Some Word-Level Language Models

Tudor Dan Dimofte

September 2005

Inference Group, Astrophysics
Cambridge

Contents

1 Introduction

This report describes my work on word-level language models for Dasher during the summer of 2005. I (re)developed and programmed a simple topic model, described in Section 2, as well as an N-gram PPM model (generally $N = 3$), in Section 3. The hope was that the two would contribute valuable information about different aspects of a language. For example, the N-gram model would know that a 'the' never follows another, while the topic model might know that someone is writing more about [computers and politics in a formal literary style] than [health and gardening in a familiar email style].[1] To gain the most benefit of the models' complementary information, they were then 'multiplied' together to form a mixture model described in Section 4.

A few tests of each model, and insights gained from them, are discussed in the respective sections. Unfortunately, the topic model with the learning algorithm described in Section 2 did not end up learning topics very well – at least not given sparse word frequency data. It behaved more like a monogram model than a topic model, in that the number of topics used was often irrelevant. However, due to an ad-hoc addition meant to allow it to learn new words during compression, it actually was much more effective than a simple monogram model, and contributed significantly to the mixture model. Two main conclusions of this research have been that

- A multiplicative model consisting of N-gram PPM $\times$ a [1-previous-topic (monogram distributions) + 1-new-learned-topic (also monogram)] topic model works reasonably well, outperforming both pure topic model and pure N-gram PPM. (Although this mixture model is much slower than its components.)
- If it is desired that a real, discriminating topic model be used, a beter method for learning topics must be employed. Several other potential algorithms seem to be known, *e.g.* [ML] and [BlNJ].

An almost complete summary of data sets (texts) and tests performed, referencing appropriate files and *Mathematica* notebooks which store results, is in Appendix B. Appendix C contains descriptions of my code, and indications for how to use it.

Finally, a note on terminology used herein. The three models described (topic, N-gram, and mixture) all work by first being trained on some data, referred to as the 'learning' or 'training' phase, and then compressing some other data, the 'compression' phase. There is usually some learning going on in the compression phase as well, but when we discuss 'learning' without qualifications we mean the initial training. Also, a model's compression (per word) of data is taken as the standard quantitative measure of that model's performance. By 'compression' we mean the $-\log_2$ probability a model assigns to a data set, divided by its length; if a data set is

[1] It was also suggested that the topic model could also embody some a-priori exclusive choices, such as writing in French or in English. This possibility was not duly explored, although predefined topics consisting of (*e.g.*) a French dictionary and an English dictionary (with monogram probabilities) could easily be included in topic matrices, and the exclusive nature of the choice would not necessarily need to be pre-specified – it would be quickly learned.

the string of words $(x_1, x_2, \ldots, x_N)$, then the compression c is

$$\begin{aligned} c &= -\frac{1}{N}\log_2[P(x_1,\ldots,x_N)] \\ &= -\frac{1}{N}\log_2[P(x_1)P(x_2|x_1)\cdots P(x_N|x_1,\ldots,x_{N-1})] \\ &= \frac{-1}{N}\left[\log_2 P(x_1) + \ldots + \log_2 P(x_N|x_1,\ldots,x_{N-1})\right]. \end{aligned} \tag{1}$$

This is measured in bits/word; lower compression (higher probability of a data set as predicted by the model) is better.

Many of the ideas discussed in this report came from discussions with David MacKay and Phil Cowans, including the structure of the topic model(s) and the N-gram model, and using a multiplied mixture model. I will try to cite other sources if I know of them, where appropriate.

2 A Simple Topic Model

The general, non-hierarchical topic model we use here assumes that a data set is pre-divided into D documents, and that it is constructed in the following way. (Denote the number of topics in the model by T, and the number of distinct words it knows about (the vocabulary size) by A.)

1. For each document d, choose a T-dimensional vector $\boldsymbol{\theta}^{(d)}$ ($\boldsymbol{\theta}^d$ when there is no confusion) from some prior distribution $P(\boldsymbol{\theta})$. The components satisfy $\sum_j \theta_j^d = 1$.

2. For each word $x_i^{(d)}$ of document d, choose a topic $t_i^{(d)}$ with probability $\theta^d_{t_i^d}$. Then choose the actual word from a pre-existing $A \times T$-dimensional topic matrix (or beta matrix) of conditional probabilities $\beta_{x_i^d, t_i^d} = P(x_i^d|t_i^d)$. Note that $\sum_a \beta_{aj} = 1\ \forall j$.

Note that the order of words is irrelevant – hence this is called a "bag of words" model.

Given the formal description above, we use Bayesian inference to learn various parameters of the model, fitting it to data we are interested in. In the learning phase, we are interested in learning all its free parameters, in particular the topic matrix. We end up learning a $\boldsymbol{\theta}$ for each document of training text, but (largely) throw this information away. Then, in the compression phase, we use the fixed pre-learned topic matrix, and actively learn $\boldsymbol{\theta}$'s for each document we are compressing.

In the next section, we derive a simple approximate algorithm for learning topics from training data, which assumes a uniform, uninformative prior $P(\boldsymbol{\theta})$, and in Section 2.2 write down its natural extension to compression. We come back to the question of a prior in Section 2.2.1; including a Dirichlet prior for $\boldsymbol{\theta}$ turns out to be important during compression, but should be largely irrelevant during learning.

The final iterative formulae are summarized at the beginning of Section 2.5.

2.1 Learning topics

As previously noted, this is a bag of words model, so only frequencies of each word in each document of training data should matter. Explicitly denote the vocabulary by $\{1, \ldots, A\}$ and the topics by $\{1, \ldots, T\}$. Then we can define N to be the total number of words in the training data, $N^{(d)}$ to be the total number of words in document d, and $n_i^{(d)}$ to be the frequency of word i in document d.

The probability of data $(x_1^1, ..., x_{N^1}^1, \ldots, x_1^D, ..., x_{N^d}^D) = (\mathbf{x}^1, \ldots, \mathbf{x}^D) = \mathbf{x}$, or $\{n_i^d\}_{\substack{i=1\ldots A \\ d=1\ldots D}}$, given parameters $\{\boldsymbol{\theta}^d\}$ and $\{\beta_{ij}\}$, is

$$\begin{aligned}
P(\mathbf{x}|\boldsymbol{\theta}, \boldsymbol{\beta}) &= \prod_d \prod_{i=1}^{N^d} P(x_i^d|\boldsymbol{\theta}, \boldsymbol{\beta}) \\
&= \prod_d \prod_{i=1}^{N^d} \left[\sum_{t_i^d} P(x_i^d|t_i^d, \boldsymbol{\theta}, \boldsymbol{\beta}) P(t_i^d|\boldsymbol{\theta}, \boldsymbol{\beta}) \right] \\
&= \prod_d \prod_{i=1}^{N^d} \left[\sum_j P(x_i^d|t_i^d = j, \boldsymbol{\beta}) P(t_i^d|\boldsymbol{\theta}^d) \right] \\
&= \prod_d \prod_{i=1}^{N^d} \left[\sum_j \beta_{x_i^d j} \theta_j^d \right] \qquad (2) \\
&= \prod_d \prod_a \left[\sum_j \beta_{aj} \theta_j^d \right]^{n_a^d}. \qquad (3)
\end{aligned}$$

Given a uniform, uninformative prior on each $\boldsymbol{\theta}^d$ and $\beta_{\cdot j}$,

$$P(\boldsymbol{\theta}^d) = (T-1)!\, \delta(\sum_j \theta_j^d - 1), \qquad (4)$$

$$P(\beta_{\cdot j}) = (A-1)!\, \delta(\sum_a \beta_{aj} - 1), \qquad (5)$$

we may use Bayes' theorem to find

$$\begin{aligned}
P(\boldsymbol{\theta}, \boldsymbol{\beta}|\mathbf{x}) &= \frac{P(\mathbf{x}, \boldsymbol{\theta}, \boldsymbol{\beta})}{P(\mathbf{x})} \\
&= \frac{1}{Z} P(\mathbf{x}|\boldsymbol{\theta}, \boldsymbol{\beta}) P(\boldsymbol{\theta}) P(\boldsymbol{\beta}) \\
&= \frac{1}{Z'} P(\mathbf{x}|\boldsymbol{\theta}, \boldsymbol{\beta}), \qquad (6)
\end{aligned}$$

with

$$Z' = \frac{P(\mathbf{x})}{P(\boldsymbol{\theta}) P(\boldsymbol{\beta})} = \int_{\{\boldsymbol{\theta}\}, \{\boldsymbol{\beta}\}} P(\mathbf{x}|\boldsymbol{\theta}, \boldsymbol{\beta}). \qquad (7)$$

Examining the form of (3), we see that it is really a sum of things that will look like Dirichlet distributions, and so Z', in principle, is analytically calculable, as are expectations of each β_{aj} under $P(\boldsymbol{\theta}, \boldsymbol{\beta}|\mathbf{x})$. However, an estimate of the number of calculations needed to find the coefficients and powers of each term in the product of (3) for any moderate number of training words quickly shows that this approach is not actually feasible. Instead, we shall derive a simple EM algorithm which (roughly speaking) converges to the mode of (6), finding the maximum likelihood.

We would expect that the distribution (6) is very highly peaked, and the simplest thing we can do is to try to fit it with delta functions $\delta(\boldsymbol{\theta} - \boldsymbol{\theta}^*)$ and $\delta(\boldsymbol{\beta} - \boldsymbol{\beta}^*)$. Minimizing the KL divergence between the real and delta distributions with respect to the $*$'d parameters should give the maximum likelihood. Indeed, if we use (6) exactly, we will get

$$\begin{aligned} D_{KL}(Q, P) &= \int \delta(\boldsymbol{\theta} - \boldsymbol{\theta}^*)\delta(\boldsymbol{\beta} - \boldsymbol{\beta}^*) \log \frac{\delta(\boldsymbol{\theta} - \boldsymbol{\theta}^*)\delta(\boldsymbol{\beta} - \boldsymbol{\beta}^*)}{P(\boldsymbol{\theta}, \boldsymbol{\beta}|\mathbf{x})} \\ &= -\log P(\boldsymbol{\theta}^*, \boldsymbol{\beta}^*|\mathbf{x}), \end{aligned}$$

plus an infinite constant (the entropy of a delta distribution), which we minimize to get precisely the maximum likelihood. But that's not very helpful! So instead we use

$$\begin{aligned} P(\mathbf{t}, \boldsymbol{\theta}, \boldsymbol{\beta}|\mathbf{x}) &= \frac{1}{Z} P(\mathbf{x}, \mathbf{t}, \boldsymbol{\theta}, \boldsymbol{\beta}) \\ &= \frac{1}{Z} P(\mathbf{x}|\mathbf{t}, \boldsymbol{\beta}) P(\mathbf{t}|\boldsymbol{\theta}) P(\boldsymbol{\theta}) P(\boldsymbol{\beta}) \qquad (8) \\ &= \frac{1}{Z'} \prod_d \prod_{i=1}^{N^d} P(x_i^d|t_i^d, \boldsymbol{\beta}) P(t_i^d|\boldsymbol{\theta}) \\ &= \frac{1}{Z'} \prod_d \prod_{i=1}^{N^d} \beta_{x_i^d t_i^d} \theta_{t_i^d} \qquad (9) \end{aligned}$$

(we can marginalize this over $\mathbf{t}$ to get (3)). We compare (9) with a fully separable distribution

$$\begin{aligned} Q(\mathbf{t}, \boldsymbol{\theta}, \boldsymbol{\beta}) &= Q(\boldsymbol{\theta}) Q(\boldsymbol{\beta}) \prod_d \prod_{i=1}^{N^d} Q(t_i^d) \\ &= \delta(\boldsymbol{\theta} - \boldsymbol{\theta}^*)\delta(\boldsymbol{\beta} - \boldsymbol{\beta}^*) \prod_d \prod_{i=1}^{N^d} q_{t_i^d}^{(i,d)}, \qquad (10) \end{aligned}$$

depending on parameters $\boldsymbol{\theta}^*$, $\boldsymbol{\beta}^*$, and $\{q_j^{i,d}\}$; the q's are distributions over topics, like the θ's, but now there is one for each word.

Considering the KL divergence, we obtain

$$\begin{aligned}
D_{KL} &= E_Q\left[\log Q(\mathbf{t},\boldsymbol{\theta},\boldsymbol{\beta}) - \log P(\mathbf{t},\boldsymbol{\theta},\boldsymbol{\beta})\right] \\
&= E_Q\left[\log \delta(\boldsymbol{\theta}-\boldsymbol{\theta}^*)\delta(\boldsymbol{\beta}-\boldsymbol{\beta}^*) + \log \prod_d \prod_{i=1}^{N^d} q_{t_i^d}^{i,d} - \log \frac{1}{Z'} \prod_d \prod_{i=1}^{N^d} \beta_{x_i^d t_i^d} \theta_{t_i^d}\right] \\
&= E_Q\left[\sum_d \sum_{i=1}^{N^d} \left(\log q_{t_i^d}^{i,d} - \log \beta_{x_i^d t_i^d} - \log \theta_{t_i^d}\right)\right] + \text{ constants} \\
&= \sum_{\{t_i^d\}} \left(\prod_d \prod_{i=1}^{N^d} q_{t_i^d}^{i,d}\right) \sum_d \sum_{i=1}^{N^d} \left(\log q_{t_i^d}^{i,d} - \log \beta^*_{x_i^d t_i^d} - \log \theta^*_{t_i^d}\right) \\
&= \sum_d \sum_{i=1}^{N^d} \sum_{j=1}^{T} q_j^{i,d} (\log q_j^{i,d} - \log \beta^*_{x_i^d j} - \log \theta^*_j), \qquad (11)
\end{aligned}$$

where we have thrown away constants (irrelevant when minimizing), and for the last step used the fact that $\sum_j q_j^{i,d} = 1$.

