

BUILDING LANGUAGE MODEL ON  
CONTINUOUS SPACE USING GAUSSIAN  
MIXTURE MODELS

Wei Chen

Supervisor: Professor Sanjeev Khudanpur

A Report for Research in Language Modeling

from January 2007 to July 2007

# Abstract

Our work focuses on exploiting word distribution on a continuous space. We aim at finding out how continuous space performs in language modeling, when compared with its discrete counterpart. Also, we propose a new method for predicting infrequent words by borrowing information from some frequent words, which are close to them in our word vector space. We use bigram counts to initialize our word vectors. Some matrix factorization processes, such as SVD and NMF, have been used to reduce the size of the vectors. Then we model the word vector distribution by a Gaussian Mixture Model (GMM), which has been trained incrementally using EM algorithm. The performance of various methods in our model training, such as MAP adaptation and Tied-Mixture System, have also been studied and discussed in detail.

# Introduction

Recently, many efforts have been contributed to combining latent semantics analysis with traditional n-gram model, in hoping to deal with the locality problem of n-gram model. Among various methods used to find hidden variables, which give out class information, SVD offers a good way to extract latent semantic information of words by finding out “word basis” (Bellegarda, 2000). Later, NMF is considered an alternative to SVD. And since NMF gives out non-negative numbers, the output of NMF has been used for assigning probabilities to words (Novak and Mammone 2001). A PLSA-based language model is also used to achieve the goal of better smoothing on word probabilities (Mrva and Woodland, 2006).

The previous works are all done in discrete space. We are more interested in learning word distribution on continuous space. Our question arises from how to assign probability to infrequent words. We notice that while exploiting latent structure of the text, infrequent words show similarity with some frequent words. Thus, it is reasonable to assign probability to infrequent words with knowledge of their structure learned from their similar frequent counterparts. For example, if “lychee” is the infrequent word, and by doing some latent semantics analysis, we find that it is close to a frequent and thus well-learned word “banana”, we could use parameters we learned from “banana” to help assign probability to “lychee”.

The very first question is how to represent each word with a vector in continuous space. One approach is to let machines search for a good representation automatically (Schwenk and Gauvain, 2005). Here, we propose using bigram to initialize our word vectors. The reasons to use this model are two-fold. First, we want to see how continuous distribution model behaves when compared to traditional bigram model with the same dependency information. Second, by using some clustering technique,

we find that after doing SVD or NMF, the words already show good semantic and syntactic information, even with this limited context knowledge. The goal for using SVD and NMF is to reduce the dimension of our word vector space. Which one fits our model better is a question we need to answer. It turns out these two methods give different structure for the distribution of the word vectors. And results show that NMF beats SVD in fitting our goal.

Another important part of our work is to choose a probability density function to model the word distribution, and then train the parameters appropriately. We assume this distribution could be modeled using a Gaussian Mixture Model (GMM). We want an appropriate model which is neither too simple to approximate the real distribution, nor too complicated to be trained and tested efficiently. Thus, the development of a nice training strategy is highly desired. In our experiments, we compare several possible methods and their related parameters, such as MAP and Tied-Mixture System for training bigram model.

# Singular Value Decomposition

Suppose we have the vector representations of our words, but the size of the vectors may be too large for computing. Thus, working on some lower dimensional vector space, which is a good approximation of the original space, is needed. Singular Value Decomposition (SVD) is hence used in our application for projecting word vectors to lower dimensional space. In this section, we first introduce a co-occurrence matrix, which gives the original word vectors, and then we explain how SVD could lower the rank of the co-occurrence matrix. We also give some discussion for the performance of SVD at the end of this section.

## Data preparation

We use text from Wall Street Journal in Workshop 1998 as our training and test data. The text is not normalized, and each sentence has an id. For example,

<s.89.01.891102-0193.1.1>

<s> Pierre Vinken sixty one years old will join the board as a  
nonexecutive director November twenty ninth </s>

means this sentence comes from document 891102-0193, paragraph 1 and is sentence one.

We split the entire data set into training and test data. Table 1 shows the size of each one.

	Number of Tokens	Number of Sentences	Number of documents
Training	4683669	190389	9339
Test	154025	6211	333

Table 1

Before we use the data, we need to normalize them. Some necessary steps include lowercasing the text, replacing numbers with a unique token called “<num>”, and getting rid of all the punctuations.

After we have our normalized training data, we define our vocabulary by choosing the most frequent 10000 words. And we replace all the out-of-vocabulary words (OOV) with “<unk>”.

## Co-occurrence Matrix

The co-occurrence matrix  $W$  between words and their histories is constructed by keeping track of which word is found after what history in the training data. So  $w_{ij}$  is the number of times the  $i^{\text{th}}$  word in the vocabulary appears after the  $j^{\text{th}}$  history word. The matrix is very sparse, with 762323 nonzero entries, which means there are only around 0.76% elements in the matrix are nonzeros. In this research work, we represent this kind of sparse matrix in Harwell-Boeing Format.

The co-occurrence matrix  $W$  defines a vector representation for each word and each history: one row of  $W$  corresponds to a word in the vocabulary, and one column of  $W$  corresponds to a history. Thus, the size of both kinds of vectors is 10001, including one for the OOV's.

## SVD Basics

The goal of SVD is to find an approximation  $\hat{W}$  such that  $W \approx \hat{W} = U\Sigma V^T$ , where  $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_s)$ , which contains  $s$  singular values sorted in descending order.  $U$  and  $V$  are orthonormal matrices, which are called left and right singular matrices, respectively.

We have two important properties following the definition of  $\hat{W}$  :

1.  $rank(\hat{W}) = s$
2.  $\min_{r(A)=s} \|W - A\|_F^2 = \|W - \hat{W}\|_F^2 = \sigma_{s+1}^2 + \dots + \sigma_m^2$ , where  $m = rank(W)$

SVD with property 1 gives us a way to find a projection from high dimensional space to lower dimensional space. Property 2 ensures that  $\hat{W}$  is a good approximation of the original matrix.

In the decomposition, rows of matrix  $V$  could be viewed as a set of basis vectors of the vocabulary, and each of the words in our vocabulary could be obtained by some linear combination of them. Since  $U\Sigma$  serves as the coefficients of the linear combinations, we could use the rows of  $U\Sigma$  as the word vectors in the  $s$ -dimensional space.

## SVD Computations

The SVD computations are based on equivalent eigenvalue problems. There are two methods to turn SVD into solving for eigenvalues of a symmetric matrix:

1. The eigenvalues of the symmetric matrix  $B = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$  are  $m$  pairs,  $\pm\sigma_i$ ,

where  $\sigma_i$  is a singular value of  $A$ .

Notice that  $A = U\Sigma V^T$ , so we could write the matrix  $B$  as

$$B = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} U\Sigma V^T - U\Sigma V^T & U\Sigma V^T + U\Sigma V^T \\ V\Sigma U^T + V\Sigma U^T & V\Sigma U^T - V\Sigma U^T \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} U & U \\ V & -V \end{pmatrix} \begin{pmatrix} \Sigma & 0 \\ 0 & -\Sigma \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} U^T & V^T \\ U^T & -V^T \end{pmatrix}$$

2. The eigenvalues of the symmetric matrix  $A^T A$  are  $(\sigma_1^2, \sigma_2^2, \dots, \sigma_m^2)$ , where  $\sigma_i$  is a singular value of  $A$ .

Notice that  $U$  and  $V$  are orthonormal matrices, so the above statement could be

shown by  $A^T A = V\Sigma^T U^T U\Sigma V^T = V\Sigma^2 V^T$ .

There are four algorithms for doing SVD (Berry, 1992): Subspace Iteration, Trace Minimization, Single-Vector Lanczos Method and Block Lanczos Method. All these methods are to find an invariant subspace which would be mapped into itself by transform B, and solve for the basis of the subspace (since any invariant subspace has a basis of eigenvectors). Subspace iteration is a block generalization of the classical power method for computing eigenpairs. It forms the sequence  $Z_k = BZ_{k-1}$ , with the hope that Z will converge to the invariant subspace. Trace minimization is a conjugate gradient method for solving an optimization problem to minimize the trace of a resulting diagonal matrix from which we could obtain the singular values. The Lanczos procedure works for generating orthonormal basis for Krylov subspaces and computing the orthogonal projection of B onto these subspaces. The reason why Krylov subspaces are desirable for eigenvalue/eigenvector approximations could be found in “Lanczos Algorithm for Large Symmetric Eigenvalue Computations” by Cullum, Birkhäuser, 1985. The basic Lanczos recursion can only compute the distinct singular values of a matrix B and not their multiplicities. As a block generalization of the single vector Lanczos algorithm, the Block Lanczos Method adds an inner loop inside the Lanczos iteration, and computes all the multiplicities of the singular values.

