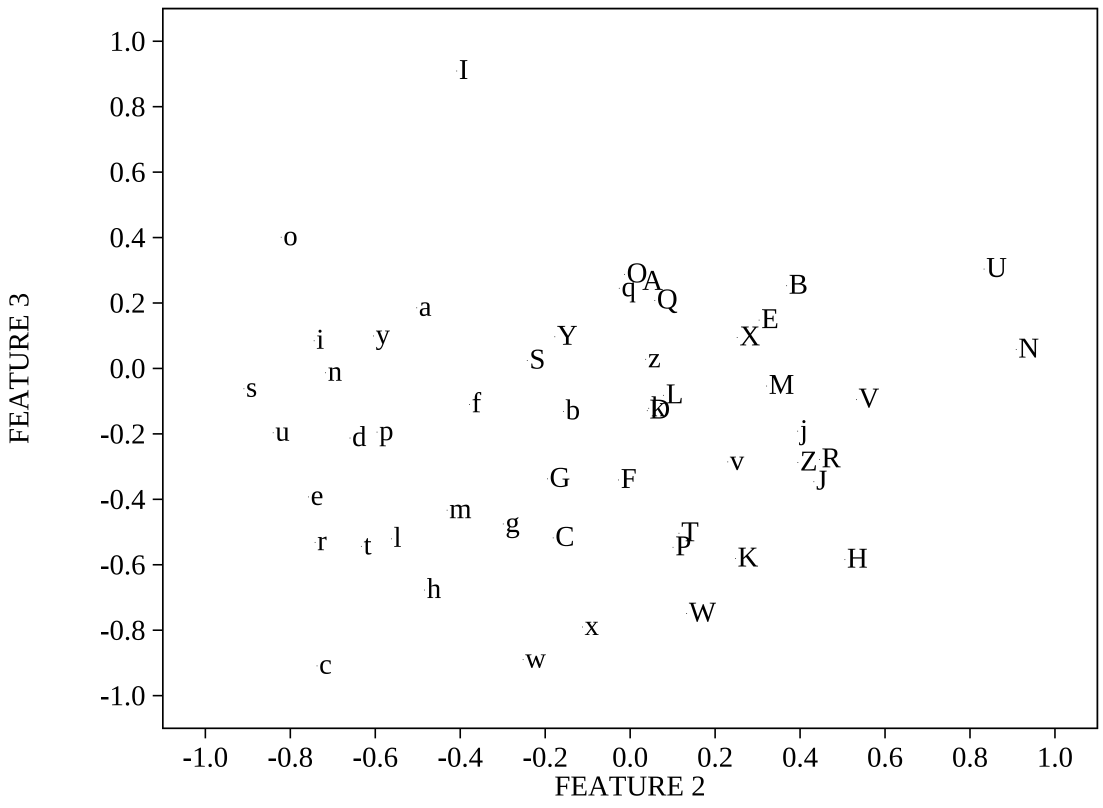
Figure 4: Probability distribution of the next character in contexts "F" and "G".