Now we minimize (11), maintaining normalizations via Lagrange multipliers, to find the starred parameters (and q's) which best fit our real distribution. We remove the $*$'s from now on for simplicity. For the q's, we add $\gamma^{i,d}(\sum_j q_j^{i,d} - 1)$, and differentiate to get

$$\frac{\partial D_{KL}}{\partial q_j^{i,d}} = \log q_j^{i,d} + 1 - \log \beta_{x_i^d j} - \log \theta_j + \gamma^{i,d} = 0.$$

This leads to a system of equations

$$\begin{cases} q_j^{i,d} + \beta_{x_i^d j}\theta_j \Gamma^{i,d} &= 0 \\ q_1^{i,d} + \ldots + q_T^{i,d} &= 1 \end{cases}$$

for each i, d, with $\Gamma^{i,d} = -e^{-\gamma^{i,d}-1}$. The solution to such a system is

$$q_j^{i,d} = \frac{\beta_{x_i^d j}\theta_j^d}{\sum_k \beta_{x_i k}\theta_k^d}.$$

Similar equations, and a system of the same form, result for the θ's and β's (with appropriate Lagrange multipliers). The final result is:

$$q_j^{i,d} = \frac{1}{\sum_k \beta_{x_i k}\theta_k^d}\, \beta_{x_i^d j}\theta_j^d \quad \forall\, i,d \qquad (12)$$

$$\theta_j^d = \frac{1}{N^d} \sum_{i=1}^{N^d} q_j^d \quad \forall\, j,d \qquad (13)$$

$$\beta_{aj} = \frac{1}{\sum_{d,i} q_j^{i,d}} \sum_{i,d|x_i^d = a} q_j^{i,d} \quad \forall\, a,j. \qquad (14)$$

This is a system of equations which might in principle be solved exactly for all the parameters; but an exact solution is a very difficult task. On the other hand, (12)-(14) appear perfectly suited for an iterative approximation. They describe a sort of EM algorithm, alternating between assigning q's for each word, and recalculating the θ's and β's from them.[2]

For a slight recasting in better notation, note that (12) implies that $q_j^{i,d} = q_j^{i',d}$ whenever $x_i^d = x_{i'}^d$. Thus we may write $q_j^{a,d}$ for any $q_j^{i,d}$ with $x_i^d = a$, and rewrite

$$q_j^{a,d} = \frac{1}{\sum_k \beta_{ak}\theta_k^d}\,\beta_{aj}\theta_j^d \tag{15}$$

$$\theta_j^d = \frac{1}{N^d}\sum_{a=1}^{A} n_a^d q_j^{a,d} \tag{16}$$

$$\beta_{aj} = \frac{1}{\sum_{d=1}^{D}\sum_{b=1}^{A} n_b^d q_j^{b,d}}\sum_{d=1}^{D} n_a^d q_j^{a,d}. \tag{17}$$

This is now explicit bag-of-words notation, with $\{n_a^d\}$ containing all pertinent information of a data set.

2.2 Learning thetas and compressing documents

When using the topic model, we will first iterate equations (15)-(16) with some training data in the learning phase to learn a topic matrix $\boldsymbol{\beta}$. Then we want to keep this matrix fixed while we try to compress another set of documents, via:

For each document d:

1. Take a word x_i^d, and find it's probability given $\boldsymbol{\beta}$ and the best current guess for $\boldsymbol{\theta}$ for this document. Use that probability to update the compression (add $-\log_2 P(x_i^d)$ to it; we divide by N at the end).

2. Try to re-learn $\boldsymbol{\theta}^d$ for this document given all the words that have been read so far. Repeat.

The distribution $\boldsymbol{\theta}$ for the first word of each document may be set uniform, or chosen based on previous documents, etc, but it should not (and indeed does not) make too much difference for sufficiently long documents, since only the probability of the first word is affected.

The easiest way to learn each $\boldsymbol{\theta}^d$ (we can ignore the d superscripts here, since during compression we're only concerned with single documents) would be to use the algorithm derived in Section 2.1, but with $\boldsymbol{\beta}$ held fixed:

$$\begin{aligned} q_j^a &= \frac{1}{\sum_k \beta_{ak}\theta_k}\,\beta_{aj}\theta_j \\ \theta_j &= \frac{1}{N+\alpha}\sum_a n_a q_j^a. \end{aligned} \tag{18}$$

[2]Convergence of this algorithm has not been proven, but all experimental tests do suggest that it happens, sometimes quite quickly. More on this later.

We tried this initially, but obtained very high compressions.[3] It was decided that a possible source of the problem was the lack of an informative prior $P(\boldsymbol{\theta})$ – we have just been using a uniform prior $(T-1)!\,\delta(\sum_j \theta_j - 1)$. To see why, consider a document that, near the very beginning, contains a word which belongs to one topic t' with very high probability, and all the other topics with negligible probability. Then in step 2 above, using (18), the compression algorithm will decide that $\theta_{t'} \approx 1$ and $\theta_{t \neq t'} \approx 0$. This is the $\boldsymbol{\theta}$ which will be used to find the probability of the next word in the document in the following step 1, and if it should happen to have very low probability in topic t' (quite possible), regardless of the other topics, it will add a huge amount to the compression.

The solution to this particular problem was to use a Dirichlet prior on $\boldsymbol{\theta}$, of the form

$$P(\boldsymbol{\theta}) = \tfrac{1}{Z(\mathbf{u})} \theta_1^{u_1 - 1} \cdots \theta_T^{u_T - 1} \delta(\textstyle\sum_j \theta_j - 1). \tag{19}$$

Letting Δ_T denote the $(T-1)$-dimensional simplex $\sum_k \theta_k = 1$, a typical Dirichlet integral evaluates to:

$$\int_{\Delta_T} d^T\boldsymbol{\theta}\; \theta_1^{u_1 - 1} \cdots \theta_T^{u_T - 1} = Z(\mathbf{u}) = \frac{\Gamma(u_1) \cdots \Gamma(u_T)}{\Gamma(u_1 + \ldots + u_T)}. \tag{20}$$

The uniform distribution is a special case of (19) with all $u_i = 1$, hence has the constant value $\Gamma(T) = (T-1)!$. It is standard to also define $\alpha = \sum_i u_i$ and $m_i = u_i/\alpha$ (so $\sum_i m_i = 1$). The parameters $\mathbf{u}$, or α and $\mathbf{m}$, are referred to as 'hyperparameters' of the topic model. α in particular roughly indicates the number of words the model should read before starting to be sure of its inferred $\boldsymbol{\theta}$. The m_i indicate which value of $\boldsymbol{\theta}$ to start from without any word-frequency information. The formula for properly inferring $\boldsymbol{\theta}$ given a Dirichlet prior is quite simple:

$$\begin{aligned} q_j^a &= \tfrac{1}{\sum_k \beta_{ak}\theta_k}\, \beta_{aj}\theta_j \\ \theta_j &= \tfrac{1}{N+\alpha}(\textstyle\sum_a n_a^d q_j^a + u_j). \end{aligned} \tag{21}$$

Note that, unless α is much greater than the typical document size, the presence of a $\boldsymbol{\theta}$ prior should be irrelevant during topic learning, as it will be overwhelmed by the amount of data present.[4]

[3] A warning – this drastic effect, compressions around 100 bits/word, was not reproduced in the large body of optimization tests performed later with the rewritten hyperparameter-enabled code. It is quite possible that there was simply a mistake in the early code (assuming everything is correct now). It may or may not be valuable to look further into this. Whatever the cause, the initial problem did spur the introduction of hyperparameters, which have been incredibly useful in both topic model and mixture model compression. Moreover, the procedure described in Section 2.2.1 is theoretically preferable.

[4] Why, when the uniform distribution has $\alpha = T$ and $m_i \equiv 1/T$, did it not take T words to get significantly away from a uniform $\boldsymbol{\theta}$? Because we also failed to use the right basis before when deriving (18). The proper choice is in the softmax basis, explained in Section 2.2.1. Again, this should not really affect topic learning, so (15)-(17) should still be reliable. (Unless some special effect happens with $\boldsymbol{\beta}$; I do not expect so, but this should be calculated....)

2.2.1 Derivation of Eqn. 21

We can restrict ourselves to a single document. To get (18), we minimize the KL divergence between $P(\mathbf{t}, \boldsymbol{\theta}|\mathbf{x})$ and $Q(\mathbf{t}, \boldsymbol{\theta}) = \text{const} \cdot \prod_i q^i_{t_i} \delta(\boldsymbol{\theta} - \boldsymbol{\theta}^*)$. A somewhat better idea, however, is to use a delta function in the more natural 'softmax' basis. In this basis, we write

$$\theta_i = \frac{1}{Z_\theta} e^{a_i}, \qquad Z_\theta = \sum_i e^{a_i}, \tag{22}$$

using the parameters $-\infty < a_i < \infty$ in favor of the θ_i. In order to preserve dimensionality, we impose an arbitrary condition $\sum_i a_i = 0$ – so now there are as many independent a_i as there were θ_i.

Choose for the moment $(\theta_1 ..., \theta_{T-1})$ and $(a_1, ..., a_{T-1})$ to be the independent sets of variables, and $\theta_T = 1 - \theta_1 - \ldots - \theta_{T-1}$, $a_T = -a_1 - \ldots - a_{T-1}$. Under change of variables, a probability density will transform as

$$P(\boldsymbol{\theta}) d^{T-1}\boldsymbol{\theta} = P(\mathbf{a}) d^{T-1}\mathbf{a} = P(\boldsymbol{\theta}(\mathbf{a})) \left| \frac{\partial\, \theta_1, ..., \theta_{T-1}}{\partial\, a_1, ..., a_{T-1}} \right| d^{T-1}\mathbf{a},$$

or

$$P(\boldsymbol{\theta}) = P(\mathbf{a}) \cdot |\mathcal{J}|, \tag{23}$$

with $\mathcal{J}$ the Jacobian. In our case, the distribution we want is (8) with an appropriate $\boldsymbol{\theta}$ prior, marginalized over $\boldsymbol{\beta}$, or

$$\begin{aligned}
P(\mathbf{t}, \boldsymbol{\theta}|\mathbf{x}) &= \frac{1}{Z} P(\mathbf{x}|\mathbf{t}, \boldsymbol{\theta}) P(\mathbf{t}|\boldsymbol{\theta}) P(\boldsymbol{\theta}) \\
&= \frac{1}{Z} \prod_{i=1}^{N} (\beta_{x_i t_i} \theta_{t_i}) \cdot \frac{1}{Z(\mathbf{u})} \prod_{j=1}^{T} \theta^{u_j - 1} \\
&= \frac{1}{Z''} \prod_i \left(\beta_{x_i t_i} e^{a_{t_i}} (\sum_k e^{a_k})^{-1} \right) \cdot \prod_j \left(e^{a_j (u_j - 1)} (\sum_k e^{a_k})^{1 - u_j} \right) \cdot |\mathcal{J}| \\
&= \frac{1}{Z''} \frac{1}{Z_\theta^{N+\alpha}} Z_\theta^T \prod_i (\beta_{x_i t_i} e^{a_{t_i}}) \cdot \prod_j e^{a_j (u_j - 1)} \cdot |\mathcal{J}|.
\end{aligned}$$

The Jacobian can be worked out using an arbitrary set of $T - 1$ independent θ's and a's as above, and it turns out that $\mathcal{J} = T Z_\theta^{-T}$. Using this, and the fact that $\prod_j e^{a_j} = 1$ (by the restriction on a's), we obtain

$$P(\mathbf{t}, \boldsymbol{\theta}|\mathbf{x}) = \frac{1}{Z'''} \frac{1}{Z_\theta^{N+\alpha}} \prod_i (\beta_{x_i t_i} e^{a_{t_i}}) \prod_j a^{a_j u_j}. \tag{24}$$

To get an iterative algorithm, we take the KL divergence between this and

$Q(\mathbf{t}, \mathbf{a}) = \prod_i q^i_{t_i}\, \delta(\mathbf{a} - \mathbf{a}^*)$.[5] Again, we ignore an (infinite) additive constant.

$$\begin{aligned} D_{KL} &= E_Q[\log Q - \log P] \\ &= \sum_{\{t_i\}} \left(\prod_i q^i_{t_i} \right) \left[\sum_i (\log q^i_{t_i} - \log \beta_{x_i t_i} - a^*_{t_i}) - \sum_j a^*_j u_j + (N+\alpha) \log Z_\theta \right] \\ &= \sum_j \left[\sum_i q^i_j (\log q^i_j - \log \beta_{x_i j} - a^*_j) - a^*_j u_j \right] + (N+\alpha) \log Z_\theta. \end{aligned} \tag{25}$$

We can drop the $*$'s on the a_j.

Taking $\partial / \partial q^i_j$ with Lagrange multipliers simply yields the old equations for q^i_j. The important difference is in the $\boldsymbol{\theta}$ update. There may be a better way to compute it with a good Lagrange multiplier, but a method which *does* work is again to consider a_T a dependent variable, and to differentiate (25) with respect to a_j, $j = 1, \ldots, T-1$. We get

$$\begin{aligned} -\frac{\partial D_{KL}}{\partial a_j} &= \sum_i q^i_j + u_j - \sum_i q^i_T - u_T - \frac{N+\alpha}{Z_\theta}(e^a_j - e^a_T) \\ &= D_j - D_T - (N+\alpha)(\theta_j - \theta_T) = 0, \end{aligned}$$

where $D_j = \sum_i q^u_j + u_j$. Then with the condition $\sum_j \theta_j = 1$, we have a linear system

$$\begin{cases} \theta_j - \theta_T &= \dfrac{1}{N+\alpha}(D_j - D_T), \quad j \le T-1 \\ \theta_1 + \ldots + \theta_T &= 1. \end{cases}$$

The solution to this is precisely

$$\theta_j = \frac{1}{N+\alpha} \left(\sum_i q^i_j + u_j \right), \quad \forall j. \tag{26}$$

Taking into account, as before, that the q equations imply $q^i_j = q^{i'}_j$ whenever $x_i = x_{i'}$, we have derived (21).

2.2.2 Values for hyperparameters

Now, we have chosen to keep the hyperparameters fixed during compression, rather than trying to learn them. What values should they be set to? There are (at least) three possible ways to answer this.

1. One might inspect the actual behavior of the topic model given enough data to overwhelm $\boldsymbol{\theta}$ priors, thereby gleaning a rough value for $\mathbf{m}$, the mean of the Dirichlet distribution. Likewise, α should be the rough number of words read in each document before the model is confident it knows what $\boldsymbol{\theta}$ distribution to

[5] There is of course an implicit $\delta(a_1 + \ldots + a_T)$ in both densities, restricting integration to a hypersurface. We have been ignoring such factors where they are not crucial to a calculation.

use; *or* one may think of what α means in a Dirichlet distribution – very small α means that samples from the distribution tend to have $\boldsymbol{\theta}$ highly concentrated on one topic, whereas large α means that samples tend to have $\boldsymbol{\theta} \approx \mathbf{m}$. Either way, a plausible value for α could be obtained via introspection.

2. One could run the topic model on various data sets, and try to pick α and $\mathbf{m}$ which optimize the compression, at least roughly, for as many data sets as possible.

3. One could infer the hyperparameters from a set of large, representative documents. Again, *large* so that priors are overwhelmed. Given D such documents with known distributions $\boldsymbol{\theta}^d$, approximate formulae for α and $\mathbf{m}$ are

$$m_j = (\theta_j^1 \cdots \theta_j^D)^{1/D} / Z_m \tag{27}$$

$$\alpha = -\frac{T-1}{2 \log Z_m}, \tag{28}$$

with normalization constant $Z_m = \sum_j (\theta_j^1 \cdots \theta_j^D)^{1/D}$. These are derived in Appendix A.

In practice, method 2 was used. The training data used in topic-learning, a set of many large, representative documents, seemed perfect for applying method 3. Unfortunately, as described in Section 2.5.1, $\boldsymbol{\theta}^d$'s during topic learning tended to converge to distributions concentrated on a single topic; the tendency sharpened with increasing iterations. Predicted values for α and $\mathbf{m}$ were generated during the construction of topic matrices, but α was always very small (for example, 0.045 after 400 iterations) and kept decreasing with each iteration.

The $\boldsymbol{\theta}$'s learned during compression, on the other hand, were much more well-behaved (perhaps too well-behaved). In retrospect, we see that it is really not appropriate to try to infer $\boldsymbol{\theta}$ hyperparameters from topic learning, due to the much greater "freedom" in the number of parameters being varied then.[6] It may be possible in the future to use method 3 with compression data, although we would expect it to produce quite the same answers as method 2 (but maybe faster).