We use the C version of SVDPACK to do the computation. We will give a comparison for the performance of all four algorithms stated above.

For bigram experiments, our input for SVD procedure is a square matrix (SVDPACKC could deal with tall matrices) stored in Harwell-Boeing Format. Since the size of our matrix is 10001×10001, which is much larger than the default size limit defined in SVDPACKC code, we need to modify the value of some constants in the source code. There are three constants that must be reset:

LMNTNW (work space for SVD): 400200001

NMAX (maximum number of columns or columns+rows): 20000  
 NZMAX (maximum number of nonzeros): 4000000

The recommended way to set the value of LMNTNW is as below:

$$\text{LMNTNW} = (6 * \text{NMAX}) + (4 * \text{NMAX}) + 1 + \text{NMAX} * \text{NMAX}$$

The c code is compiled under ansi-c compiler. Once it is successfully compiled, we could run the experiments on our co-occurrence matrix. We compute a 500-factor singular value decomposition. For the bigram matrix, the SVD procedures take about 100-200 mega bytes of memory to run. We give out a running time comparison, including the parameter setting for the four algorithms mentioned above. For this experiment, we use a variation of our original co-occurrence matrix:  $\tilde{w}_{ij} = \log(w_{ij}) + 1$  for all the nonzero values, and we leave the zero entries unchanged.

For each algorithm, the first experiment is done by solving the eigenvalues of  $B = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$ , and the second one is done by solving the eigenvalues of  $A^T A$ . The size of the subspace is always set to be 550, and the residual tolerance for approximated singular triplets is set to be 1.0e-3 (this is determined by the magnitude of the matrix, setting smaller values may cause problem). Computation of singular vectors is also required. For Trace Minimization method, Ritz-shifting is used for acceleration. We run the procedures on Linux i686, with 4G memories.

	Method 1	Method 2
Subspace Iteration	00:42:44	02:34:07
Trace Minimization	09:51:52	07:19:51
Single-Vector Lanczos	00:03:02	00:02:19
Block Lanczos	00:37:20	01:06:48

**Table 2**

By far, it shows that the second method on Single-Vector Lanczos Algorithm is the

fastest. However, the output file gives less than 500 singular triplets. So the following experiments all rely on Block Lanczos Method.

## SVD Analysis

Figure 1 and figure 2 shows the magnitude of the singular values. Figure 1 plots the values of singular values from 100 to 500. And figure 2 plots the first 100 values. We use another variation of the original raw-count matrix here. We normalize the rows by their entropy, and then we scale the columns:

$$\varepsilon_i = -\frac{1}{\log(m)} \sum_{j=1}^m \frac{w_{ij}}{\sum_j w_{ij}} \log \frac{w_{ij}}{\sum_j w_{ij}}$$

$$\tilde{w}_{ij} = (1 - \varepsilon_i) \frac{w_{ij}}{\sum_i w_{ij}}$$

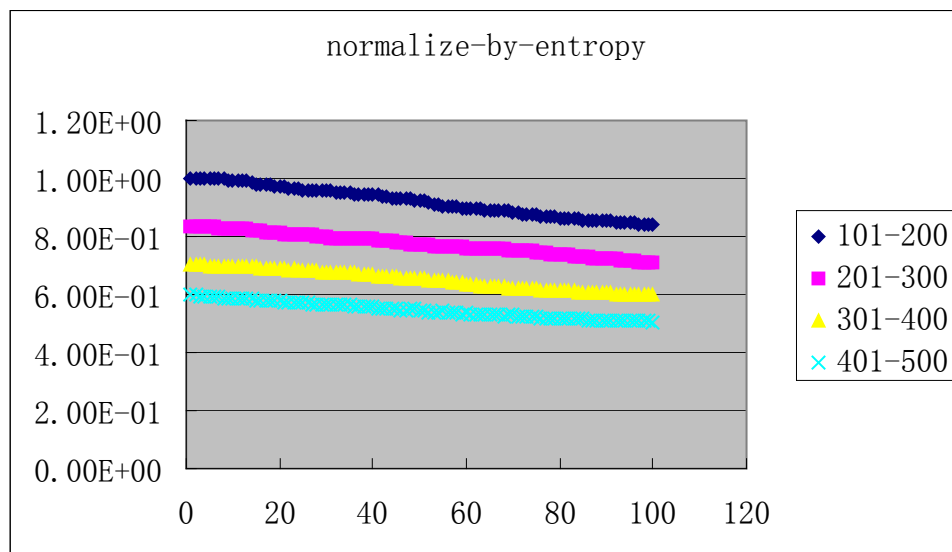


Figure 1

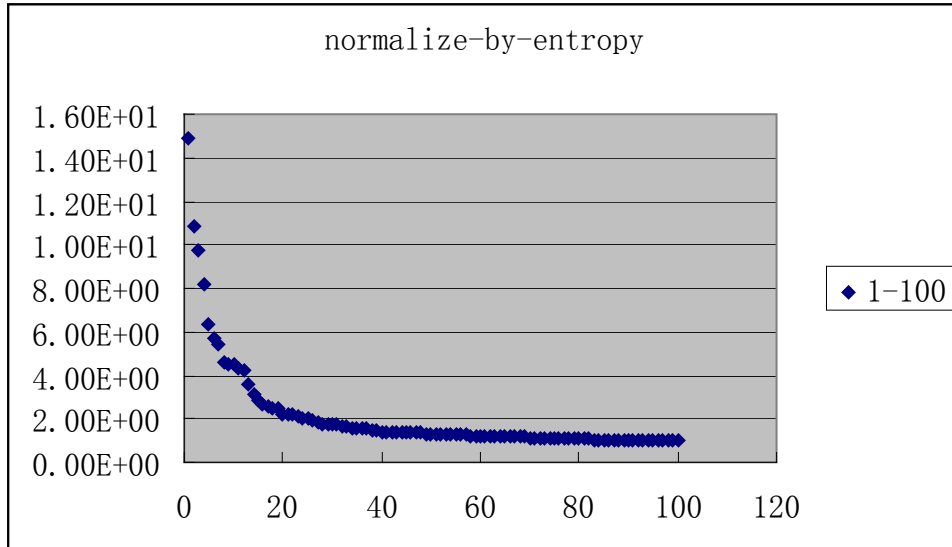


Figure 2

Another set of plots is also provided for the raw-count matrix, in figure 3 and figure 4. Here, the convergence rate is different from that of the normalized-by-entropy case. Also note that the error of the approximation is the sum of the square of the neglected singular values, so the error of the raw-count matrix is much larger than the normalized-by-entropy matrix.