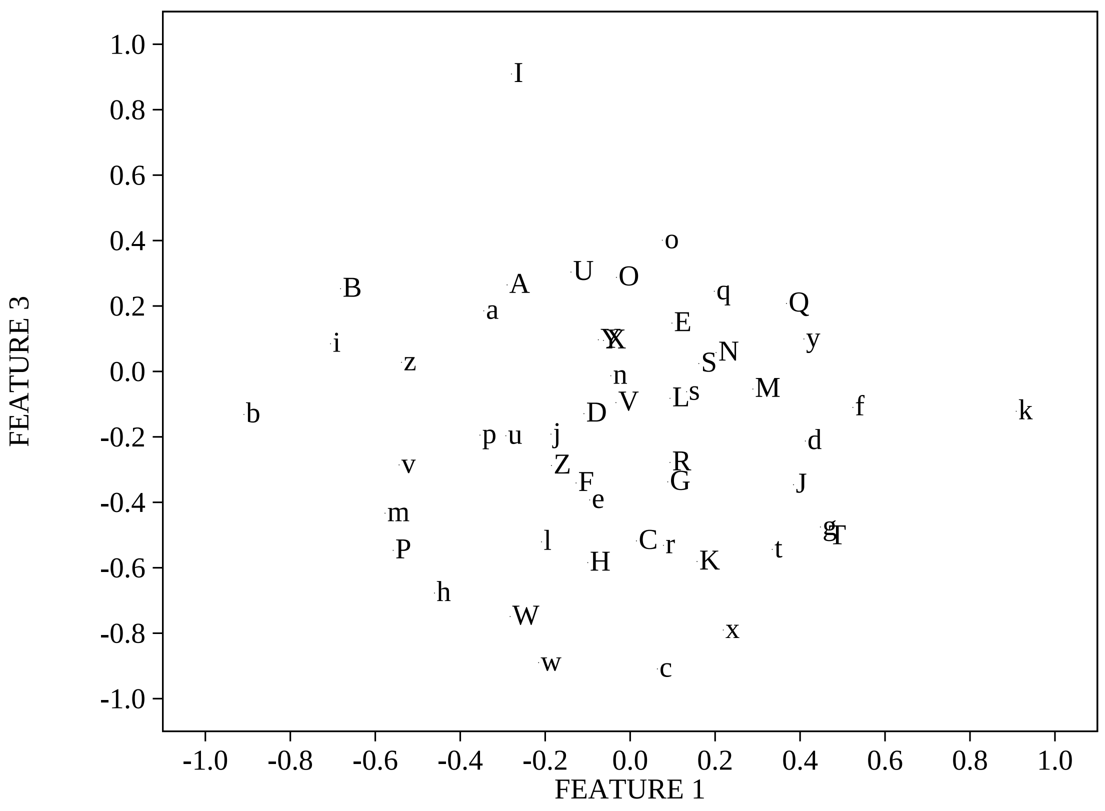
C_1
C_2
C_n
0
255
d
h
1
P_0
P_1
P_{255}

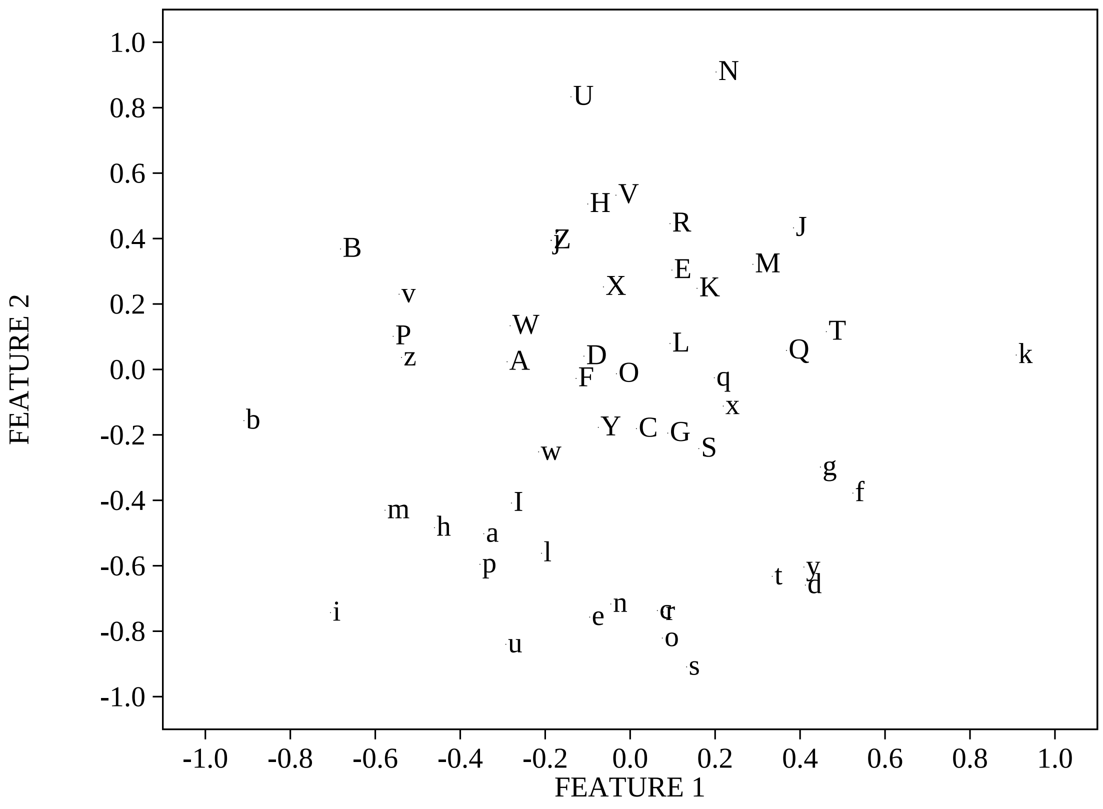

FEATURE 2
FEATURE 1
1.0
0.8
0.6
0.4
0.2
0.0
-0.2
-0.4
-0.6
-0.8
-1.0
k
j
N
U
K
J
W
H
M
V
E
T
Z
R
A
i
w
x
F
L
X
D
Q
I
C
y
G
Y
S
q
O
u
l
f
z
g
P
v
B
r
t
a
d
n
p
b
e
h
s
o
c
m