2.3 New words

So far we have implicitly assumed we were working with a fixed vocabulary, which in reality is never the case. Several approaches were taken to fix this problem during testing. (The following applies to all three models discussed in this report, not just the topic model.)

The easiest thing to do when encountering a brand new word during compression is to ignore it, and indeed this is how much optimization was done. One must be careful, though, to divide the final compression (bits for the data set) by the number of old words encountered, as opposed to the total size of the data set.

We didn't want to completely ignore new words, however, and also tried to use something like a Dirichlet process to predict their probabilities. The initial version of

[6] See further discussion in Section 2.5.2.

this method involved a parameter α_{new}, a sort of escape probability for new words. Given a known vocabulary size of A words, the probability of getting a new word was set to be $\alpha_{new}/(A + \alpha_{new})$, and the probablity of an old word $A/(A + \alpha_{new})$. Thus during compression, if an old word came up and the model predicted it with probability p, then its recorded probability (included in compression statistics) would be set to the slighly smaller $p \cdot A/(A + \alpha_{new})$. If a new word came up, it would automatically get probability $\alpha_{new}/(A + \alpha_{new})$.

This approach did account for new words, but often led to compressions *better* than those achieved by ignoring unknown vocabulary. This was highly counter-intuitive, and it soon became clear that assigning $\alpha_{new}/(A + \alpha_{new})$ to any new word was unrealistic and much too high – for the quantity should be somehow split among 'all possible new words,' not assigned to each of them. So the final solution was to use the fact that in the character-based PPM already employed in Dasher, average information content (compression) is about 2 bits/character. So the probability of a random word of length n may be, on the average, 2^{-2n}, and our final solution to the new word problem was to assign probability

$$\frac{\alpha_{new}}{A + \alpha_{new}} \cdot 2^{-2n}$$

to new words of length n, and

$$\frac{A}{A + \alpha_{new}} \cdot (\text{prob. predicted by model})$$

to words already in the vocabulary. This seemed a very reasonable approach, and possibly very similar to something that would be used in Dasher – new words would in fact need to be spelled out! (This final approach is also more properly normalized.)

These two methods of including new words were tested with both the topic and N-gram models, as explained in the respective results sections. The tests suggested that value of α_{new} used (if at all) may have an effect independent of the other model parameters, though the evidence for this was *not* conclusive. For this reason, no α_{new} tests at all were done during mixture model compression.

2.4 Topic-learning during compression

When using the topic model (with a fixed topic table) to read and compress some text, we thought it would be a good idea to allow the model to continue to learn somehow from this text, rather than just blindly compressing it. This could allow the model to adjust to compression data that is very different from training (topic learning) data. However, we did not want to formally infer any more topics – and given the skewed $\boldsymbol{\theta}$ distributions coming out of topic learning, it is a good thing we didn't.

Our solution was to add one new topic to the pre-learnt topic matrix which stored monogram probabilities (*i.e.* proportional to single word frequency) of all words read during compression. This topic was created after the first document in compression data was read, and updated at the end of each subsequent document. In practice, we

set all hyperparameters u_i equal for the pre-existing topics (so $m_i \equiv 1/T$, $\alpha = T \cdot u$), and gave a different u_{nd} to the new topic.[7]

This ad-hoc approach to learning turned out to have significant consequences. Most importantly, even though the pre-learnt topics themselves did not discriminate too effectively, making these topics look just like a single monogram model, the addition of the new topic (with u and u_{nd} optimized) improved pure monogram compression by as much as 0.6 bits/word. (Compare the first data point in lists 3.1 and 3.2, Appendix B.) When used multiplicatively with the N-gram model, the hybrid topic model may also have improved monogram compression by as much as $0.5 - 0.6$ bits/word.[8] On the other hand, the hybrid model may well have clouded some of the behavior of the topic model itself, and if the topic model is to be diagnosed and improved at some point, this compression-time learning feature should be removed. (It should not be difficult to do so, but the code does not yet accomodate this.)

2.5 Compression tests and optimization

We now present the results of several real tests of the topic model. We can summarize it's main components and adjustable parameters, most of which have been described above. In the variety of the model which was used for testing, we have:

- The number of learned topics T.
- A 'beta matrix' (topic matrix) $\boldsymbol{\beta}$, which is learned from training data during a learning phase.
- A hyperparameter u for each topic in the learned beta matrix.
- A hyperparameter u_{nd} for the new topic learned during compression.

The hyperparameters u and u_{nd} were the main focus during topic model optimization, and it was eventually realized that the two parameters

$$Tu = T \cdot u, \tag{29}$$

$$\mu = \frac{u_{nd}}{Tu} \tag{30}$$

may be better quantities to work with, so we introduce them here. μ is referred to as the 'multiplier.'

All text files were reformatted to contain only lowercase words, with no numbers or punctuation. (Uppercase letters were converted to lowercase.) Initially, document separations were put in by hand, but it was then decided that it would be reasonable to automatically make every paragraph a document. All the data used for testing of the topic model (and mixture model) follows this convention, with an additional restriction that no document may contain less than 40 words. A full list of text files used is in Appendix B.1.

[7] *nd* for 'new document.' Not the best way to name it, but it made sense during coding, given that the new topic was updated with every new document.

[8] More data is needed to support this claim, however.

In addition, there exists an option to consider sentence breaks as words, which was not used with the pure topic model. It might be used in the future, to allow better comparisons to N-gram and mixture models which do use this; it seems likely, however, that the effect of including sentence breaks will be orthogonal to other optimization performed (if they are included in *both* beta matrices and compression data).

2.5.1 Building beta matrices

Equations (15)-(17) were iterated to calculate topic matrices from several of the files in B.1. The `bmxbuild` program, described in Appendix C.3, was the general tool for doing this. Since some calculations could take quite a long time, the matrices were stored in `.bmx` files, which are summarized in Appendix B.2.

The number of iterations performed ranged between 40 and 400. The latter was considered enough iterations for all practical purposes, but sometimes the number was limited by an `NaN` error in `C++` floating point calculations. As we have mentioned before, the $\boldsymbol{\theta}^d$ during topic learning tended to become very highly concentrated around a certain topic; that is, many $\boldsymbol{\theta}^d$ looked like $(0, \ldots, 0, 1, 0, \ldots, 0)$, with the 1 in some seemingly random place. This was not a rule, and splits between 2 or sometimes even 3 topics did happen, but it was a strong tendency. Possibly due to numbers very close to 0 appearing in $\boldsymbol{\theta}^d$'s, possibly due to their appearing elsewhere, floating point overflows often occurred after a certain number of iterations, especially for large training data sets. The number of iterations indicated in Appendix B.2 for each beta matrix, unless 200 or 400, is the maximum number of iterations for which we were able to run the algorithm without error.

Note that equations (15)-(17) always have a fixed point solution at $q_j^{a,d} \equiv 1/T$, $\theta_j^d \equiv 1/T$, and $\beta_{aj} \equiv 1/A$. This seems to be an unstable fixed point, and it is just unfortunate that these values would be the most natural ones from which to start a calculation. Every topic-learning calculation was thus initialized with random (normalized) values of q's, θ's, and β's. We make note of this, because it seems that the randomness in initial values tended to propagate (somewhat) to a randomness in *which* topic(s) each $\boldsymbol{\theta}^d$ eventually converged to, even accounting for permutations.[9]

This is not necessarily a problem, but may indicate that one exists. The observation does support the conclusion we reached during compression that learned topics are largely irrelevant. On the other hand, we should note that when (at the beginning of this research) the topic-finding algorithm was run with a small amount of dense data – 3 topics, 6 documents, 10 words in the vocabulary, and $1 - 25$ counts of *each* word in every document – the resulting topic matrix and $\boldsymbol{\theta}$ distributions were exactly what one would have expected, and did *not* hit any extreme values. Thus it seems that the topic-learning algorithm itself may be fine, but is not behaving properly due to very *sparse* word counts in real data.

One thing to try in the future would be to use much larger documents, maybe only about twice as many documents as topics, and to see if anything changes. Using paragraph-length documents may not have been the right thing to do.

[9]This phenomenon, and the generation of beta matrices in general, should be inspected more closely at some point.

These problems should have been recognized and addressed sooner, but they were not, and so the rest of this section, and the mixture section, uses the topic model as it is, with beta matrices calculated as best as possible, and listed (again) in Appendix B.2.

The topic-calculating program does have elementary functions for calculating the difference between two beta matrices, or two $\boldsymbol{\theta}$ distributions, essentially using a Euclidean distance formula. It was thought that these values may aid the the study of convergence and could even be compared to threshholds to end the algorithm, but not much has been studied about them or the method of their calculation so far. The difference between the final two $\boldsymbol{\beta}$ and $\boldsymbol{\theta}$ iterations for each beta matrix is also listed in B.2.

2.5.2 Compression tests

Many topic model compression tests were done with the beta matrices described above, attempting to optimize the model's various parameters. The tests are summarized in Appendix B.3.1, and here we shall discuss some of the main conclusions drawn from them.

First let us review the compression process. When compressing a document, the topic model reads one word at a time. For each word, it computes that word's probability to appear given all the information it already knows (*i.e.* $\boldsymbol{\theta}$ and the topic matrix), adds $-\log_2$ of this to the compression, and then updates its guess of $\boldsymbol{\theta}$ given all the words it now knows in the current document. After every document, the $(T+1)$st topic of the topic matrix is updated with new monogram statistics.

The first set of tests shown in B.3.1 gave us a an idea for how the hyperparameters u and u_{nd} of the model behaved, and led to the important realization that for each pair of learning and compressing data the best compression lay along a diagonal line of (almost) minima in the u-u_{nd} plane. (This is not evident from B.3.1, but it is from *Mathematica* notebook 2 cited there.) Now, in retrospect, it is clear we should have switched to coordinates (u, μ) with a multiplier μ rather than (u, u_{nd}) right away, as these are much more independent, but we did not do this explicitly. In terms of Tu (as in (29)), the multiplier μ varied from 0.1 to about 10. In particular, we can estimate for the following pairs of learning and compressing texts:

files	$\boldsymbol{\mu}$
`china_c` → `china_tr`	0.1
`critique_l` → `critique_c`	1
`cvwce_ql` → `enron_c`	1
`china_c` → `vicomte_tr`	1.6
`critique_l` → `women_c`	10

Note that the multiplier roughly scales with the (dis)similarity of the learning and compression texts. We couldn't, however, find a good way to quantify this.

The next set of tests performed assessed the number of $\boldsymbol{\theta}$-learning iterations needed when learning $\boldsymbol{\theta}$ for each word. For 5 topics, about 10 iterations appeared enough to generate compressions reliable enough for optimizing parameters, reproducible to at least three decimal places. Using 5 iterations (list 12, notebook 2) was a bit bumpy, and no good for optimization, but fine for estimating compression at a given point

to within ±0.005 bits/word. 10 iterations were used in all topic model calculations from then on.

The number of topics in the beta matrices of the μ table above is not explicitly listed because of the third set of tests in B.3.1. Here we varied the number of topics in the `china_c` → `vicomte_tr` compression and discovered that u_{nd}/u seems to scale with T – or that $u_{nd}/(Tu)$ stays constant. This observation, checked with several other pairs of texts not listed in B.3.1, is the basis for our definition of μ in (30). During the course of topic and mixture model testing, it also appeared that the minimum value of Tu may be independent of T. This is not completely visible in the `china_c` → `vicomte_tr` tests due to the resolution of u values.

A warning bell may have just gone off. For the data mentioned above show not only that the optimized values of μ and Tu may be T-independent, but that the minimum compressions for these optimized values T-independent as well, or at least as good for $T = 1$ as for larger values! Hence the comment we first made in the introduction, and have alluded to several times since: the topic model isn't functioning like a topic model. An inspection of $\boldsymbol{\theta}$'s during compression roughly confirms this (see Appendix B.4). The model is useful to compression because of its [old topic(s) + new topic] structure rather than significant discrimination by its old topics. This behavior is also highly evident in the mixture model, with the new topic playing an even more important role.

Since the number of topics was found not to be too important, subsequent topic model and mixture model tests were performed with a semi-standard 5 topics, or sometimes just 1.

The last set of tests in B.3.1 investigated the behavior of α_{new}, first with the (problematic) uniform $\alpha_{new}/(A + \alpha_{new})$ probabilities, and then with "strict" $\alpha_{new}/(A + \alpha_{new}) \times 2^{-2 \cdot \text{length}}$ probabilities, as the method was developed. In both cases, the introduction of α_{new} appeared orthogonal to the other parameters of the model, although the evidence is not shown and this claim should be re-checked. From lists 19-21, one sees the problem with uniform new word probabilities: the compressions are lower than non-new-word probabilities, and in the wrong order. `china_c` → `vicomte_tr` should certainly not compress better than `critique_l` → `critique_c`! Using the improved strict method, both issues are remedied.

When actually using a topic model in, *e.g.* , Dasher, one generally does not have the luxury of optimizing parameters for a given pair of learning and compression data sets; they must either be fixed ahead of time, or learned in a standard way. The shapes of the surfaces and curves plotted in the second *Mathematica* notebook help suggest some standard fixed values. It generally seems better to (slightly) overshoot values of Tu and μ than to pick them too small. $Tu \approx$ 10-15 and $\mu \approx 1$ do not seem to do too much harm to the compressions of any pairs of data sets we looked at. (It would, however, be very useful to actually learn μ during compression, since it can vary so much.) Similarly, overshooting low values of α_{new} is better than undershooting them (see Lists 23-25), and we can set $\alpha_{new} \approx 300$.

Finally, for practical purposes we should note that the main limiting factor in carrying out topic model tests on large data sets was the amount of time these calculations required, although, for many topics, an issue of memory consumption sometimes arose. It may have been wise to write a smarter program for finding optimal

values of various topic model parameters in just a few steps (by specifically searching for minima), and such a program could be quite useful in the future; but for these tests, it *was* helpful to see the full shape of certain surfaces and curves.

3 N-Gram PPM

The second language model used was an N-gram PPM model. To train the model, we recorded frequency data from a data set in a trie of depth N – so we obtained information on the frequency of every word given every preceding context of length $\leq N-1$. We then used the PPM algorithm and this frequency information to generate probabilities of words (given previous contexts) for another data set, calculating its compression. Unlike the topic model, the free parameters in PPM are optimized rather than inferred.[10] In practice, the N-gram model turned out to be much faster than the topic model, but also much more memory-intensive.

Generally we used a depth of $N = 3$, though other values were also explored. (It appeared that the best compression happened for $N = 4$ or 5, but such depths also tended to cause memory faults, and $N = 3$ was not significantly worse.)

3.1 Building a trie

We will demonstrate here how a trie is built, and explain "update exclusion" (as we used it).