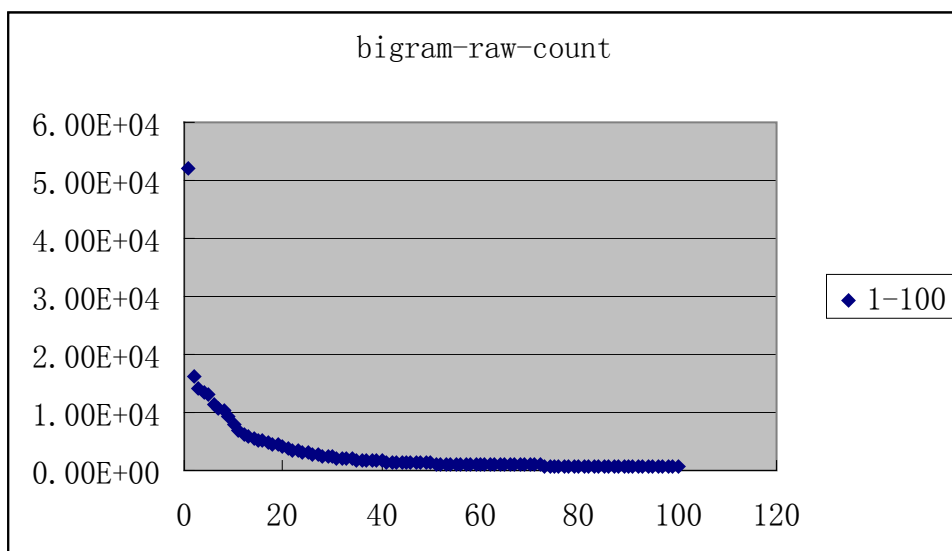


Figure 3

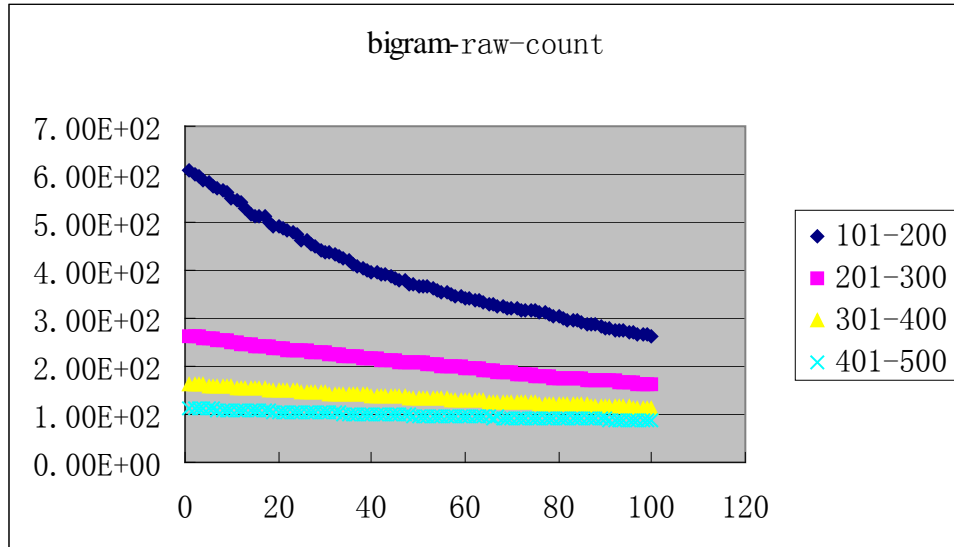


Figure 4

To help observe the error of the reconstructed matrix, we plot the error matrix:  $W - \hat{W}$  below in figure 5. The matrix has been sampled to be of size 1000\*1000, and the absolute values have been scaled by their log. Here, the dark lines indicate small errors, and the light lines indicate large errors.

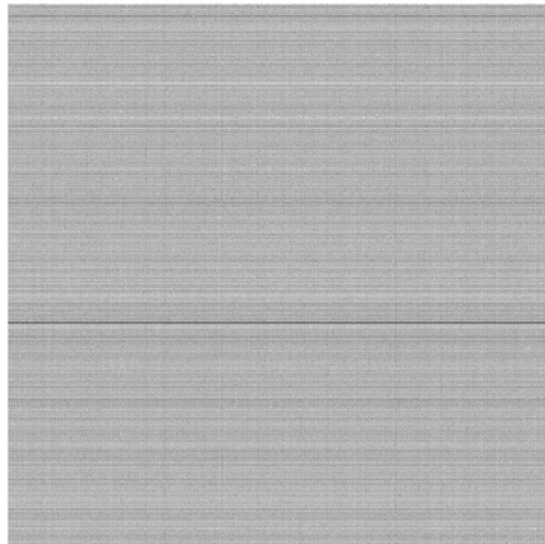


Figure 5

In order to scale the values by their log, we used the absolute values of the original numbers, thus it is not clear from figure 5 which words have been under-estimated and which one have been over-estimated. These will be discussed in detail in the

following sections.

## K-Means Clustering

Doing K-means clustering gives us information about how the word vectors are distributed in the vector space. We do the clustering in the sense of Euclidean distance. It shows that our clusters reveal some knowledge on bigram structure, semantic and syntactic information. Below are some word cluster examples from 150 clusters:

<b>Bigram Structure (all appear after the word “new”)</b>
York, Jersey, Orleans, Yorker, Zealand, Hampshire
<b>Semantic Information (business)</b>
business, industry, manufacturer, sales, companies, giant manufacturers, manufacture, maker, makers, retailing, operations, shipments, receivables, wholesalers, interdiction, businesses, garments, cans, divisions, division, owners, exports, bottles, making, wings, belonging, plants, sector, subsidiary, factory, farmers, bookings, trafficking, production, franchiser, locations, recorders, fronts, roof, repair, closings, importers, trademark, exporters, tanks, restaurants, toshiba, operator, tracks, fur, giants, tabloid, rooms, studio, ink, categories, exporter, routes, billings, unit, reduces, car, dealership, cleaning, firms, trips, commercials, hotel, dealers, producers, driver, chains, hut, restaurant, rental, plant, consulting, insurer, spots, chain, brand, processors, curriculum, merchandising, district, paint, imports, supplier, bailout, bankruptcy, franchise, model, bullion, tapes, warehouses, competitors, hog, operated, processing, advertising, firm, pits, subsidiaries, crop, growers, cola
<b>Syntactic Information (past participle)</b>
taken done sunk encountered gone got begun lagged yet tended risen seen noticed always gotten forgotten kibbutz been ever corrected changed hammered intentionally never shrunk squeezed taped fallen counted cleaned consume determined rewarded confronted worked eroded lowest ours requested allowed pursued reorganized stabilized practicing opposed traveled skeptics shown adequately checked designing grown kept realized avoided notified examined heard surrendered discontinued unidentified really chosen made buoyant paid achieved planted promote authorized promoted readily structured looked regulate represent decided slowed hurt spoken embraced contracted stopped shaken accepted provided reached dealt powerful matching affected expected attempted satisfied invested dragged suspected spawned thrown touted planned borrowed missed stepped watched answered proved generated commented submitted endangered circulating discussed tightened deemed given projected entered delivered held auctioned executed cooled stalled placed served revised tried built acted disappeared

Figure 6

Grouping on history words also shows some good semantic and syntactic information. We are interested in this because when we train our bigram HMM, we need to tie some states together in order to get a robust system given limited data.

<b>History Cluster 1 (cities near New York)</b>
buffalo brooklyn charlotte seabrook brunswick armonk newark rochester princeton bronx albany syracuse rock township unilever gardens madison lakes holly county cliffs hampton valley orlando tampa lauderdale chiat Wilmington

<b>History Cluster 2 (function words)</b>
into with toward through after beside for despite by onto upon at off within behind under among <s> on during beyond of inside since until

**Figure 7**

We could get reasonable clusters even when we group words and the histories together:

<b>Word-History Cluster (Health and Medical)</b>
<b>History:</b> child health nursing catastrophic comprehensive skin outpatient intensive acute cooperative urgent medical mental turf hair psychiatric patient preferential bonwit polly
<b>Word:</b> care illness counseling domestically psychiatric rehabilitation audits embarrassment surgical pulling fueling albeit facility illnesses screening surtax homes hungry shortages health advocacy reviews hazard unisys abuse hospitals proven necessity coverage shaking bright clean undertaking labor malpractice conscious posting shortage prose home review setting structural patient covered contest facilities relies problems task trigger kidney aids disappointments therapy moments trials powers centers laser discretion treatment child condition infected wars given archer tab color protection negotiations approach hospital advice accident fat battle neck custody civil procedure matters polluted attitude oriented clinics wheeler liquidation wearing sciences confusion hearings depression mass habits higgins extension maintenance psychologist

**Figure 8**

Another question is how good a word performs in clustering. Since the starting point of K-Means clustering is randomly choosing K centroids, a word may be grouped in different clusters when doing several times of clustering. Thus, we are interested those “stable” words which always stays in one cluster. To do this evaluation, we construct

a 10000\*10000 table to store the number of times two words are grouped in the same cluster. We accumulate the counts when doing 10 times of clustering by choosing different initial centroids. The table is big, symmetric, and it is also very sparse, with only 3% to 5% nonzero entries. Among the nonzero entries, around 22.72% are greater than 5. In this experiment, we group words into 100 clusters.