FEATURE 3
1.0
0.5
0.0
-0.5
-1.0
-1.0
-0.5
0.0
0.5
1.0
FEATURE 1
-1.0
-0.5
0.0
0.5
1.0
FEATURE 2
I
B
U
N
A
O
X
E
q
V
Q
M
z
b
a
v
o
Y
j
Z
L
D
R
S
i
J
k
P
n
F
y
H
p
m
u
G
f
K
s
W
C
d
T
h
l
e
g
r
w
t
x
c

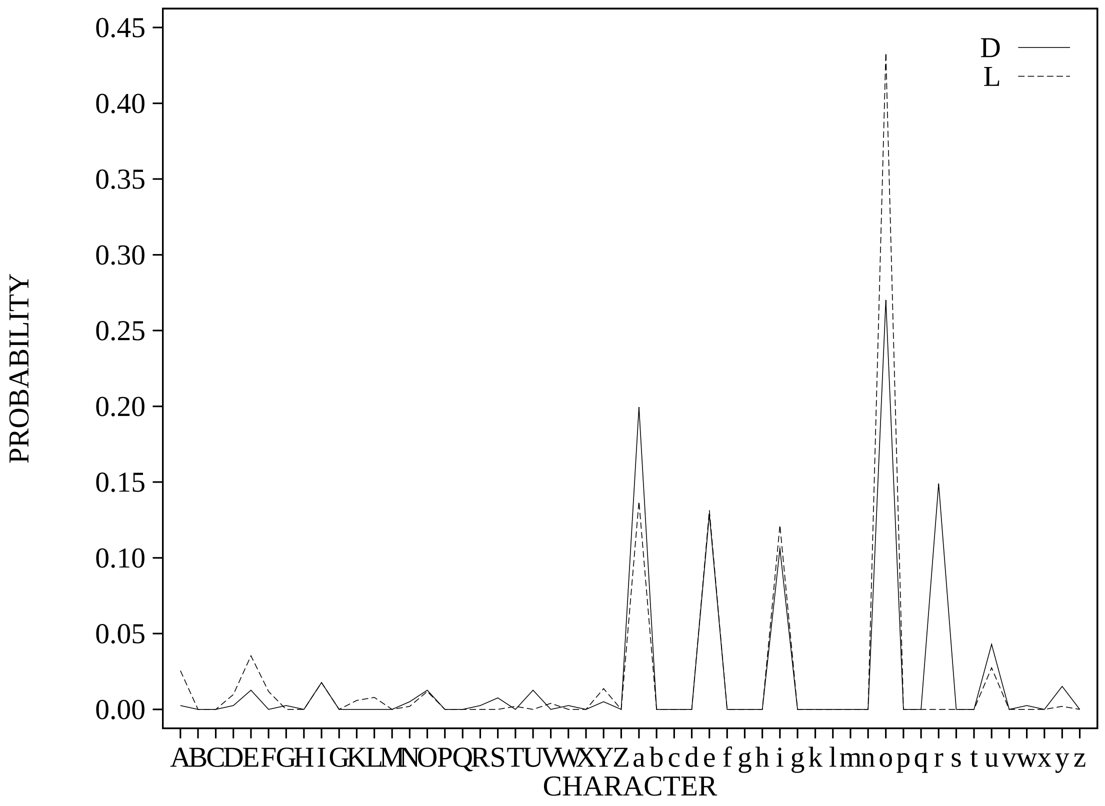


FEATURE 3
FEATURE 2
1.0
0.8
0.6
0.4
0.2
0.0
-0.2
-0.4
-0.6
-0.8
-1.0
-1.0
-0.8
-0.6
-0.4
-0.2
0.0
0.2
0.4
0.6
0.8
1.0
I
o
U
O
A
q
Q
B
a
E
X
N
i
y
Y
S
z
n
M
s
L
V
f
b
D
u
d
p
j
Z
R
v
J
G
F
e
m
g
r
t
l
C
T
P
K
H
h
W
x
c
w