A trie is a tree of depth N, where every node holds a word and an integer – the frequency of that word, in the context described by the path to the node. Consider the beginning of the file `text2.sf` (referenced in Appendix B.1):

```
the
widening
circle
it
was
$
...
```

The '`$`' symbol denotes a sentence break (unrealistically inserted here). To insert this into a trie, start with the first word, `the`; it gets added to the empty trie with frequency 1, as in Figure 1 (a). Next, consider the 2-word sequence `the widening`; it says that `widening` occurs once by itself, and also once after the context `the`, so we add nodes as in Figure 1 (b). Then we do the same with the 3-word sequence `the widening circle`, counting `circle` by itself, after `widening`, and after `the widening`, producing Figure 1 (c). Now, if our trie is to have depth 3, we can just shift our 3-word frame along the rest of this first sentence, inserting three occurences of `it` and three of `was` into the trie, producing Figure 1 (d).

Essentially, this process just repeats for every sentence in the data set. If a word in a certain context has been seen before, the count of that specific node in the trie is

[10] Though there does exist a decent Bayesian approach to compression from N-gram statistics. See [MBP]. It just (presumably) does not work quite as well as PPM with optimized parameters.

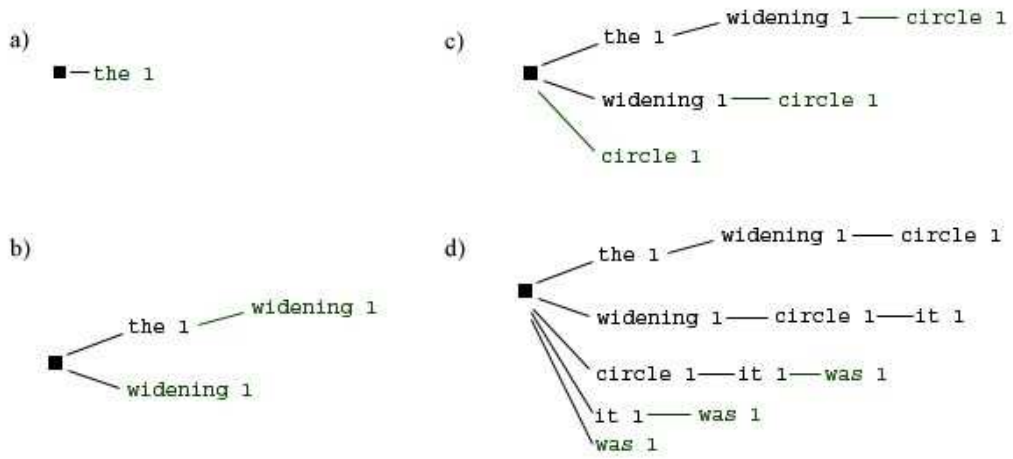

Figure 1: Building a Trie

incremented, instead having another node created. The only potential complication is an option called 'update exclusion.'[11] The next sentence of file `text2.sf` is

```
it
was
very
$
```

The sequence `it` is inserted in Figure 2 (a). Without update exclusion, inserting the sequence `it was` would result in the "normal" version of Figure 2 (b), whereas with update exclusion we get the other version (note the frequency of the depth-1 node `was`). Update exclusion says that, given an N-word sequence $x_1 \cdots x_N$, we start inserting x_N after $x_1 \cdots x_{N-1}$, then x_N after $x_2 \cdots x_{N-1}$, and so on, until we reach a context that has been seen before. The x_N node in the first such context (if it exists) is incremented, and then we stop.

This process may seem strange and ad-hoc, but is easily motivated by the following example. Suppose that some training text has (say) abundant occurences of the sequence `it was`, but few occurances of `was` otherwise. Without update exclusion, a trie we build would have the depth-1 node `was` with very high frequency, potentially leading someone reading the trie to predict that `was` should occur in any context with high probability. However, the information we *should* be encoding is that `was` only occurs with very high frequency after `it`. Update exclusion has precisely this effect.

Inserting the final sentence of `text2.sf`,

```
burdensome
to
ursula
that
she
was
the
```

[11] See [B] for more information.

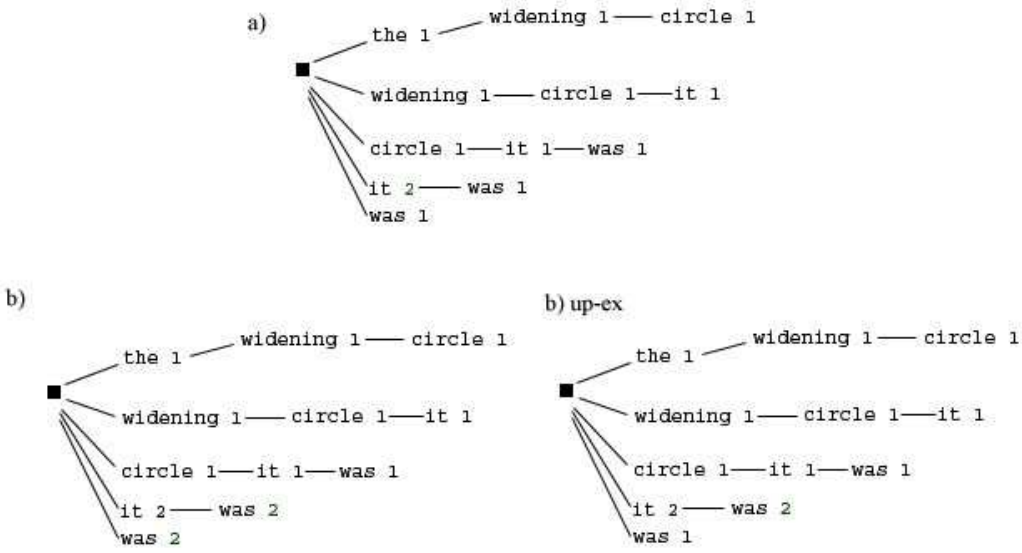

Figure 2: Update Exclusion

`eldest`
`one`
`$` (end),
into the trie (with update exclusion) we get Figure 3.

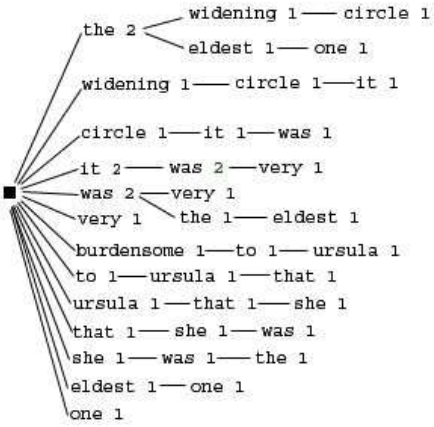

Figure 3: Finished trie of `text2.sf`

Generally, one ignores sentence breaks (except for resetting contexts), though it occurred to us that including the beginnings of sentences as actual nodes in the trie would provide an easy way to store information about what words tended to come first, or at the beginning of, a sentence. Thus we eventually included the option of considering '`$`' symbols at the beginning of sentences to be real words, and inserting them (and contexts they form) into tries. This did help compression, as is dicussed in Section 3.3.

3.2 PPM

The PPM algorithm uses a trie to predict the probability of a word (*i.e.* in data to be compressed) given the string of words that has preceded it, its context. If we are using depth N, it considers not only the conditional frequency of (say) x_N given the preceding context $x_1 \cdots x_{N-1}$, but the conditional frequency given each smaller context $x_{>1} \cdots x_{N-1}$ as well, weighing them by how much information they contain.

To accomplish this, PPM effectively adds an 'escape word count' α (not to be confused with a Dirichlet hyperparameter) to the sub-nodes of every node in the trie. Denote nodes in the trie by the path to them, *e.g.* $[x_1 \cdots x_m]$. Let $S_{[x_1 \cdots x_m]}$ be the sum of the frequencies of sub-nodes of node $[x_1 \cdots x_m]$ (*i.e.* the total number of words seen following the context $[x_1 \cdots x_m]$), and $F_{x_{m+1}|[x_1 \cdots x_m]}$ be the frequency of word x_{m+1} following context $[x_1 \cdots x_m]$. Then, using depth N, if we have seen $x_1 \cdots x_{N-1}$, PPM recursively defines the probability of the next word x_N to be $P_{N-1}(x_N)$, where

$$\begin{aligned} P_n(x_N) &= \frac{F_{x_N|[x_{N-n} \cdots x_{N-1}]}}{S_{[x_{N-n} \cdots x_{N-1}]} + \alpha} + \frac{\alpha}{S_{[x_{N-n} \cdots x_{N-1}]} + \alpha} P_{n-1}(x_N) \quad 1 \le n \le N \\ P_0(x_N) &= \frac{F_{x_N|[\,]}}{S_{[\,]}}. \end{aligned} \tag{31}$$

By $[\,]$, we mean the root node of the trie, unconditional frequencies or sums of frequencies. A generalization of this is to let α depend on n. Then we would just need the first line of (31), with the understanding that $\alpha_0 = 0$. But for the N-gram language model used here, we assume $\alpha_n \equiv \alpha$ for $n \ge 1$.

Notice that PPM gives perfectly normalized probabilities. It just additively mixes the conditional probabilities $F_{x_N|[x_{N-n} \cdots x_{N-1}]}/S_{[x_{N-n} \cdots x_{N-1}]}$ for different values of n with mixing factors $\frac{S_\cdot}{S_\cdot + \alpha}$ and (escape probability) $\frac{\alpha}{S_\cdot + \alpha} = 1 - \frac{S_\cdot}{S_\cdot + \alpha}$. (*Cf.* how α_{new} was used.) Note that α need not be an integer, but can take any nonnegative real value.

We heard mention of a possible modification to PPM, where instead of bare frequencies $F_{x_N|[x_{N-n} \cdots x_{N-1}]}$ we use modified frequencies $F_{x_N|[x_{N-n} \cdots x_{N-1}]} - \beta$, $0 \le \beta \le 1$. We tried to implement this and optimization tests appeared to suggest that β should be zero, *i.e.* nonexistant. However, it now occurs to us that the lack of normalized probabilities when implementing this directly is a definite problem, and that the conditional sums need to be modified as well as the frequencies, $S_{[x_{N-n} \cdots x_{N-1}]} \to S_{[x_{N-n} \cdots x_{N-1}]} - (\text{no. of distinct subnodes}) \cdot \beta$. The sub-normalized probabilities obtained without this could explain why $\beta = 0$ was preferred. It would be a good idea to explore this corrected algorithm in the future. Thankfully, all our N-gram model tests that did not vary β and all mixture model tests used the "suggested" $\beta = 0$, equivalent to not including β in the first place, which was not incorrect.

New words were dealt with in the N-gram model as in the topic model, either ignored, or included via an escape word count α_{new} – *after* the PPM probability was calculated. So the *total* PPM probability of a old word was modified by $A/(A+\alpha_{new})$ and the probability of a new word was $\alpha_{new}/(A+\alpha_{new})$, A being the number of distinct words already known. Both uniform (incorrect) and bits-per-character (more properly normalized) new word probabilities were used; see the discussion in the next section.

Another way of accounting for new words would have been to add escape counts at level of the trie via PPM. While this could be explored later, the α_{new} method as we used it is not necessarily wrong – and was really the only way to do things (it seems) in the mixture model.

3.3 Compression tests and optimization

As has been more or less assumed throughout this section, compression tests with the N-gram model proceeded in two phases, much like tests with the topic model. During the learning/training phase the model created a trie from training data set(s), and during the compression phase the model tried to compress documents with probabilities generated by PPM. The compression formula (1) lends itself perfectly to word-by-word, previous context-dependent probabilities. The relevant previous context is of length $\leq N-1$, and gets reset at the beginning of every sentence.

One optional modification (in fact, quite a good one) is to allow the model to keep learning during compression, adding information to the trie the usual way, after every word in the compression data set is read.

The parameters we tried to optimize during testing were:

- PPM parameters α and β[12]
- whether or not to learn during compression
- whether or not to count beginning-of-sentence markers as words (as explained above)
- the depth N
- using an α_{new} (overall) escape count for new words.

The tests we performed are by no means comprehensive, but more could easily be performed in the future as these compressions tend to be quite fast. We will try to discuss some of the main conclusions we came to. Every recorded test is tabulated and cross-referenced with a list (and plot in *Mathematica* notebook `optimization1.nb`) in Appendix B.3.2.

$N = 3$ was the standard trie depth for most tests. The first series of tests, in lists 1-6, established that we should always have $\beta = 0$, since this was always a minimum. Together with lists 7-17, they provided a good idea of what values of α we could expect from various pairs of learning and compression data sets. Much as with some of the continuous parameters of the topic model, the compression curves in the α direction increased sharply below their minimum, and increased much more gradually (and usually linearly) above it. Hence it is better to overshoot than to undershoot. The second derivative at the minimum was much lower for pairs of sets with high optimized α's. We decided that if it were needed, a standard value of $\alpha = 200$ would not be too bad at all.

For tests 1-17, compression-time learning and sentence markers as words were not yet used. Tests 18 and 19 were done to check values of self-compression (note the

[12]Really only α, as per the comments in the previous section.

tendency toward very small α). Learning during compression was finally investigated in tests 20-23. Comparing these with tests 15, 13, 10, and 11 (respectively), we see that using learning seems to improve compression by about 0.2 bits per word, and reduces the optimal value of α, perhaps by 20% or so. Neither of these is a surprising result – we would expect learning to make the trie more closely related to the compression data, producing both effects.[13]

Tests 24-27 add the feature of counting sentence markers as words at the beginning of every sentence. Again, this seems to help, except for the `critique_l` → `practical_c` test (a mistake?). We would hope to be able to better predict initial words of sentences. On the other hand, the inclusion of these markers must be considered carefully, for they do become the predominant 'word' in the data, and compressions could be lower just because all the other words are getting washed out. This should be investigated further. One important observation is that the optimal α remains virtually unchanged by the addition of this feature, so for optimization purposes it doesn't matter whether we use it.

The depth N was then tested in 28 and 30. It is difficult to reach definite conclusions with these few tests, although it seems that $N = 3$ is generally not too bad. For the two pairs of data sets shown, testing higher N was not possible due to computer memory limits.[14] `critique_l` → `practical_c` has a minimum at $N = 4$, and `cvwce_hl` → `enron_c` probably at $N = 6$, but neither of the minimum compression values (probably) differ from the $N = 3$ values by more than 0.04 or so bits/word. (Though not shown here, $N = 2$ and especially $N = 1$ *did* appear to be significantly worse.) Note that for these tests as well, the optimal α was independent of N – not quite predictable behavior this time.

Finally, we looked at α_{new}, with both uniform and corrected new word probabilities, as in the topic model. As before, using the unnormalized uniform probabilities gives compressions that are (generally) too low, and overly dependent on the percentage of new words in the compression data. The 2 bits-per-character new word probabilities correct the problem nicely, giving realistic compressions. The shape of the plots of α_{new} resemble those of α and of the topic model parameters (suggesting that we overshoot rather than undershoot, etc), and overall a standard value of $\alpha_{new} \approx 600$ would seem acceptable.

Before we move on, note that N-gram compressions tend to be lower than those for the topic model. We shall see a bit more of this in the next section with the mixture model, combining the two. There are many more tests that can and should be done with the N-gram model, especially to try to establish independence of its various parameters (only α vs various others was really tested). Whether adding sentence markers actually helps is not entirely decided; we hope they did not obscure results of the tests done after their inclusion. And of course it would be useful to implement and optimize a corrected version of the PPM β parameter.