## SVD for Smoothing

We would like to learn how SVD performs directly as a smoothing method for language model. The goal is to have a sense on how SVD preserves the properties of the original distribution after lowered the rank of the vector space. To do this experiment, we first normalize our original raw-count matrix by the entropy of the rows, and then we scale the columns as we did in analyzing the magnitude of singular values. While in this procedure, we store the multiplicands in a file. Then we feed this normalized-by-entropy matrix in our SVD procedure. After SVD is done, we restore the original matrix by multiplying the entries with the reciprocals of the numbers that have been saved when doing the normalization procedure before SVD. And after that, we normalize each column to make them probability vectors. Until now, we could use the numbers in the new matrix  $\tilde{W}$  as the smoothed probability values for the bigrams:

$$p(w_i | h_j) = \tilde{w}_{ij}.$$

Notice that we are unable to use the entries of  $\tilde{W}$  directly as the probability values, nor can we normalize the columns directly after we reconstructed the original matrix by the reciprocals. This is because SVD may give out negative numbers, which cannot be used as probability numbers. Here, we assume that SVD suggest small values for these negative numbers, so we simply set them to be zero, and then we could normalize the columns and get the probabilities. We will also introduce another approach for dealing with this problem soon.

The main problem of the negative numbers is that those cannot be used for probability values. We notice that a reconstruction is done after SVD in the above model. If we are guaranteed to obtain non-negative numbers through the reconstruction procedure, it would be convenient to get probability numbers in the following step. Also, we are interested in getting comparable approximation errors through SVD. Thus, we could replace the raw counts in the original co-occurrence matrix with their log value. We set the value of  $\log 0$  to be  $-99$ . And after SVD is done, we reconstruct the matrix with  $\exp(\tilde{w}_{ij})$ . In order to keep the errors small, we use the scaled frequency rather than the raw count. So each entry of  $W$  is calculated as:

$$w'_{ij} = -\log \frac{w_{ij}}{\sum_i w_{ij}}, \quad \text{if } w_{ij} \neq 0$$

$$w'_{ij} = 99, \quad \text{otherwise.}$$

When we do the reconstruction  $p(w_i | h_j) = \exp(\tilde{w}_{ij})$ , we no longer have the negative- number problem.

The reason that we normalize the entries of the raw-count matrix by the entropy of the rows is that we want to make the matrix reconstruction error comparable. In other words, we want to have larger error on large numbers, and smaller error on small numbers. But whether this works better for approximating the distribution is unclear. So we set another experiment on the raw-count matrix, do SVD directly on it, and then make the columns probability vectors.

Before we compare the above models with the traditional bigram model, we are interested in another model and wish to compare these together. The other experiment differs with our new model in the matrix setup before SVD. This time, we do not start with the raw count. Instead, we replace the numbers in the raw-count matrix with the probability numbers from Kneser-Ney Smoothing. Once we have the matrix, we do SVD on that and normalize the columns of the output. Now, we could compare these

models together with the traditional bigram models.

**Baseline:**

	Training Perplexity	Test Perplexity	Zero Probs in Test
Good-Turing	109.366	161.617	0
Kneser-Ney	109.567	164.315	0

**Raw Count:**

	Training Perplexity	Test Perplexity	Zero Probs in Test
300-factor SVD	210.748	218.945	5405/147804
400-factor SVD	194.453	205.705	5744/147804
500-factor SVD	182.900	194.497	6317/147804

**Normalize-by-Entropy:**

	Training Perplexity	Test Perplexity	Zero Probs in Test
300-factor SVD	234.745	241.576	4499/147804
400-factor SVD	209.634	217.274	5433/147804
500-factor SVD	194.282	203.463	5921/147804

**Traditional model:**

	Training Perplexity	Test Perplexity	Zero Probs in Test
300-factor SVD	7072.520	7101.240	0
400-factor SVD	7068.473	7099.184	0
500-factor SVD	7067.247	7100.823	0

**Log**

	Training Perplexity	Test Perplexity	Zero Probs in Test
300-factor SVD	432135.880	320161.981	0

**Table 3**

From table 3 we could see that the log model does not give good perplexity. This is mainly because we have a lot of large numbers in the log matrix: 99 for the original zero entries. By observing the magnitude of the singular values, we find that the smallest singular value we used is 2006.479. And recall that  $\min_{r(A)=s} \|W - A\|_F^2 = \|W - \hat{W}\|_F^2 = \sigma_{s+1}^2 + \dots + \sigma_m^2$ , so the overall error for the log matrix is quite large. The SVD output from Kneser-Ney smoothing probabilities also does not show good performance. This is because we used the log likelihood to construct the input matrix, rather than the likelihood itself, so the reason for the poor performance is the same as the log model we just explained.

## Smoothing Analysis

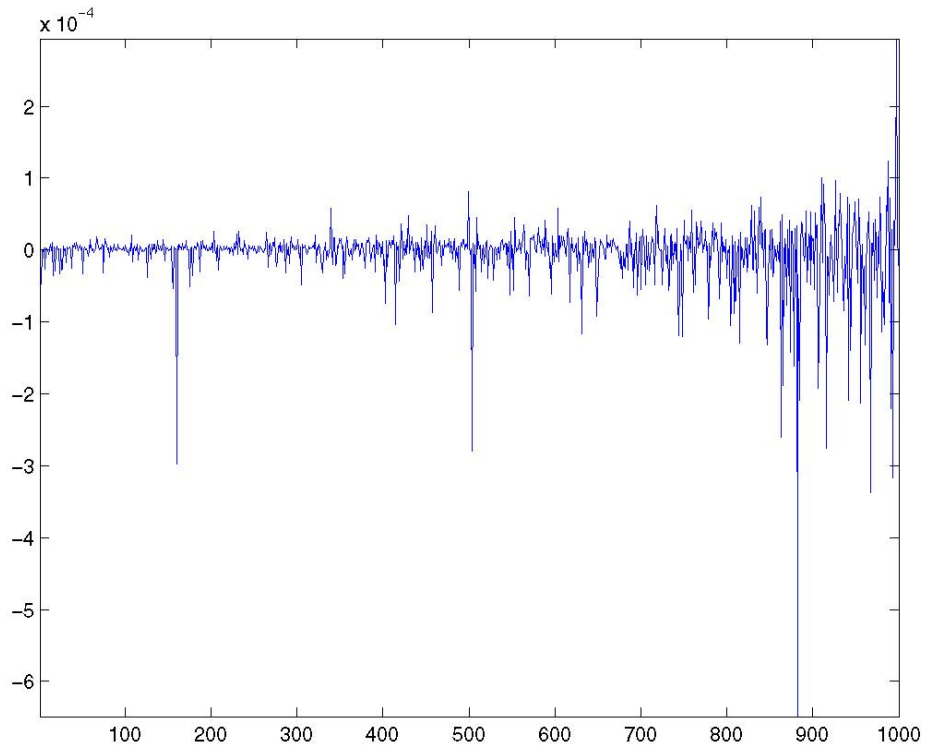
To further analyze our smoothing method using SVD, we sort the vocabulary by the word frequency in training data, and compute the error between the original word frequency and the one given after SVD. Figure 9 through figure 12 shows the relationship between word frequency and smoothing error. The x-axis represents the words in descending frequency order. The values in y-axis are calculated as an average error for one word with all the possible history words:

$$\frac{\sum_j error}{m}, \quad \text{where } error = \log\left(\frac{|w_{ij} - \hat{w}_{ij}|}{w_{ij}}\right)$$

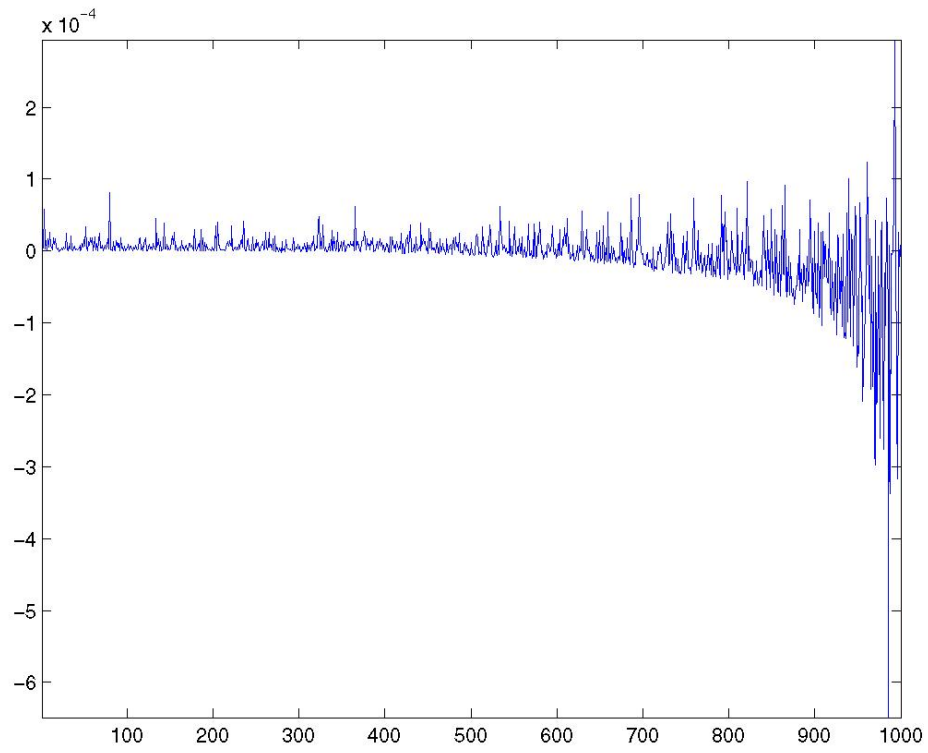
The plots have been sampled every 100 words, so they only have 1000 words each.