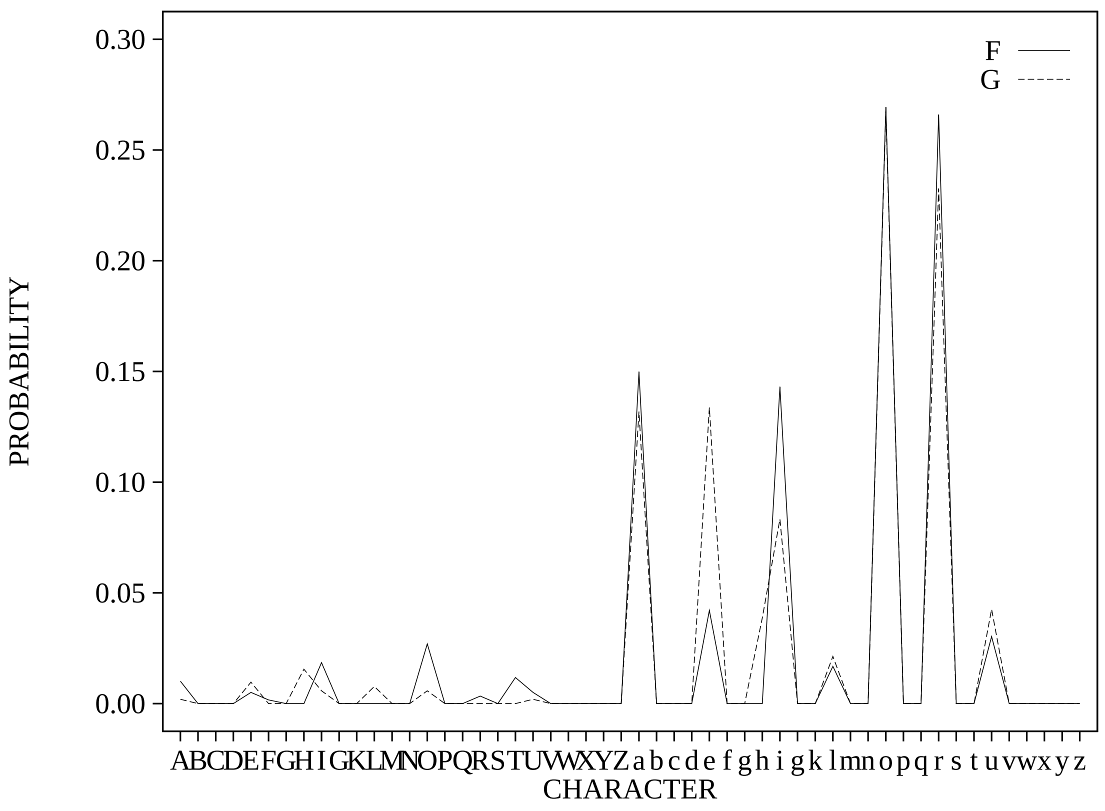

FEATURE 3
FEATURE 1
I
o
U
O
A
B
q
Q
a
E
i
y
Y
X
N
z
S
n
M
L
s
V
b
D
f
k
p
u
j
d
R
v
Z
G
F
J
e
m
g
T
l
C
r
P
H
K
t
h
W
x
w
c



FEATURE 1
FEATURE 2
1.0
0.8
0.6
0.4
0.2
0.0
-0.2
-0.4
-0.6
-0.8
-1.0
N
U
H
V
R
J
B
Z
j
M
E
X
K
v
T
W
P
L
Q
k
z
A
D
O
F
q
b
x
Y
C
G
S
w
g
f
I
m
h
a
l
p
t
y
d
i
n
e
c
r
o
u
s



PROBABILITY
CHARACTER
D
L
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45
ABCDEFGHIGKLMNOPQRSTUVWXYZ a b c d e f g h i g k l m n o p q r s t u v w x y z



F
G
PROBABILITY
0.30
0.25
0.20
0.15
0.10
0.05
0.00
A B C D E F G H I G K L M N O P Q R S T U V W X Y Z a b c d e f g h i g k l m n o p q r s t u v w x y z
CHARACTER