[13]For these and the remaining tests, we tried to use a smaller representative set of pairs of text files, varying in relatedness, but none too unrelated since we do expect training text to be somewhat similar to text that is to be compressed.

[14]This, however, was with 5k of RAM; testing on other computers may be better.

4 The Multiplied Mixture Model

The final model we looked at is a 'multiplicative' mixture of the topic and N-gram models. Both presumably provide different information about word probabilities. Although the topic model was functioning more like a 2-topic [old words (from training text) + new words (from compression text)] model, it could still help out the N-gram. We multiplied the two, motivated by the idea that (*e.g.*) using a topic almost exclusively should eliminate other topics from the N-gram model (and vice versa). During learning, both models were trained (individually) on the same training data, forming a T-topic beta matrix and a trie. Then, during compression, prior to reading each word from compression data, the 'next-word' probability distributions predicted by the topic and N-gram models were multiplied word-wise and renormalized. That is, given what we already know about the compression data ($\boldsymbol{\theta}$ distribution, previous context, etc), if the topic model predicts a next-word probability distribution $P_{TM}(a)$, and the N-gram model $P_{NG}(a)$, we compute

$$P_{mix}(a) = \frac{[P_{TM}(a)]^{1-\gamma}[P_{NG}(a)]^{\gamma}}{\sum_b [P_{TM}(b)]^{1-\gamma}[P_{NG}(b)]^{\gamma}}, \tag{32}$$

with a continuous parameter $0 \leq \gamma \leq 1$. Again we emphasize that

$$P_{TM}(a) = 0 \text{ or } P_{NG}(a) = 0 \quad \Rightarrow \quad P_{mix}(a) = 0. \tag{33}$$

This is the eliminating behavior we sought to achieve.

4.1 Parameters and tests

Due to (33), either both or neither of the component models should learn during compression; for any new words that exist in one but not the other will just have 0 probability in the mixture. Partly because the topic model naturally (as we implemented it) learns during compression, and partly due to the positive results of compression-time learning in the N-gram model, we decided to have both models actively learn from every new word that is read. The limits $\gamma \to 0$ and $\gamma \to 1$ of the mixture model do correspond to pure topic and N-gram compression, respesctively, but due to the way these limits are taken we must be careful in comparing especially the latter to the learning N-gram model of the previous section. The normal learning N-gram model updates its trie and its PPM probabilities every time a word of compression data is read. The $\gamma \to 1$ mixture model updates its trie with every new word, but this only affects probabilities once the topic model knows about these new words as well – *i.e.* after every new document. The compressions of the two N-gram models will not be exactly the same.

Sentence markers were *not* counted as words in (almost) any of the mixture tests; a few exceptions are listed in Appendix B.3.3. This is because their effect in N-gram compression (and especially on other N-gram parameters) was not fully understood, and because we wanted to reuse the non-`$` beta matrices from the topic model. In addition, new words were completely ignored in the tests done here, since we hoped (based on some past topic and N-gram tests) that once other parameters were understood α_{new} might be varied more or less independently later on.

We used the standard $N = 3$ throughout, and either $T = 5$ or $T = 1$, as the number of (old) topics seemed to make no difference. These simplifications, however, still left the following continuous parameters to be dealt with:

- the new mixture parameter γ
- the topic model parameters μ and Tu
- the N-gram parameter α.

Following what has already been said about the N-gram model, we set $\beta = 0$.

Due to the need for calculating and renormalizing complete probability distributions for every word of compression data, the mixture model was usually very slow to test. We were not able to do as many multi-dimensional optimization tests a we would have liked, and were limited to using some rather short learning and compression texts. The three pairs of files we looked at, hoping the would be representative (and not too short!), were `china_c` → `china_tr`, `china_c` → `vicomte_tr`, and `critique_tr` → `practical_tr`.

We first did a series of "sequential" tests with each pair, starting with what we thought were standard values of the four main parameters and optimizing each in turn, using newly found optimal values while optimizing the next parameter. These appear in lists 8-17 of `optimization3.nb` and Appendix B.3.3.

No matter what the other parameters are, the plots of compression vs. values of γ always tended to look like smooth "depressions" between the topic and N-gram endpoints, roughly symmetric about their minima; they could probably be well-approximated by parabolas.[15] This is highly fortunate, for it means the mixture we are using generally does improve things!

Regarding the other parameters, the compression at the minimum of the γ plots seems much more highly affected by the values of μ and α used than the endpoints. A much higher μ and a lower α (than for the pure topic and N-gram models, respectively) are generally favored by the mixture, perhaps supporting a theory that, in the mixture model, the topic component deals more with new words and the N-gram component deals more with old (trained) ones. This claim is further supported by the different optimal γ's for the three pairs of data; more similar learning-compression pairs seem to have higher γ's.[16]

Note in general the $\gamma \to 0$ (topic) compression is higher than the $\gamma \to 1$ (N-gram), and that the optimal value of γ tends to be ≥ 0.5. The number of topics was varied between 1 and 5 in a few of the tests, with no noticible difference.

Throughout the sequential tests, we tried to check whether the optimal values of various parameters where changed by various other ones – *i.e.* whether γ, μ, Tu, and α were independent, and could be independently optimized. (These small tests are not in `optimization3.nb` or Appendix B.3.3.) They all did *seem* to be roughly independent, motivating the next set of tests.

[15] They are reminiscent of melting point depression graphs for mixtures of liquids.

[16] Also see the file ∼`/Dasher_Devel/languagemodel/testing/output/thetas_mix`, as in Appendix B.4.

The next tests we did were a series of "blind" optimization tests, on `china_c` → `china_tr` and `china_c` → `vicomte_tr` (similar and dissimilar pairs). Like the sequential tests, parameters were optimized one at a time; but this time the values of the parameters not being optimized were fixed from the beginning (rather than updated as optimal values were found). Note that we used two different sets of values of fixed parameters for `china_c` → `china_tr` tests, the first some more or less standard values suggested by the seqential tests, and the second resembling the initial values for the `china_c` → `china_tr` sequential test.

The blind tests performed sometimes better and sometimes worse than the sequential ones. Interestingly, a good indicator of how well one test or the other seemed to be doing was whether μ and Tu had relatively small minima (good) or liked to escape to very large values. The sequential `china_c` → `china_tr` and the blind `china_c` → `vicomte_tr` tests both had such large values, and performed worse than their respective blind and sequential countarparts. Such behavior *may* be caused by holding fixed a value of γ too far from its real (*i.e.* optimized in 4 dimensions) minimum. During the sequential tests, this in turn may have been caused by using a value of α that was too large while optimizing γ. These interdependencies show that the four parameters we looked at are *not* completely independent.

For some final comparisons, we calculated mixture model compressions for our three pairs of texts with what we thought could be "standard" parameters: $\gamma = 0.7$, $\mu = 25$, $Tu = 25$, and $\alpha = 20$. (These were picked much the same way as standard parameters in the other two models, looking at shapes of graphs and increase in compression away from minima.) We also tried to optimize pure topic model and pure N-gram compressions for our data sets, as shown in Appendix B.3.3.[17] Table 1 compares the standard mixture model values to the best optimized compressions obtained via sequential or blind tests, and to the optimized pure component values. Although we must remember that these are by no means thorough tests, it is nice to see the mixture model outperforming the others, and the standard parameter compressions not being too far from the (quasi) optimized ones.

text pair	**std mixture**	**opt mixture**	**opt topic**	**opt N-gram**
`china_c` → `china_tr`	7.881	7.861	8.816	7.991
`china_c` → `vicomte_tr`	8.092	8.030	8.652	8.567
`critique_tr` → `practical_tr`	7.125	7.125	8.148	7.341

Table 1: Comparisons Across Models

[17]The endpoints of an optimized mixture model do not correspond to optimized or even standard-parameter component models. And using optimized N-gram parameters, the $\gamma \to 1$ mixture endpoint does not perfectly correspond to a pure N-gram model due to new-word learning, as noted above. So for a good comparison it was necessary to redo these pure model tests.

5 Some concluding remarks and suggestions

The mixture model tests we have done provide excellent evidence that a general multiplication scheme of topics and N-gram PPM could work very well. The multiplied mixture outperforms both its components, even without a fully functional topic model.

One of the main disadvantages of the mixture model is that it was slow and difficult to test, probably due to computing a full probability distribution for every word of compression data. The full distribution is computed solely to obtain the normalization constant in (32). It would be useful to find a way to guess this normalization without having to calculate it.[18]

We have observed that the current topic model is flawed, in that it does not seem to differentiate well between topics. This of course could also be investigated in the future[19]; or perhaps one could decide that the simple [1 old topic + 1 new topic] model is sufficient to work with. Likewise, the use of the PPM β parameter in N-gram compression may be corrected and tested, as $\beta = 0$ may not necessarily be the best choice.

In test/list 2 of `optimization3.nb` (see the end of Appendix B.3.3), we briefly tried to look at a monogram $\times$ trigram model, to compare its behavior with the topic $\times$ trigram model, checking if perhaps the mixture model only outperforms N-gram because the topic model provides a bit of pure monogram data that the trigram lacks. (It only later became clear that the flawed topic model had the other advantages we have discussed.) The preferred γ was almost or exactly equal to 1.0, favoring almost exclusively trigram probabilities, quite different from the behavior of the topic $\times$ trigram tests.

In addition to improvements of the models via adding more parameters, learning topics better, etc, one might also consider building a mixture model a completely different way. One possiblity would be to include something like a trie in every topic, either previously defined, or to be learned. If learned, note that it could not be an actual trie holding frequency data; for how could word frequencies possibly be learned? It seems that one might nevertheless be able to include context-dependent probability distributions in each topic (rather than context-independent as we have used), and learn these. Replacing PPM by the Bayesian approach to N-gram models presented in [MBP] may be useful or indeed necessary.

Ultimately, once one obtains a satisfactory word-level model – and perhaps we have one already, with one topic and no β – one would still have to do a little work, mostly programming, to extract from it character-level probabilities that could be used in Dasher. We hope that the tests and models examined here have been helpful, and that the code that has been written, albeit at word-level, may still be of some use. The code is briefly reviewed in Appendix C.

Acknowledgements

[18] If one were interested in doing this, the `sum` function in the HashList class could provide a starting point; just call the function within the loop of `FileStat::calc_AT_NT`.

[19] If so, both the $\boldsymbol{\theta}$-displaying option in `superheader.h` (see Appendix B.4) and the suggestions for better algorithms in [ML] and [BlNJ] could be useful.

Many thanks to David MacKay, Phil Cowans, and the summer 2005 Dasher Team for ideas, help, and encouragement.

A Formulae for inferring hyperparameters

Here we want to justify equations (27) and (28). The following derivation uses quite a few approximations, but in the end it *does* seem to give a comfortable answer. (It would nevertheless be helpful to find a better way to do this.)

First recall that the Digamma function is defined as

$$\Psi(x) = \frac{d}{dx}\log\Gamma(x) = \frac{\Gamma'(x)}{\Gamma(x)}. \tag{34}$$

When the argument $x \gtrsim 0.1$, a decent approximation is

$$\Psi(x) = \log x - \frac{1}{2x} + O\left(\frac{1}{x^2}\right). \tag{35}$$

Now consider our problem. The topic model with hyperparameters says:
- there exist fixed hyperparameters $\mathbf{u} = \alpha\mathbf{m}$ and a topic matrix $\boldsymbol{\beta}$
- select a $\boldsymbol{\theta}^d$ from $P(\boldsymbol{\theta}|\mathbf{u})$ for each document
- select a topic for each word from $P(t|\boldsymbol{\theta})$
- select a word from each topic given $\boldsymbol{\beta}$, *i.e.* $P(x|t)$,

where by the first statement we mean that when inferring $\mathbf{u}$ and $\boldsymbol{\beta}$, we should use a noninformative prior. Suppose that we already know $\boldsymbol{\beta}$, and that we have a collection of "well-formed" documents, which are long enough so that we can be sure of their $\boldsymbol{\theta}$'s.[20] We are interested in

$$\begin{aligned} P(\mathbf{u}|\mathbf{x},\boldsymbol{\theta},\boldsymbol{\beta}) &= P(\mathbf{u}|\boldsymbol{\theta}) \\ &= \frac{P(\boldsymbol{\theta}|\mathbf{u})P(\mathbf{u})}{P(\boldsymbol{\theta})} \\ &\sim P(\boldsymbol{\theta}|\mathbf{u}) \end{aligned} \tag{36}$$

since the prior on $\mathbf{u}$ is noninformative. The likelihood $P(\boldsymbol{\theta}|\mathbf{u})$ is

$$\begin{aligned} P(\boldsymbol{\theta}|\mathbf{u}) &= \prod_d P(\boldsymbol{\theta}^d|\mathbf{u}) \\ &= \frac{1}{Z(\mathbf{u})^D}\prod_d (\theta_1^d)^{u_1-1} + \ldots + (\theta_T^d)^{u_T-1}, \end{aligned} \tag{37}$$

and $Z(\mathbf{u}) = (\prod_j \Gamma(u_j))/\Gamma(\alpha)$.

[20] Assuming that we know $\boldsymbol{\beta}$ indicates that we really shouldn't be using any formulae derived here on topic-learning data....

To get the maximum likelihood, we can differentiate the log of (37) with respect to each u_j,

$$\begin{aligned}\frac{\partial}{\partial u_j}\log P(\boldsymbol{\theta}|\mathbf{u}) &= \frac{\partial}{\partial u_j}\left(D\log\Gamma(\alpha) - D\sum_k \log\Gamma(u_k) + \sum_{d,k}(u_k-1)\log\theta_k^d\right)\\ &= D\Psi(\alpha) - D\Psi(u_j) + \sum_d \log\theta_j^d\\ &= 0,\end{aligned}$$

or

$$\Psi(u_j) - \Psi(\alpha) = \log\left(\prod_d \theta_j^d\right)^{1/D} \quad \forall\, j. \tag{38}$$

It is too bad this does not have an exact solution. From here, we assume that $u_j, \alpha \gtrsim 0.1$; given $\boldsymbol{\theta}$'s that actually appear during compression, this does seem to be the case most of the time – though it does not hold during topic learning.[21] Using (35), we write (38) as

$$\begin{aligned}&\log u_j - \log\alpha - \tfrac{1}{2u_j} + \tfrac{1}{2\alpha} = \log\left(\textstyle\prod_d \theta_j^d\right)^{1/D}\\ \Rightarrow\quad &\log m_j - \tfrac{1}{2\alpha}\left(\tfrac{1}{m_j} - 1\right) = \log\left(\textstyle\prod_d \theta_j^d\right)^{1/D}.\end{aligned} \tag{39}$$

First make the approximation that $m_i \approx 1/T$, which is usually not too far from the truth, and also suggests that we should have $\left(\prod_d \theta_j^d\right)^{1/D}$ roughly independent of j. Then from (39), we obtain

$$\begin{aligned}&\log\tfrac{1}{T} - \tfrac{1}{2\alpha}(T-1) = \log\left[\left(\textstyle\sum_j\left(\prod_d \theta_j^d\right)^{1/D}\right)/T\right]\\ \Rightarrow\quad &\alpha = -\frac{T-1}{2\log\left(\sum_j\left(\prod_d \theta_j^d\right)^{1/D}\right)},\end{aligned}$$

which is precisely (28). Then notice that especially for large T, so that the m_j are small, (39) also suggests that $m_j \sim \left(\prod_d \theta_j^d\right)^{1/D}$. Since the m_j add up to 1, the natural solution then is

$$m_j = \frac{\left(\prod_d \theta_j^d\right)^{1/D}}{\sum_j\left(\prod_d \theta_j^d\right)^{1/D}},$$

which is (27).