Figure 9 shows the error between raw frequency and its SVD result. There is a trend that small values get bigger relative errors. But this is satisfactory since the real difference for the small numbers are also small. Figure 10 evaluates the normalized-by-entropy method. We could see from the plot that the small values in the original matrix get big relative errors, while the large values get small relative errors. Thus, we need to see whether the real difference is large for the small values or not. Figure 11 is another plot showing the performance of normalized-by-entropy method. The values of y-axis are calculated by subtracting the word frequency from

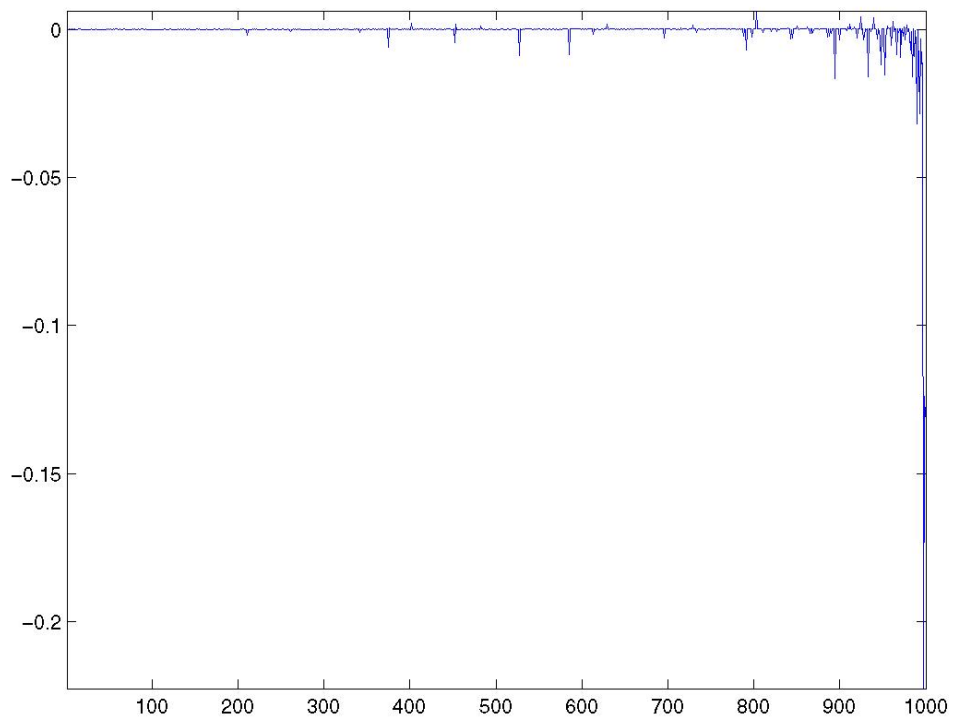
the numbers given by SVD, and then do the log scaling. It is now very clear that small frequencies are getting smaller errors, which is what we are looking for when we introduced the method.



**Figure 9: compare with raw count**

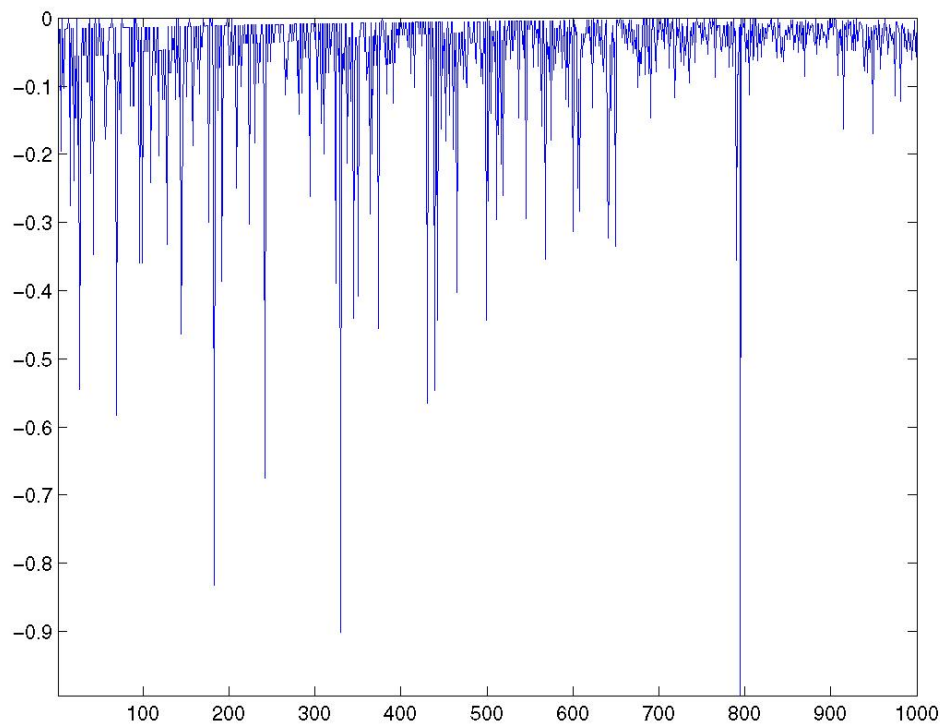


**Figure 10: compare with normalized-by-entropy**

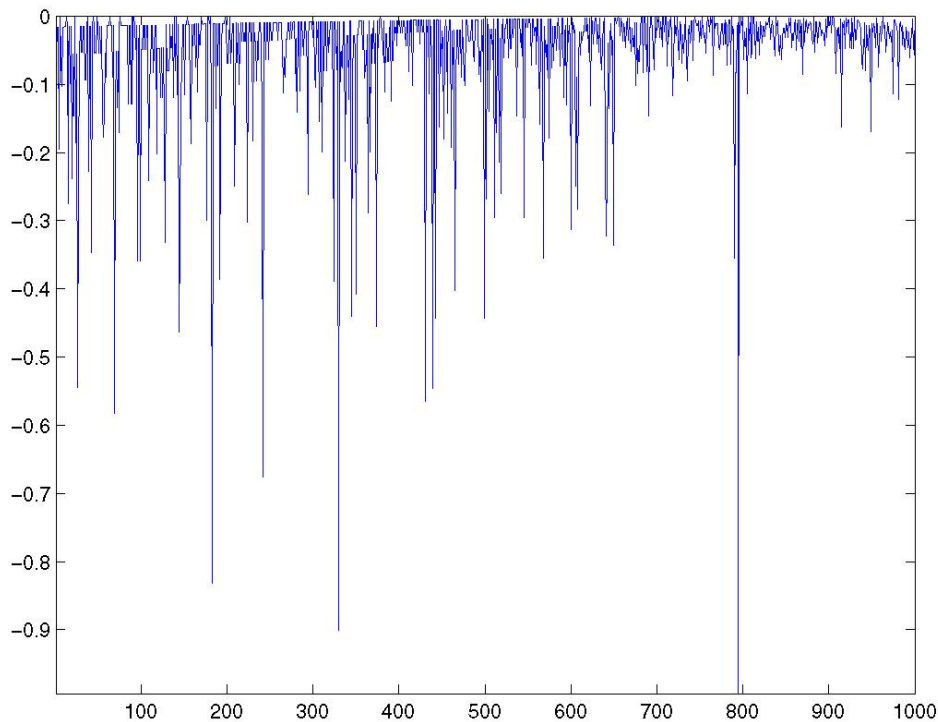


**Figure 11: compare with normalized count -  $\log(w^2-w)$**

Figure 12 shows the difference between normalized-by-entropy matrix after SVD and the probability values given by Kneser-Ney smoothing method. Notice that there are some words that are very well estimated (the long lines in the plot). The corresponding words are mostly person names: vargas, allocating, Charlie, Jacques Nashville, Jefferson, Harvey, Nancy, Ann, Susan, Messrs. But the numbers for other words seem to have large difference from the traditional bigram model. Figure 13 looks very similar to Figure 12. And it should resemble the result for the log model we mentioned earlier, since the input matrix is composed of the log likelihood from the traditional bigram model.