Again, this derivation is not quite rigorous; but the solutions *do* make sense. $\mathbf{m}$ is a component-wise geometric average of the $\boldsymbol{\theta}^d$, which is quite reasonable. If the $\boldsymbol{\theta}^d$ tend to extreme distributions concentrated on random single topics, the m_j can still be roughly $1/T$, while the actual normalization constant $\sum_j\left(\prod_d \theta_j^d\right)^{1/D}$ will be very small, hence so will α. On the other hand, if the $\boldsymbol{\theta}^d$ tend to be uniform, $\left(\prod_d \theta_j^d\right)^{1/D}$ can be about $1/T$, so the argument of the log can close to 1, making α very large. α behaves as we would expect for a Dirichlet parameter.

[21] Although (27) and (28) do seem reasonable for wider ranges of hyperparameters.

B Summary of data

This appendix was constructed in the hope of clarifying what data and result files exist, and where they can be found. The `~/` directory here (and in the next appendix) refers to `tdd23`'s home directory on `flotta.ra.phy.cam.ac.uk`.

B.1 Texts

Almost all the text files used for testing came from Project Gutenberg online. The one notable exception is the Enron email corpus, also found online. All files below are found in `~/Dasher_Devel/languagemodel/testing/`; for actual use with the various language model programs, the formatted `filename.sf` versions must be used. (See Section C.1 for further information.)

Each file name contains a main descriptive word, plus one or several letter designations after an underscore. The main word indicates what (larger) work the text came from, as follows:

- `critique` – Kant's *Critique of Pure Reason*
- `practical` – Kant's *Critique of Practical Reason*
- `china` – Rousseau's *The Problem of China*
- `vicomte` – Dumas' *Le Vicomte de Bragelonne* (in English)
- `women` – DH Lawrence's *Women in Love*
- `aaron` – DH Lawrence's *Aaron's Rod*
- `enron` – the Enron email corpus
- `cvwc` – parts of `critique`, `vicomte`, `women`, and `china`
- `cvwce` – parts of `critique`, `vicomte`, `women`, `china`, and `enron`.

The letters following an underscore indicate:

- `l`: to be used for learning; usually the first 2/3 of a file (except in the case of `enron` and the two composite files)
- `c`: to be compressed; usually the last 1/3 of a file, except as above, though not always used for compression
- `tr`: 'truncated', a very short section, usually part of the `_l` file
- `h`: 'half', half of another file
- `q`: 'quarter', a quarter of another file.

With all this in mind, we can list the

Available Data Files

filename	# words	# sentences	# documents	vocab size
critique_l	142934	4088	751	5197
critique_c	66603	1769	357	4202
critique_tr	25415	726	117	2665
china_l	40418	1653	250	5378
china_c	26548	1123	161	3934
china_tr	951	29	5	395
vicomte_l	125901	6070	1626	8364
vicomte_c	65041	3804	821	5962
vicomte_tr	3614	244	54	967
practical_l	41184	955	218	3095
practical_c	65041	3804	107	2469
practical_tr	3723	99	22	868
women_l	122719	8850	1626	8982
women_c	63017	4715	875	6270
aaron_l	79950	6569	1105	7235
aaron_c	36858	2976	447	4267
cvwc_l	431972	20661	4253	17036
cvwc_hl	226859	10420	2163	13142
enron_l	199999	10661	111	13646
enron_hl	110198	5878	67	10221
enron_c	40001	2147	29	5911
cvwce_hl	278201	13143	2186	16510
cvwce_ql	143989	6792	1042	12400

Note that these all have a minimum of 40 words in every document. The vocabulary size (number of distinct words) has not been documented for all of them. We expect texts by the same author to be quite similar. The works by DH Lawrence contain much more conversation (hence significantly shorter documents, created at paragraph breaks) than those by Kant. All document boundaries were created automatically, except those in the Enron data, which did not have detectable paragraph breaks; documents there correspond to individual emails. The mixed (`cvwc` and `cvwce`) data were generally used for topic learning, and then individual documents were compressed with the trained model.

In addition to the above, there are a few short files that were used for code testing, which we should mention should they be of further use:

	w	s	d	vocab
text2	17	3	1	13
sampletext	607	45	9	268
china_trd	166	6	1	107
short	9	1	0	9
empty	0	0	0	0

`text` and `sampletext` are both (very) short excerpts from DH Lawrence's *The Rainbow*, and `china_trd` is the first document of `china_tr`.

B.2 Beta matrices

The bete matrices cited here were calculated as explained in Section 2.5.1. The distance functions are

$$\Delta(\boldsymbol{\beta}, \tilde{\boldsymbol{\beta}}) = \sqrt{\sum_{a=1}^{A}\sum_{j=1}^{T}(\beta_{aj} - \tilde{\beta}_{aj})^2}, \tag{40}$$

$$\Delta(\boldsymbol{\theta}, \tilde{\boldsymbol{\theta}}) = \sqrt{\sum_{d=1}^{D}\sum_{j=1}^{T}(\theta_j^d - \tilde{\theta}_j^d)^2}. \tag{41}$$

These should scale directly with T and D, respectively, but it also appears that there is some scaling with the other variables, especially in the case of $\Delta\boldsymbol{\beta}$ scaling with A. This is because $\boldsymbol{\beta}_{\cdot j}$ is not some roughly uniform distribution over all words for each j, but a decently sharply peaked one (with, *e.g.* , 'the' and 'of' and 'to' right at the top). So we have not yet found a way to define a scale-independent version of these differences. Nonetheless, (40) and (41) between the respective values of $\boldsymbol{\beta}$ and $\boldsymbol{\theta}$ in the final two iterations of each beta matrix calculation are shown below.

(One decent use of these values was to moniter them during algorithm iterations in order to see how convergence was coming along. Though this data was not recorded anywhere, we should note that $\Delta\boldsymbol{\beta}$ and $\Delta\boldsymbol{\theta}$ between subsequent iterations generally *did* decrease – fast enough to indicate that some convergence certainly was happening – but did *not* do so monotonically.)

The beta matrices are stored in `.bmx` files which the `BetaMatrix C++` class can read, in the directory `~/Dasher_Devel/languagemodel/testing/`. The main word in the name of each one refers to the text file from which it was created, and the number after the last underscore is the number of topics which it contains. Beta matrices with only one topic contain an exact monogram distribution.

Beta matrix files (all with extension `.bmx`)

name	# iterations	last $\Delta\boldsymbol{\theta}$	last $\Delta\boldsymbol{\beta}$
critique_l_1	-	-	-
critique_l_5	70	0.16	0.005
critique_l_10	80	0.14	0.008
critique_tr_1	-	-	-
critique_tr_5	400	0.0094	0.00012
china_c_1	-	-	-
china_c_5	200	0.011	0.000043
china_c_10	70	0.021	0.00033
china_c_15	50	0.049	0.00096
china_c_20	50	0.052	0.0014
enron_hl_1	-	-	-
enron_hl_5	400	0.00011	0.00000025
enron_hl_10	400	0.011	0.000015
enron_hl_15	400	0.0046	0.000049
enron_hl_20	400	0.0013	0.00025
cvwce_ql_1	-	-	-
cvwce_ql_5	40	0.35	0.00085
cvwce_ql_10	30	0.45	0.0034
cvwce_ql_15	40	0.30	0.0014

B.3 Test reference

The results of most of the compression tests that were performed are stored in lists in three *Mathematica* notebooks, `optimization1.nb`, `optimization2.nb`, and `optimization3.nb`, in `~/Dasher_Devel/languagemodel/testing/output`. There are usually graphs below each list, which can be easily manipulated; see in particular `Built-in Functions > Graphics and Sound > Basic Options` and `3D Options` in the *Mathematica* Help Browser. Here we summarize these tests, and reference where they can be found.

B.3.1 Topic model tests

We begin with pure Topic Model tests. The main parameters varied were u and u_{nd} (the better parameters Tu and μ were not yet in use). The value of T appears in the name of the beta matrix used. The number of $\boldsymbol{\theta}$-learning iterations used per word during compressions has its own column. α_{new} was usually turned off and new words were ignored; this appears as -1 in the chart below. When α_{new} *was* used, uniform new word probabilities of $\alpha_{new}/(A+\alpha_{new})$ were usually employed (the mistake in this was not yet realized), unless indicated by "strict" in the chart (then 2 bits per new word character were taken into account). Sentence markers were never calculated, so the option does not appear.

Best values (giving minimum compression) in each case are indicated after a '/', as is the compression at this minimum. See however Section 2.5.2 for a better discussion of the optimization results in these tests.

Topic Model Tests, in `optimization2.nb`

some general tests

.bmx file	comp. file	θ its.	list	u	u_{nd}	α_{new}	comp.
critique_l_5	critique_c	10	1	$0-100/0$	$0-100/0$	-1	8.442
critique_l_5	critique_c	10	2	$0-25/5$	$0-25/25$	-1	8.439
critique_l_5	critique_c	10	3	$0-6/1$	$0-40/8$	-1	8.423
critique_l_5	critique_c	5	6	$0-8/1$	$0-40/10$	-1	8.430
critique_l_5	critique_c	5	7	7	$30-70/40$	-1	8.446
cvwce_ql_5	enron_c	5	4	$0-25/0$	$0-25/0$	-1	9.592
cvwce_ql_5	enron_c	5	5	$0-5/0$	$0-25/0$	-1	9.592
cvwce_ql_5	enron_c	20	22	$0-5/1$	$0-25/5$	-1	9.580
china_l_5	women_c	5	8	$0-10/1$	$0-100/50$	-1	8.920
china_l_5	women_c	20	9	$0-10/1$	$0-100/50$	-1	8.919

tests for number of iterations

.bmx file	comp. file	θ its.	list	u	u_{nd}	α_{new}	comp.
china_c_5	china_tr	5	10	$0-15/4$	$0-500/0$	-1	8.802
china_c_5	china_tr	40	11	$0-7/7$	$0-7/3$	-1	8.796
china_c_5	china_tr	5	12	$0-7/7$	$0-7/3$	-1	8.796
china_c_5	china_tr	10	13	$0-7/4,7$	$0-7/2,3$	-1	8.796

tests for importance of T, and how multiplier scales with T

.bmx file	comp. file	θ its.	list	u	u_{nd}	α_{new}	comp.
china_c_1	vicomte_tr	10	18	$0-7/2$	$0-300/5$	-1	8.653
china_c_5	vicomte_tr	10	14	$0-7/1$	$0-300/10$	-1	8.653
china_c_10	vicomte_tr	10	15	$0-7/1$	$0-300/20$	-1	8.657
china_c_15	vicomte_tr	10	16	$0-7/1$	$0-300/20$	-1	8.660
china_c_20	vicomte_tr	10	17	$0-7/1$	$0-300/\ 50$	-1	8.664

tests for α_{new}

.bmx file	comp. file	θ its.	list	u	u_{nd}	α_{new}	comp.
china_c_5	vicomte_tr	10	19	1	8	$0-3000/500$	8.150
critique_l_5	critique_c	10	20	1	5	$0-3000/150$	8.392
cvwce_ql_5	enron_c	10	21	1	5	$0-3000/700$	9.379
china_c_5	vicomte_tr	10	23	1	8	strict $0-3000/500$	9.790
china_c_5	vicomte_tr	10	25	3	15	strict $0-3000/500$	9.809
critique_l_5	critique_c	10	24	1	5	strict $5-300/150$	8.744

B.3.2 N-gram model tests

For the N-gram model, the parameters being tested are listed in Section 3.3. The results are summarized in the same manner as the topic model tests. We use the convention that $\alpha_{new} = -1$ means new words are being ignored, and "strict" means the 2-bits-per-character new word probabilities are being used. The columns marked "N," "learn," and "`$`" refer to the trie depth, whether there is compression-time learning, and whether beginning-of-sentence markers are included as words, respectively. The full lists appear in `optimization1.nb`, as discussed at the beginning of Section B.3.

NOTE: that the first 12 tests were not initially properly normalized, calculating bits per word by dividing by the total size of the compression file rather than (total

size − words skipped because they were new). This affected the compressions slightly, but not the optimization of parameters. The raw data appears in `optimization1.nb`, but so does corrected data in lists below it; the compressions listed in the table here are all corrected.[22]

N-Gram Model Tests, in `optimization1.nb`

general, varying both α and β

train. file	**comp. file**	**N**	**list**	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	**learn**	**$**	$\boldsymbol{\alpha_{new}}$	**comp.**
critique_l	critique_c	3	1	10-100/100	0-0.9/0	n	n	-1	7.220
critique_l	critique_c	3	3	25-250/75	0-0.4/0	n	n	-1	7.220
critique_l	critique_c	3	5	70-106/86	0-0.4/0	n	n	-1	7.219
critique_l	women_c	3	2	10-100/100	0-0.9/0	n	n	-1	9.340
critique_l	women_c	3	4	25-250/250	0-0.4/0	n	n	-1	9.252
critique_l	women_c	3	6	250-295/295	0-0.4/0	n	n	-1	9.244

general, varying α with $\beta = 0$

train. file	**comp. file**	**N**	**list**	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	**learn**	**$**	$\boldsymbol{\alpha_{new}}$	**comp.**
critique_l	critique_c	3	7	70-94/85	0	n	n	-1	7.219
cvwce_hl	critique_c	3	8	0-240/140	0	n	n	-1	8.119
cvwce_hl	women_c	3	9	0-375/240	0	n	n	-1	8.665
cvwce_hl	enron_c	3	10	0-250/190	0	n	n	-1	9.049
cvwce_hl	vicomte_c	3	11	0-250/210	0	n	n	-1	8.572
cvwce_hl	china_c	3	12	0-375/255	0	n	n	-1	8.811
critique_l	practical_l	3	13	10-250/130	0	n	n	-1	7.548
women_l	aaron_l	3	14	10-250/160	0	n	n	-1	8.278
critique_l	critique_c	3	15	10-250/80	0	n	n	-1	7.219
cvwce_hl	women_c	3	16	10-300/230	0	n	n	-1	8.665
cvwce_hl	china_c	3	17	10-300/250	0	n	n	-1	8.811

as controls

train. file	**comp. file**	**N**	**list**	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	**learn**	**$**	$\boldsymbol{\alpha_{new}}$	**comp.**
vicomte_c	vicomte_c	3	18	20-200/$\ll 20$	0	n	n	-1	5.975 at 200
vicomte_c	vicomte_c	3	19	20-200/$\ll 20$	0	y	n	-1	5.954 at 200

to test compression-time learning

train. file	**comp. file**	**N**	**list**	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	**learn**	**$**	$\boldsymbol{\alpha_{new}}$	**comp.**
critique_l	critique_c	3	20	30-150/70	0	y	n	-1	7.054
critique_l	practical_c	3	21	70-200/90	0	y	n	-1	7.214
cvwce_hl	enron_c	3	22	130-230/150	0	y	n	-1	8.922
cvwce_hl	vicomte_c	3	23	150-250/160	0	y	n	-1	8.315

to test beginning-of-sentence 'words'

train. file	**comp. file**	**N**	**list**	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	**learn**	**$**	$\boldsymbol{\alpha_{new}}$	**comp.**
critique_l	critique_c	3	24	10-200/70	0	y	y	-1	6.962
critique_l	practical_c	3	25	10-200/100	0	y	y	-1	7.463
cvwce_hl	enron_c	3	26	10-200/160	0	y	y	-1	8.659
cvwce_hl	vicomte_c	3	27	10-200/150	0	y	y	-1	8.054

to test N

[22]The program `fs_compare.cpp`, described in Appendix C, was a useful tool for finding the correction ratios.

train. file	comp. file	N	list	α	β	learn	$	α_{new}	comp.
critique_l	critique_c	4	28a	10-200/70	0	y	y	-1	6.954
critique_l	critique_c	5	28b	10-200/70	0	y	y	-1	6.955
cvwce_ql	enron_c	3	29a	10-200/140	0	y	y	-1	8.590
cvwce_ql	enron_c	4	29b	10-200/140	0	y	y	-1	8.573
cvwce_ql	enron_c	5	29c	10-200/140	0	y	y	-1	8.563

to test new word probabilities/methods, at best α, if known

train. file	comp. file	N	list	α	β	learn	$	α_{new}	comp.
critique_l	critique_c	3	30	70	0	y	y	0-1000/125	6.965
critique_l	practical_c	3	31	100	0	y	y	0-1000/175	7.430
critique_l	enron_c	3	34	200 (guess)	0	y	y	0-1000/775	8.479
cvwce_hl	enron_c	3	32	160	0	y	y	0-1000/675	8.566
cvwce_hl	vicomte_c	3	33	150	0	y	y	0-1000/375	8.031
critique_l	critique_c	3	35	70	0	y	y	strict 0-1000/150	7.292
critique_l	practical_c	3	36	100	0	y	y	strict 0-1000/175	7.948
critique_l	enron_c	3	39	200 (guess)	0	y	y	strict 0-1000/775	9.827
cvwce_hl	enron_c	3	37	160	0	y	y	strict 0-1000/675	9.107
cvwce_hl	vicomte_c	3	38	150	0	y	y	strict 0-1000/375	8.355

B.3.3 Mixture model tests

The mixture model parameters and testing methods are described in Section 4.1. The full results of all mixture tests are in `optimization3.nb`, referenced with list numbers as usual.

For the "sequential" tests and checks, there was always compression-time learning, no sentence markers recorded as words, $N = 3$, no α_{new}, and 10 θ-iterations. Organized by pairs of files, we have:

`china_c` → `china_tr`

T	list	γ	μ	Tu	α	comp.
5	8	0-1/0.9	2.7	15	200	8.019
5	9	0.9	1.3-1333.3/666.6	15	200	7.985
5	10	0.9	1333.3	15	10-500/50	7.879

`china_c` → `vicomte_tr`

T	list	γ	μ	Tu	α	comp.
5	3	0-1/0.55	1	15	200	8.386
1	5	0-1/0.55	5	3	200	8.282
1	6	0-1/0.55	1	15	200	8.377
5	(missing)	0.55	?/ 27	15	200	8.21
5	11	0.55	27	0.5-50/12	200	8.210
5	12	0.55	27	12	10-500/20	8.036

`critique_tr` → `practical_tr`

T	list	γ	μ	Tu	α	comp.
1	13	0-1/0.7	20	15	20	7.129
1	14	0.7	15-100/25	15	20	7.128
1	15	0.7	25	5-100/20	20	7.128
1	16	0.7	25	20	15-100/20	7.128
5	17	0-1/0.7	25	20	20	7.125

Next, the sets of "blind" optimizations, with the same suppressed parameters as the sequential tests; the last line in each series is a compression with "optimized" parameters.

china_c → china_tr

T	list	γ	μ	Tu	α	comp.
5	18g	0-1/0.8	25	25	20	7.874
5	18m	0.7	1-100/1	25	20	7.850
5	18tu	0.7	25	5-100/5	20	7.856
5	18a	0.7	25	25	10-200/20	7.881
5	-	0.8	1	5	20	7.861

china_c → china_tr

T	list	γ	μ	Tu	α	comp.
5	20g	0-1/0.9	1	5	200	8.019
5	20m	0.9	1-100/100	5	200	7.991
5	20tu	0.9	1	5-100/100	200	8.014
5	20a	0.9	1	5	10-200/50	7.904

china_c → vicomte_tr

T	list	γ	μ	Tu	α	comp.
5	19g	0-1/0.6	25	25	20	8.036
5	19m	0.7	1-100/100	25	20	8.077
5	19tu	0.7	25	5-100/100	20	8.089
5	19a	0.7	25	25	10-200/30	8.085
5	-	0.6	100	100	30	8.111

Note that the number of topics T in each case determined which pre-calculated beta matrix file was used for the topic model component.

Pure topic model and 3-gram tests were done for comparison, with the same fixed parameters as above (learning, \$, θ iterations, etc), optimizing the various continuous parameters. Full lists are not available, but we can summarize the optimal parameters and compressions found.

Topic model:

beta mx	comp file	μ	Tu	comp.
china_c_5	china_tr	0.005 ± 0.025	1 ± 0.5	8.816
china_c_5	vicomte_tr	2.25 ± 0.25	3 ± 1	8.652
critique_tr_5	practical_tr	0.75 ± 0.1	7 ± 1	8.148

3-gram model:

learn file	comp file	α	comp.
china_c	china_tr	70 ± 10	7.991
china_c	vicomte_tr	90 ± 10	8.567
critique_tr	practical_tr	70 ± 10	7.341

A few other tests were done, which we can include here for completeness. They are all for the pair china_c → vicomte_tr. Lists 1 and 2 use the same fixed parameters as above, except for sentence markers. List 1 is a mixture model test with an N-gram component trained on \$ markers and a beta matrix *not* including them. List 2 is a multiplied 1-gram × 3-gram mixture model, both components trained on \$ markers. List 4 is a pure 3-gram test with no \$ markers, yielding the 3-gram result

cited immediately above.
List 1: γ =0-1/0.6, $\mu = 1$, $Tu = 15$, $\alpha = 200$; comp=8.119
List 2: γ =0-1/1, $\alpha = 200$; comp=8.307 (seems to not want any 1-gram)
List 4: α =10-400/90; comp=8.567.

B.4 $\boldsymbol{\theta}$ files

There is an option in `superheader.h` (see Appendix C) to display the inferred $\boldsymbol{\theta}$ after each word of compression data. Three such outputs were saved in `~/Dasher_Devel/languagemodel/testing/output/` for easy reference, although it is not hard to create them for any given test. They correspond to:

`thetas_topic`
Topic model compression, `critique_tr_5.bmx` → `practical_tr`, with no α_{new} and the best parameter values known, $Tu = 7$, $\mu = 0.75$. Compression was 8.148 bits per word.

`thetas_mix`
Mixture model compression, `critique_tr` → `practical_tr`, with no α_{new} or sentence markers, and $T = 5$, $Tu = 25$, $\mu = 25$, $\alpha = 20$, $\gamma = 0.7$. Compression was 7.125 bits/word.

C Guide to the Code

All programs used for this project are in `~/Dasher_Devel/languagemodel`, with `~/ tdd23`'s home directory on `flotta.ra.phy.cam.ac.uk`. They are all written in `C++`.[23] We shall try to describe a little of how they work together, and give some advice for anyone wanting to use or change them. Perhaps the best way to actually get to know the code is to start by looking at the main optimization program `mixopt.cpp` and the `makefile`, and to use this as a reference guide.

Note that all the executable programs mentioned here can be built with the `makefile`.

The header file `superheader.h` was a late addition to the code, but many important options were moved to it, and can be toggled by editing it. We shall reference these options in the relevant sections.

The main functional code is contained in several classes, each defined in its own `.cpp` file and declared in a header `.h` file. To each class there also corresponds a testing program, used to test (some) proper functionality. We will describe these classes in more detail in Section C.2, but list them here in Table 2 for easier reference.

In addition, there are three programs that use the main classes to actually compress documents, `ntopt.cpp`, `topicopt.cpp`, and `mixopt.cpp`. These are discussed

[23] The author's programming style and knowledge of `C++` changed somewhat during the course of this project. While there was an attempt to standardize things at the end, he apologizes that some of the earliest code may not be completely streamlined.

class	header file	definition	testing file
Trie	trie.h	triepointers.cpp	trietest.cpp
TopicTable	topic.h	topic.cpp	topictest.cpp
BetaMatrix	betamx.h	betamx.cpp	betatest.cpp
ActiveTrie	activetrie.h	activetrie.cpp	activetest.cpp
NTrie	ntrie.h	ntrie.cpp	ntrietest.cpp
HashList	hashlist.h	hashlist.cpp	hltest.cpp
FileStat	filestat.h	filestat.cpp	fstest.cpp

Table 2: Main Classes and Associated Files

in Section C.3, along with `bmxbuild.cpp`, which generates beta matrices for the topic model.

First however, we will discuss the simpler problem of text formatting, accomplised with `textformat_s.cpp`. `sf_compare.cpp`, for finding the number of different words in one file vs. another, is also explained in this next section.

C.1 Text formatting

Aside from depending on `superheader.h`, `textformat_s.cpp` is entirely self contained. The executable file `textformat`, like all other executables used in this project, is created with the `makefile`, via "`make textformat`".

The original text formatting program was just called `textformat.cpp`. It read all characters from an input file, counting any sequence of one or more consecutive letters as a 'word,' and wrote all 'words' one-per-line in an output file. All letters were automatically converted to lowercase. All other characters (including *e.g.* apostrophes) were ignored, *except* document separators. These were a special sequence of characters manually inserted into input text to indicate document breaks. The sequence still is "`<%d%>`", defined on line 42 of `textformat_s.cpp`. Whenever a document break occurred, a single '`%`' character was written on a line of the output file. The output file was by default called `inputfilename.f`, the `.f` extension signifying "formatted."

There are some disadvantages this crude form of formatting, such as (*e.g.*) not properly dealing with contractions. But some kind of formatting was needed, and this is simple and (it seems) effective.

`textformat_s` extended the functionality of `textformat` in two ways. First, in addition the manually inserted "`<%d%>`" document breaks, every paragraph break (defined as two '`\n`' characters in a row) became a document boundary – so long as the document created had at least `MIN_WPD` words (default 40). This value is defined in `superheader.h`.

Second, after every *sentence* break in the input text, the 2-line sequence "`\n$`" was added to the output file, because it was necessary to indicate sentences to the N-gram model. Both the empty line and the '`$`' character indicate a sentence break. The advantage in using them both is that the '`$`' can optionally be read as an actual word at the beginning of every sentence, and the empty line will still indicate a sentence boundary.

The default output file names for `textformat_s` are `inputfilename.sf`, the `.sf` extension signifying "sentence format" (or such). The old `.f` system is completely obsolete.[24] *Every file that is read by any other program or module here must be in* `.sf` *format!*

After formatting, `textformat_s` writes to standard error the number of words, sentences, and documents written to the output file.

A simple tool to obtain some more information about text files is `sf_compare.cpp`. This program takes two (`.sf` format) input files, and calculates the number of new words appearing in the second (not contained in the first). It also calculates the number of *distinict* new words. This is a crude measure of the similarity of the text files. If run with `testing/empty.sf` as the first input file, it will calculate with total size and total vocabulary size (distinct words) of the second file.

C.2 Main modules

The main modules are listed in Table 2. We shall (briefly) go through each one in turn, describing its main functionality. As usual, the best way to get a feel for how the modules are used is by looking at their testing files. There are quite a few (hopefully) helpful comments in module files themselves, especially the header files.

class `Trie`
Header file `trie.h`, auxiliary header `trie_sup.h`, implemented by `triepointers.cpp`.

This was our first attempt at creating a "hashmap-lookup linked list" data structure, quite frequently used in all the modules. For the languge model algorithms that have been described, it was important to both look up probabilities/frequencies of words and to recourse (and possibly order) entire collections of word probabilities quickly, motivating the use of such a data structure.

The `Trie` is just a monogram trie (though potentially extendible, as we shall see), into which one can insert both words and document breaks. It keeps track of the total frequency of each word inserted, as well as the frequency of each word in each document that has been seen. Words are stored as `struct word_node`'s, defined in `trie_sup.h`, which are more commonly referenced by `Word_T` pointers to them. The structs contain the word as well as frequency information. The Trie itself is a giant hashmap (`hash_map` from the `C++ STL`) whose keys are words (`char*`) and whose entries are `Word_T`'s. The word structs also contain previous and next `Word_T`'s, and are linked in a (null-terminated) double-linked list. The pointers `Trie::top` and `Trie::bottom` point to the top and bottom of the list.

As words are inserted, the linked list is automatically reordered by decreasing total frequency (essentially a bubble sort); so the word with highest frequency is pointed to by `top`. It may be good to write a word insertion function (rather, to rewrite `first_insert` and `later_insert`) that skips the ordering, as it is unncessary for actual language model compressions, and only helps if words are to be displayed.

`Trie` can read directly from a `.sf` file, via the constructor or the `read_file` func-

[24] Nonetheless, we wanted to mention it in case any traces are left.

tion. The maximum length of words is defined by `MAX_WORD_LENGTH` in `superheader.h`, 80 by default. NOTE that difficulties were encountered for values of this number smaller than the lengths of words in input text. This is a bug that has not yet been fixed; rather we just hope that no word occuring in our texts is longer than 80 characters. (It would nevertheless be good to fix this bug.)

One nice thing about `Trie` is that it can be extended (derived from) to include word nodes with more information. Both `TopicTable` and `ActiveTrie` are examples of this. The `struct word_node` has a pointer `Oth_T` to a `struct Oth` which may be redefined. The public `insert` and `remove` functions both call respective private versions which take functions of `Word_T`'s as arguments; these functions can be specifically written in any derived class, and the `insert` and `remove` functions overloaded to call them. (Taking a look at how this is done in `TopicTable` should clarify things.)

class `TopicTable`
Derived from class `Trie`, header file `topic.h`, implemented by `topic.cpp`.

This module was built to calculate topics from a given data set. Besides the information stored in `Trie`, it also has `theta*`, a 2-dimensional array of $\boldsymbol{\theta}^d$'s, and `N*`, an array of total word counts in each document. It uses the `oth` field in word nodes to store the parameters $\{\beta_{aj}\}$ and $\{q_j^{a,d}\}$ as 1- and 2- dimensional arrays, respectively, since these depend on words a.