**Figure 12: compare new model with traditional bigram**



**Figure 13: compare SVD input and output from traditional bigram likelihoods**

## Trigram Experiments

Trigram Experiments are based on a fat co-occurrence matrix  $T$ , with  $T_{ij}$  = number of times word  $i$  appears after history bigram  $j$  in the training data. Since it would be too big for the matrix to include all the possible histories, we only use seen bigrams in training data, and a unique symbol is assigned to all unseen bigrams. The size of the co-occurrence matrix is  $10001 \times 754239$ . SVDPACKC only deals with tall matrix, so a transpose procedure in Harwell-Boeing Format is necessary before doing SVD. The matrix is even sparser than the bigram matrix, with 2254788 nonzero entries, which is about 0.03% of the entire matrix. There is a limitation for doing trigram matrix SVD when using SVDPACKC. It can only compute 150 singular values. Otherwise allocating memory would fail. We directly use the raw frequency matrix to feed SVD, and normalize columns of the output matrix to get probability values. Negative numbers obtained after SVD are set to be zero. Table 4 shows the performance of

trigram matrix:

	Training Perplexity	Test Perplexity	Zero Probs in Test
Good-Turing	65.5193	119.151	0
Kneser-Ney	65.2815	120.566	0
Trigram-150-SVD	872.962	893.955	6264

**Table 4**

Probably because our approximation error is too big since we used only 150-factor SVD, the perplexity we got is much higher than the traditional trigram models. Moreover, doing trigram experiments is a heavy burden for the machines, so no further experiments are done.

# Training the GMM

## The Basic Idea

After having vector representations for words, we are ready to train a continuous probability density function for language modeling. We choose to use Gaussian Mixture Model, so the probability of a word given its history is defined as:

$$P(w_i|h_j) = \sum_{k=1}^K P(w_i|C_k)P(C_k|h_j)$$

where

$$P(w_i|C_k) = \mathcal{N}(w_i; \underline{\mu}_k, \Sigma_k)$$

and  $K$  is the number of clusters.

We use EM algorithm to train the parameters of the GMM. The EM algorithm works by first choosing starting values  $\mu_k^{<0>}$ ,  $\Sigma_k^{<0>}$ ,  $P(C_k|h_j)^{<0>}$ , and then updating them iteratively according to the E-step and M-step.

## The Initial Step

$\mu_k^{<0>}$ ,  $\Sigma_k^{<0>}$ , and  $P(C_k|h_j)^{<0>}$  are chosen randomly<sup>1</sup>.

## The Expectation Step

$$P(C_k|h_j, w_i)^{<t+1>} = \frac{P(C_k|h_j)^{<t>} \mathcal{N}(w_i; \underline{\mu}_k^{<t>}, \Sigma_k^{<t>})}{\sum_{k=1}^K P(C_k|h_j)^{<t>} \mathcal{N}(w_i; \underline{\mu}_k^{<t>}, \Sigma_k^{<t>})}$$

## The Maximization Step

$$P(C_k|h_j)^{<t+1>} = \frac{\sum_{i=1}^M n(h_j, w_i) P(C_k|h_j, w_i)^{<t+1>}}{n(h_j)}$$

where

$n(h_j)$  total number of tokens after  $h_j$

---

<sup>1</sup> We will introduce other initialization methods in the later sections.

$n(h_j, w_i)$  number of the occurrences of the word  $w_i$  after history  $h_j$ .

$$\underline{\mu}_k^{(t+1)} = \frac{\sum_{j=1}^N \sum_{i=1}^M n(h_j, w_i) P(C_k | h_j, w_i)^{(t+1)} (u_i S)}{\sum_{j=1}^N \sum_{i=1}^M n(h_j, w_i) P(C_k | h_j, w_i)^{(t+1)}}$$

$$\Sigma_k^{(t+1)} = \frac{\sum_{j=1}^N \sum_{i=1}^M n(h_j, w_i) P(C_k | h_j, w_i)^{(t+1)} ((u_i S) - \underline{\mu}_k^{(t+1)}) ((u_i S) - \underline{\mu}_k^{(t+1)})^T}{\sum_{j=1}^N \sum_{i=1}^M n(h_j, w_i) P(C_k | h_j, w_i)^{(t+1)}}$$

## Unigram Model Definition

In order to find a good initialization point for the bigram model, we first train a unigram language model on the word vector space. We define a single state HMM called “word”, and allow the state emit all the words in the vocabulary. The HMM is illustrated in Figure 14.



Figure 14: “word”

In this research, we use HTK to help training HMM. Thus, all our data should be translated into HTK format files.

## Data preparation

As stated earlier, the vector representation of the  $i^{th}$  word is defined as the  $i^{th}$  row of  $US$  from the SVD output. These vectors appearing in training text should be written one by one into htk format parameter files. The htk format parameter file has a header which indicates the sample rate, number of samples, vector size, and the file format. It is also written as a binary file. Since a lot of HTK tools cannot deal with large files, so splitting the training data by the document boundaries is necessary. Now we get 9339 training files, each of which contains data only from a single document.

We could then check the content of the files by Hlist. It shows the header and the word vectors:

```
----- Source: 11.usr -----
Sample Bytes: 2000      Sample Kind:  USER
Num Comps:    500      Sample Period: 0.1 us
Num Samples:  209     File Format:   HTK
----- Samples: 0->1 -----
0:      0.000  0.000  0.000  0.000  0.000  0.000  0
        0.000  0.000  0.000  0.000  0.000  0.000  0
```

Figure 15

An MLF file (Master Label File) is required for supervised training as a transcript for the network. Since we only have one HMM in the unigram model, our MLF file only consists of “word” for each vector. Figure 15 shows the content of the MLF file.

```
MLF!#
"/1.lab"
word
word
word
word
word
word
word
word
word
word
word
word
word
```

Figure 16

## Training

Given a HMM prototype, we do a flat start using HCompV which computes global mean and variances from the entire training data. We also set a floor on the variance during this procedure to prevent over fitting. We start with a single Gaussian mixture component and then we are able to train the HMM incrementally.

Our training strategy is like the following: we use HERest to do training for 4 times, and then increment the number of mixture components using HHed. It has been suggested in HTK book that HERest are commonly used of 2 to 5 times for training. So we also tried doing training only for 2 times. Results show that this makes little

difference for the training procedure. For example, when we reached 100 mixture components, the average log probability per frame is 2.665236e+03 for the “2 times” training, and 2.668681e+03 for the “4 times” training. The results we show in this report are all done by the “4 times” training.

After splitting training data and accelerating HERest by parallel computing on 10 machines, HERest takes around 6-10 minutes per iteration of training.

HHed uses the *MU* command to increment the number of Gaussian mixtures:

```
MU 2 {*.state[2].mix}
```

Now many mixture components should be added at the HHED step depends on the number of mixtures before the increment. In general, we could make a larger step when we have only a few mixture components, and a small step afterwards. In our experiments, the number of mixtures is incremented in this sequence: 1, 2, 4, 8, 16, 30, 40, 50. We then add 10 mixture components each time after 50 mixtures.

When HHed do the mixture increment, it first split the mixture with the heaviest score, then clones the mixture component, divides the weight by 2, and perturb the means by  $\pm 0.2$  standard deviation. The score of a mixture component is mainly determined by its weight. But when a mixture component has been divided too many times, the score will drop down. In this procedure, a mixture component may become defunct. In some cases, a defunct mixture will be replaced by some other mixture components and HHed can still increase the number of mixtures to the number required; but in some other cases, the number of mixtures required cannot be reached, which indicates over fitting.

One of our HMM has been developed to have 200 mixture components, we could see the increasing of likelihood from the results below. Notice that there are defunct mixture components during the mixture incrementing, so although we asked for 50 mixtures, HHed gave us 48 non-defunct components. But this has been corrected

during the following step of mixture incrementing.

For 200 mixtures, there are 71866 floored variance elements. Since we used the diagonal covariance matrix, and our vector size is 500, each mixture component has 500 variance elements. Thus, there are approximately 70% variance elements been floored. This may already indicates over fitting.

One other clue that indicates over fitting is the  $GConst$  value defined at the end of each mixture component. A small  $GConst$  value means very low variance in the Gaussian mixture component. The  $GConst$  value is calculated as:

$$GCONST_i = (2\pi)^n \det(\Sigma_i).$$

#### Training Results:

Number of Mixture: 1

average log prob per frame = 1.643967e+03

Number of Mixture: 2

Total 21 floored variance elements in 2 different mixes

average log prob per frame = 1.833349e+03

Number of Mixture: 4

Total 151 floored variance elements in 4 different mixes

average log prob per frame = 2.212935e+03

Number of Mixture: 8

Total 664 floored variance elements in 8 different mixes

average log prob per frame = 2.364563e+03

Number of Mixture: 16

Total 2088 floored variance elements in 16 different mixes

average log prob per frame = 2.480190e+03

Number of Mixture: 30

Total 5927 floored variance elements in 30 different mixes

average log prob per frame = 2.581880e+03

Number of Mixture: 40

Total 9269 floored variance elements in 40 different mixes

average log prob per frame = 2.606847e+03

Number of Mixture: 50

Total 12322 floored variance elements in 48 different mixes

average log prob per frame = 2.625593e+03

Number of Mixture: 100

Total 30258 floored variance elements in 100 different mixes

average log prob per frame = 2.668681e+03  
Number of Mixture: 150  
Total 48930 floored variance elements in 148 different mixes  
average log prob per frame = 2.701295e+03  
Number of Mixture: 170  
Total 57326 floored variance elements in 170 different mixes  
average log prob per frame = 2.723302e+03  
Number of Mixture: 200  
Total 71866 floored variance elements in 200 different mixes  
average log prob per frame = 2.744611e+03

## Testing

After each iteration of training or mixture incrementing, the new parameter values obtained, including the definition of the whole HMM are stored in some HMM macro files. Then we are able to use the parameters to calculate likelihoods for each word. HTK provides a tool called HVite to do testing. It will output the likelihood for each recognized word and store them into a recognition file, in the same order as the test data file. Since we are using a unigram model, which does not depend on any context information, we could run HVite on a vocabulary file to get the likelihood for each word.

Before doing HVite, we need to build a network which defines the allowed path for traversing the trellis of HMM during testing. In other words, a network describes the sequence of words that can be recognized. To build such a network, we will first write a grammar file, and then compile it to a word network file using the HParse tool. Our grammar only accepts sentences consisted with “word”, so the only line in our grammar file is: (<WORD>).

A dictionary file defines a mapping from the word to the corresponding HMM's. We only have a one-to-one mapping here: “WORD word”. Thus, we could get a HMM level network from the word level network and the mapping. Since each HMM already contains a network definition for the states, we finally get a fully compiled state level network and is ready for testing.

Form the likelihood of test data reported below (used the HMM definitions obtained from the training above), we could see that the likelihood is close to that of the training data.

#### Testing Results (average likelihood):

Number of Mixture: 1  
average log likelihood per sample: 1646.4020585513254  
length: 154025  
Number of Mixture: 2  
average log likelihood per sample: 1891.3547502669887  
Number of Mixture: 4  
average log likelihood per sample: 2257.963955406453  
Number of Mixture: 8  
average log likelihood per sample: 2365.6243727869796  
Number of Mixture: 16  
average log likelihood per sample: 2478.6826059498244  
Number of Mixture: 30  
average log likelihood per sample: 2580.262006589316  
Number of Mixture: 40  
average log likelihood per sample: 2603.4247704931595  
Number of Mixture: 50  
average log likelihood per sample: 2621.5510810525257  
Number of Mixture: 100  
average log likelihood per sample: 2672.685428929045  
Number of Mixture: 150  
average log likelihood per sample: 2697.4258390083737  
Number of Mixture: 200  
average log likelihood per sample: 2741.9203684773893

The perplexity we obtained directly from likelihood does not sum to one, so we need to map likelihood to PMF. The straightforward approach is to directly scale the likelihood to make them sum to one. This method gives us a large number of around  $e^{160}$  for perplexity.

One method I used is to approximate integral on a small 500 dimensional window. To choose the size of the small window, I find the variance for each dimension, and chose the window size of that dimension to be one fourth of the variance. This gives

me a perplexity from  $e^{160}$  to  $e^{56}$ .

Another approach is to scale log likelihood by a common factor K. In other words, we divide the log likelihood by K and then normalize them to obtain probability values. This gives us a perplexity number which is close to uniform distribution.

## **Semi-Tied Transform**

Since we did not get reasonable perplexity on test data, we doubt whether our model is trained accurately or not. One thing that has been brought to our attention is the diagonal covariance matrix, which is the HTK default for training. This, in our case, assumes words are independent given different histories, which is obviously not true. We would like to relax this restriction and use full covariance matrix. However, calculating full covariance matrix is extremely time consuming, so we find a trade-off between the diagonal and the full covariance matrix.

A global Semi-Tied Transform is hence incorporated in our model. In the HMM, each mixture component has its own diagonal covariance matrix, but share a common transform to make them full covariance matrices. The actual covariance matrix is the Semi-Tied Transform times the diagonal covariance matrix.

A “global” macro is required to define the baseclass, another macro is also needed to store the trained Semi-Tied transform. The Semi-Tied transform is trained once at some point of the training procedure and been fixed. The following training iterations only train the diagonal covariance matrices. We have tried training Semi-Tied transform in each iteration. It shows there are two problems here. The first one is that the training time for Semi-Tied transform is very long, the second problem is the result turns out to be a hierarchical transform, which means a parent transform would be added to the original Semi-Tied transform. Suppose the original transform is S, now another transform P is attached, so the actual Semi-Tied transform is now P\*S.

Some other things are needed to be done before training the Semi-Tied transform. The HTK code only supports Semi-Tied transform of size 100\*100. We need to set the constant to be at least 500 to do our experiment or other large size vector experiments. When running the experiment or simply reading the HTK code, it is not hard to find that training the Semi-Tie transform requires large memory. This memory requirement comes from creating a list of matrices (500 in total) with size 500\*500 when updating the transform. HTK uses double precision, so the update uses  $500*500*500*8=1\text{G}$  memory.

Before we go any further and introduce an alternative to Semi-Tied transform, we shall evaluate the performance of the Semi-Tied transform. One question is where to incorporate the Semi-Tied transform. To see this, we do three sets of experiments: one starts from the first iteration of training, one starts from 16 mixture components, another one starts from 50 mixture components. Results show that the perplexity before any scaling drops after Semi-Tied transform has been incorporated, but the scaled version doesn't make any difference.