After an input file is read and a desired number of topics specified, the function `refresh()` should be called before calculating topics. `calc_topics` can then calculate topics; it must take as a parameter a maximum number of iterations for which to run the learning algorithm. `re_calc_topics` can then be called, if desired, to add more iterations. There is also the option of defining minimum distances (threshholds) to be reached between $\boldsymbol{\beta}$'s or $\boldsymbol{\theta}$'s of consecutive iterations, as discussed at the end of Section 2.5.1; calculations will stop if these threshholds are reached. Default threshholds can be turned on; their values are defined by `TOPIC_B_ERROR` and `TOPIC_TH_ERROR` in `superheader.h`.

The `calc_topics` and `re_calc_topics` functions both take a boolean argument `show_it`, which, if true, prints a message every few iterations. The number of iterations skipped between messages is defined by `THETA_SHOW_NUM` in `superheader.h`.

The topic matrix which is calculated can (and should) be stored in a `BetaMatrix`, which can display the top words in each topic and/or their probabilities. The testing file `topictest.cpp` contains an example of this.

Note that when reading from a file into any of these modules there is an option of counting '`$`' sentence markers to be words. (This is particularly relevant for `TopicTable`, `ActiveTrie`, `NTrie`, and `FileStat`.) The option is toggled on and off via `BOSM` in `superheader.h`; code must be recompiled if it is changed.
class `BetaMatrix`
Header file `betamx.h`, implemented by `betamx.cpp`.

This class was written exclusively to store topic matrix information. Again, its basic data structure is a linked list with hashmap entry. But this time its nodes, pointed to by `Wordb_T`'s, each contain a length-T *array* of "next" pointers; we are

using a unidirectional D-times linked list, in order to store a different ordering of words for each topic. Each node also has a length-T array of probabilities. (This implementation was chosen to minimize the space required by the BetaMatrix.)

Two of the most useful `BetaMatrix` functions are `save` and `load`, allowing storage of calculated topic matices in `.bmx` files. Note that the loading a `BetaMatrix` from a file automatically renormalizes each topic.

Besides outputting a `BetaMatrix` from `TopicTable`, it is possible to manually insert words, or to update their probabilities, but only for the latest topic. The ability to alter previous topics, or to remove words, has not been implemented yet. Note that there is a bug in the ordering of words in the last topic if their probabilities are updated (*i.e.* if they are not newly added). Running `activetest` shows its symptoms. It may not be too hard to fix, but it was not necessary to do so for compression tests.

class `ActiveTrie`
Derived from `Trie`, header file `activetrie.h`, implemented by `activetrie.cpp`.

This is the main class for topic model compression calculations. It cannot read and compress an entire file – that functionality is in `FileStat` – but it stores words and documents of compression text and performs the iterations needed to learn $\boldsymbol{\theta}$ for the current document. It must be given a pointer to a `BetaMatrix` to perform these calculations. `ActiveTrie` is the class which deals with topic model hyperparameters (but not α_{new}). Its redefined `new_document` function is what adds the new $T+1$st topic to the `BetaMatrix`. (Hence `ActiveTrie` will actually modify the `BetaMatrix` it is given.)

Note that there is a `new_document_rebuild` version of this new document function which, instead of adding compression text words into new documents as `Trie` would do, completely erases `ActiveTrie`'s trie with every new document and rebuilds it. This was implemented to make it more efficient. This is also the reason some removal functionality was introduced in `Trie`.

Much like in `TopicTable`, the maximum number of $\boldsymbol{\theta}$-calculating interations must be specified, although a threshhold can also be set. One can set the default threshhold defined by `AT_TH_ERROR` in `superheader.h`. If, in `superheader.h` `THETA_SHOW` is turned on, a message will be output every `THETA_SHOW_NUM` iterations.

There is also a `THETA_BUILD` option in `superheader.h`. If turned on, during the course of a compression text document the initial calculation values for $\boldsymbol{\theta}$ will not be reset randomly after every word. Rather, they will be kept from the previous iteration. This feature was implemented in hope of increasing compression speed by not needing to do very many iterations before reaching a theta threshhold for later words in the document. It has not been throroughly tested, and unfortunately does not seem to work very well.

Note that default hyperparameters are defined by `DEF_U` and `DEF_U_ND` in `superheader.h`; the u_i's for all previously known topics are set $\equiv$ `DEF_U`, and u_{T+1} is set to `DEF_U_ND`.

class `NTrie`
Header file `ntrie.h`, implemented by `ntrie.cpp`.

This, finally, is a real trie, of arbitrary depth N which is set when `NTrie` objects

are created. It is largely independent of the other modules, and can both build its trie from training data and use PPM to compute probabilities or calculate the compression of a file. Its code was written from scratch in order to be more streamlined – it might have been possible but awkward to derive from `Trie`.

The code is decently commented and mostly self-explanatory. There are options to use update exclusion before building the trie, and to compress with or without compression-time learning. PPM parameters α and β, as well as α_{new} can be set at the beginning of compression calculations; otherwise, default values are used. There is a rather complicated system of keeping track of the number of distinct words the model knows about, which has been tested and seems to work quite well.

There is a `show_it` option during compression, which if set to `true` will cause a message to be displayed every `COMP_SHOW_NUM` words of compression text – defined in `superheader.h`. The maximum number of characters per word `NT_MAX_WORD_LENGTH` is also defined in `superheader.h`. If sentence markers are to be included as words, `BOSM` must be turned on in `superheader.h`.

`NTrie`, performing full compression, also has several options for dealing with new words. If α_{new} ("`alpha_new`") is negative, they will be ignored. If it is set to a positive value, there will be an escape probability for new words as described in Section 2.3. Whether uniform or 2-bits-per-character new word probabilities are used depends on whether `STRICT_NEW_WORD` is turned on in `superheader.h`.

Note that the implementation of `NTrie` uses the usual linked lists with hashmap access, but now the word nodes (pointed to by `sWord_T`'s) each contain pointers to such data structures – their subtrees. In addition to the counts discussed in Section 3.1, the nodes also keep track of the sums of frequencies in their direct subnodes. Particularly when update exclusion is used, these sums are *not* always equal to the frequency counts in the nodes themselves. Figure 4 illustrates what we mean, with subnode sums in parentheses for the first few nodes of the trie from Figure 3; note how the sum in [`it-was`] is not the same as its count.

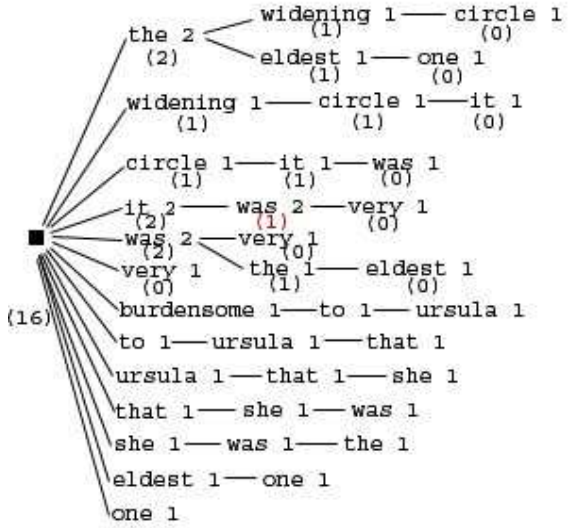

Figure 4: Subnode Sums in NTrie

The sub-linked-list of every node in `NTrie` is singly-linked and automatically sorted upon each insertion/update in order of decreasing frequency. Note finally that throughout `ntrie.cpp` the term "context" usually means a previous context (as used in Section 3) *plus* the current word.

class `HashList`
Header file `hashlist.h`, implemented by `hashlist.cpp`.

This class was intended to be a simple (or better written) version of `Trie` with increased functionality, and storing `char*` words and `double` probabilities (rather than `int` frequencies) in word nodes. It was written mainly for the mixture model, to hold the probability distributions over words that must be calculated and multiplied during compression. `NTrie` has a function that can output probability distributions to a `HashList` (this is then not necessary for the topic model).

The insertion (and removal) functions were optimized for efficiency. There are several versions of insert: `insert` is like the old `Trie` insert, sorting after every insert or update; `insert_unordered` simply places any inserted word at the bottom of the linked list, without checking to see whether the `HashList` already contains it; and `insert_unordered_check` does the same but checks first and updates a word's probability if it is already there. There is a sorting function which can be called any time; the linked list is double (vs. single) linked, to enable a quicksort. It is possible to remove one word at a time, but also to simply clear the entire `HashList`.

All these various options were implemented to allow programs that use `HashList` to choose the most efficient way of inserting or retrieving information from it.[25] All the members of `HashList` are `public` for easy manipulation.

The maximum length of words in `HashList`, `HL_MAX_WORD_LENGTH`, is defined in `superheader.h`.

Note that the later classes `NTrie`, `HashList`, and `FileStat` all use an integer `exit_flag` to track the behavior of some functions; this is quite useful in debugging.

class `FileStat`
Header file `filestat.h`, implemented in `filestat.cpp`.

This is the last main module, and possibly the most important. Through it, one can calculate file compressions via the topic model, the mixture model, and the N-gram model; it implements the first two, and standardizes the use of the last.

One should take a look at `fstest.cpp` and `mixopt.cpp` as well as the `filestat.h` header file itself for some examples of how the module is used. For the mixture model, `FileStat` is given pointers to an `NTrie` and an `ActiveTrie`, both of which should be trained before any compression calculations are done – the `NTrie`'s trie should be built, and `ActiveTrie` should contain a pointer to a `BetaMatrix`. The α and β PPM parameters should be set within `NTrie` and `ActiveTrie`, and hyperparameters should

[25] For example, the mixture model in `FileStat` initially used `insert_unordered` to store the N-gram probability distribution for every word of compression data read, and then erased the entire `HashList` before the next word. This was later changed to using `output_unordered_check` and not erasing, possibly (but not certainly) faster. (`NTrie` has functions that output to `HashList` both ways; these are what `FileStat` called.) If one wants to display the `HashList`, it can be ordered right then.

be set within `ActiveTrie`. On the other hand, the maximum number of $\boldsymbol{\theta}$-learning iterations to do for each word in the topic model, a possible $\Delta\boldsymbol{\theta}$ threshhold, and a possible value for α_{new} should be set within `FileStat`; *any values of these parameters present in the other modules will be overridden.* Note that `FileStat` may modify the `NTrie` and `ActiveTrie` it is given. The mixing parameter *gamma* is also set in `FileStat` before compression.

The strength of `FileStat` is that it can also just use one component model or the other for compression. By default it uses all models it is given, but the `use_AT` and `use_NT` functions (topic and N-gram models) can be called to change this. Even when using just one component model, however, the α_{new} and possibly maximum $\boldsymbol{\theta}$-iterations or $\Delta\boldsymbol{\theta}$ threshhold from `FileStat` still take precedence over these values contained in `NTrie` or `ActiveTrie`.

Like `NTrie`, `FileStat` has a somewhat complicated method of keeping track of the vocabulary size (number of distinct words) it knows about. For the mixture model, this is the minimum of the vocabulary size of the two component models. This information is necessary if new word escape counts (α_{new}) are used.

The number of compression-text words that are calculated to have zero probabilities are counted in both `NTrie` and `FileStat`, and output to standard error in `FileStat` at the end of compressions (if undesired, this can easily be changed in the `FileStat::calc_...` functions). If no new word probabilities are used, this is the number of new words skipped, and is necessary in calculating the final compression per word. If $\alpha_{new} > 0$ is defined, this number should be zero – otherwise something is going wrong.

As for other modules, `THETA_BUILD`, `THETA_DISP`, `STRICT_NEW_WORD` and `BOSM`, as well as default hyperparameter values, maximum word length, etc, can all be turned on and off from `superheader.h`.

C.3 Compression programs

There is not much left to say about the actual compression programs themselves. They simply make use of the various modules we described above. Looking at them may be a good way to become familiar with how the modules work. The four programs described here are all built with the `makefile`. Various parameters can be set in, *e.g.* `superheader.h` before (re)compilation. For each of these programs, there is a shell script in `~/Dasher_Devel/languagemodel/scripts` illustrating proper command line usage. (These were placed in a separate directory so they are not executed by accident.)

`bmxbuild.cpp`

This program was used to create beta matrices from training text, using the algorithm in (15)-(17). This is a memory-intensive operation, and the program, as it is written now, stores all the beta matrices it calculates in each round. (This may not be a good idea.) There may also be some memory leaks in the topic model. The upside of all this is that, for a large number of topics, memory overflows may occur after a while. The program detects these by noting the time it takes to calculate

iterations, and breaking if this increases too much. Again, there may be a better way to do things, but this was at least functional.

When using the program, it is a good idea to adjust the number of rounds (`NO_ROUNDS`) or in particular the number of iterations done per round (`SETNO`) in accordance with the number of topics desired. Less for more topics. `bmxbuild` outputs various statistics after each round of `SETNO` iterations, including inferred values of hyperparemters – although, as we have noted elsewhere, the hyperparameter statistics calculated during topic-learning are not good values of compression-time hyperparameters.

Examples of usage are found in `bmx_script`.

`ntopt.cpp`

This program (for calculating N-gram model compression) is slightly different from the rest in that it doesn't take command line parameters. The files it operates on are written directly into the code. It uses a loop (or can use a double loop) to vary values of interesting parameters. Since the N-gram model is quite quick, this is a nice way to test a large range of (for example) PPM α and β parameters.

For testing less uniform ranges of parameters on different input files, etc, `mixopt` can also be used.

(There is no sample script for `ntopt`, but see the comments in the code.)

`topicopt.cpp`

This is the topic model compression program. An attempt was made to write it in a loop(s) like `ntopt`, but it seems there is a memory leak somewhere in the topic model, and this didn't work. (The source of this leak is still unknown.) Thus we resorted to command line input and shell scripts. The script for `topicopt` is `topic_script`.

The program can be adjusted to read values for various parameters from command line input; these are currently referred to as `opt1` and `opt2`.

`mixopt.cpp`

`mixopt` is the most powerful and flexible compression program. It works essentially like `topicopt`[26], but uses FileStat to do the compression. Thus it is possible to change a few lines of `topicopt.cpp` (as indicated in the code) to do compression with only the topic model, only the N-gram model, or with the mixture model.

Currently, the program takes five different command line parameters, besides the N-gram training file, the topic beta matrix to be used, and the file to be compressed. Sample uses are in `mix_script`.

Compression information from the three programs is usually output to standard output, while any other descriptive statistics go to standard error – enabling redirec-

[26]Note the "`opt`" stands for optimization – optimizing compression by adjusting paramters.

tion of compression data to, *e.g.* a single file. The compressions can be formatted as one wishes. They are currently written to enable easy conversion to *Mathematica* lists.

References

[B] T.C. Bell, J.G. Cleary, and I.H. Witten, *Text Compression*, Englewood Cliffs: Prentice Hall.

[BlNJ] D. Blei, A. Ng, and M. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, 3:993-1022, January 2003.

[MBP] D.J.C. Mackay and L.C. Bauman Peto, "A Hierarchical Dirichlet Language Model," *Natural Language Engineering*, 1(3):289-307, 1995.

[ML] T. Minka and J. Lafferty, "Expectation-Propagation for the Generative Aspect Model," *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence*, 352-359, 2002.