### **Starting from the first iteration of training:**

Training:

Semi-Tied

Number of Mixture: 1

average log prob per frame = 1.022683e+03

Number of Mixture: 2

Total 230 floored variance elements in 2 different mixes

average log prob per frame = 1.245610e+03

Number of Mixture: 4

Total 497 floored variance elements in 4 different mixes

average log prob per frame = 1.349340e+03

Number of Mixture: 8

Total 1056 floored variance elements in 8 different mixes

average log prob per frame = 1.379674e+03

Number of Mixture: 16

Total 2430 floored variance elements in 16 different mixes

average log prob per frame = 1.404705e+03

Number of Mixture: 30

Total 5391 floored variance elements in 30 different mixes

average log prob per frame = 1.421529e+03

Number of Mixture: 40

Total 7890 floored variance elements in 40 different mixes

average log prob per frame = 1.431857e+03

Number of Mixture: 50

Total 10517 floored variance elements in 50 different mixes

average log prob per frame = 1.439504e+03

**Test:**

Number of Mixture: 1

average log likelihood per sample: 1035.2562123845078

Number of Mixture: 2

average log likelihood per sample: 1253.964321292814

Number of Mixture: 4

average log likelihood per sample: 1352.1057977187188

Number of Mixture: 8

average log likelihood per sample: 1380.6572633653957

Number of Mixture: 16

average log likelihood per sample: 1405.3157488397019

Number of Mixture: 30

average log likelihood per sample: 1422.5985901944505

Number of Mixture: 40

average log likelihood per sample: 1431.929505284274

Number of Mixture: 50

average log likelihood per sample: 1439.6513457420015

**Incorporate Semi-Tied transform at 16 mixtures:**

**Training:**

Number of Mixture: 16

Total 5842 floored variance elements in 16 different mixes

Average log prob per frame = 2.443782e+03

Number of Mixture: 30

Total 11262 floored variance elements in 30 different mixes

Average log prob per frame = 2.454201e+03

Number of Mixture: 40

Total 15492 floored variance elements in 40 different mixes

Average log prob per frame = 2.472738e+03

Number of Mixture: 50

Total 20102 floored variance elements in 50 different mixes

Average log prob per frame = 2.479595e+03

**Test:**

Number of Mixture: 16  
average log likelihood per sample: 2446.513  
Number of Mixture: 30  
average log likelihood per sample: 2463.059  
Number of Mixture: 40  
average log likelihood per sample: 2475.995  
Number of Mixture: 50  
average log likelihood per sample: 2480.632

### Incorporate Semi-Tied system at 50 mixtures:

#### Training:

Number of Mixture: 50  
Total 22285 floored variance elements in 50 different mixes  
Average log prob per frame = 2.301666e+03  
Number of Mixture: 60  
Total 26938 floored variance elements in 60 different mixes  
Average log prob per frame = 2.303622e+03  
Number of Mixture: 70  
Total 31450 floored variance elements in 70 different mixes  
Average log prob per frame = 2.305963e+03  
Number of Mixture: 80  
Total 36395 floored variance elements in 80 different mixes  
Average log prob per frame = 2.306537e+03  
Number of Mixture: 90  
Total 40786 floored variance elements in 90 different mixes  
Average log prob per frame = 2.308693e+03  
Number of Mixture: 100  
Total 45746 floored variance elements in 100 different mixes  
Average log prob per frame = 2.309216e+03

#### Test:

Number of Mixture: 50  
average log likelihood per sample: 2302.405  
Number of Mixture: 60  
average log likelihood per sample: 2304.461  
Number of Mixture: 70  
average log likelihood per sample: 2306.758  
Number of Mixture: 80  
average log likelihood per sample: 2307.274  
Number of Mixture: 90  
average log likelihood per sample: 2309.329  
Number of Mixture: 100  
average log likelihood per sample: 2309.800

By observing the HTK code for updating the Semi-Tied transform, we find that the transform is initialized by the identity matrix, which is obviously too far away from the real transform to approximate the full covariance matrix. Thus, we may need some other technique to initialize the transform or avoid training it.

One method is to compute a transform to diagonalize the covariance matrix for the entire training data, and thus avoid training the Semi-Tied transform at the single mixture GMM. First we use HCompV to compute full covariance matrix  $\Sigma$  of the training data.  $\Sigma$  turns out not to be diagonal, nor is it singular (we used the raw count co-occurrence matrix to get the word representations), but we could diagonalize it by transforming the data. We do eigenvalue decomposition on  $\Sigma$ :  $\Sigma = U\Lambda U^T$ , and then  $U$  could be used as the Semi-Tied transform  $S$ , and  $\Lambda$  should be the diagonal covariance matrix. Note that  $UU^T = I$ , we could then transform the data to let them have a diagonal covariance matrix. If we call our original word vector  $w$ , the transformed vector would now become  $y = S^T w = U^T w$ . Now we have the transformed data with diagonal covariance matrix  $\tilde{\Sigma} = U^T \Sigma U = U^T U \Lambda U^T U = \Lambda$ .

Doing eigenvalue decomposition is fast. And the only thing we need to do is to write the obtained transform matrix to be compatible with HTK. Also notice that when using HCompV for computing full covariance matrix, the matrix is actually stored in them inverse form. But fortunately, we are only interested in the eigenvectors which gives the transform, and one matrix and its inverse have the same eigenvectors:

$\Sigma^{-1} = U\Lambda^{-1}U^T$ . So no extra efforts are needed for dealing with this problem.

When we have the transform to diagonalize the training data, we could start training the single mixture GMM without Semi-Tied transform. And when we get to some point (like 16 mixtures or 50 mixtures), we could incorporate Semi-Tied transform. And this time, we are not using the identity matrix to initialize training. Rather, we are

using matrix  $U^T$ , which is considered to be closer to the true transform.

When doing testing, we do not need to transform the data; we simply compute the real Semi-Tied transform by  $SU^T$ , where S is the Semi-Tied transform trained by HTK, and then do testing on the raw test data.

The method stated above not only gives us a better approximation of Semi-Tied transform, it also saves a lot of time training the transform. In HTK, the Semi-Tied transform is trained in a doubly iterative process. The inner loop updates one row of the transform at a time. Once at least the whole matrix has been updated, the outer loop updates the diagonal covariance and other model parameters. Usually in our case, we set the number of iterations for the inner loop to be double the vector size, which is 1000. And after 10 iterations of the outer loop, 80% of the rows stop to change.

Although we save time training the Semi-Tied transform, computing the full covariance matrix using HCompV is slow. Table 5 shows the running time for computing full covariance matrix for 300 and 500 dimensional data vectors:

<b>Size of the Data Vectors</b>	<b>Running Time</b>
300	30min 39sec
500	84min 27sec

**Table 5**

Our next experiment tests the performance of the “diagonalize data” method, and Semi-Tied transform is trained at 50 mixture components. The results below show the likelihood of training data after semi-tied transform has been added:

Number of Mixtures: 50  
average log likelihood per sample: 2245.6459605656673

Number of Mixtures: 60  
average log likelihood per sample: 2247.4409418537593

Number of Mixtures: 70  
average log likelihood per sample: 2248.635507320272

Number of Mixtures: 80  
average log likelihood per sample: 2249.5347559843617

Number of Mixtures: 90  
average log likelihood per sample: 2250.6359807107146

Number of Mixtures: 100  
average log likelihood per sample: 2250.6359807107146

We expect the Semi-Tied transform would give us a fast improvement on the likelihood when compared with the default covariance matrix, but the results are disappointing. The reason is unclear to us. Training Semi-Tied transform is a new function added to HTK 3.4, so more efforts should be put here for further understanding of the technique.

By searching for a scale factor  $K$  to minimize the perplexity, we find that we still cannot achieve better results than uniform distribution. However, the new method decreases the perplexity before any scaling from  $e^{100}$  down to  $e^{10}$ .

## **Vector Representations for the Words**

So far, we have not got a reasonable number for perplexity, and it is time to observe the distribution of the word vectors in detail. By doing k-means clustering, we find that it does not give out relatively balanced clusters. There is always a huge cluster which contains around 9000 words, and other clusters only has several words in them. The huge cluster are mainly consisted of the infrequent words which carries too little semantic or syntactic information. The infrequent words are tied together to resemble a frequent word and get high likelihood during training. To avoid this, a more balanced structure for the vector space is desired.

Notice that we were using the raw count vectors in our previous experiments, and we believe some other possible vector representation may give us more balanced clusters. To find a good structure for the word vector space, four representations have been tried. We examine the construction of the input matrix for SVD:

1. Raw count:  $w'_{ij} = w_{ij}$
2. Divide by the square root of the sum:  $w'_{ij} = \frac{w_{ij}}{\sqrt{\sum_i w_{ij}}}$
3. Divide by the vector length:  $w'_{ij} = \frac{w_{ij}}{\sqrt{\sum_i w_{ij}^2}}$
4. Divide by the sum:  $w'_{ij} = \frac{w_{ij}}{\sum_i w_{ij}}$

Each of the above trials has been done for training and testing. And the vocabulary is sorted by the likelihood given by each word representation. The results show a big difference between the ordering by the likelihood and the original frequency order given directly from training data. Table 6 lists the likelihood rank (in descending order) for the most frequent word “the” for each representation:

Method	Raw Count	Divide-By-Sqrt-Of-Sum	Divde-By-Length	Divide-By-Sum
Rank of “the”	8401	8889	8481	8935

Table 6

Since raw count seems to work better, we focus more on the vector distribution given by this representation. When doing K-Means clustering, we notice that using cosine distance gives balanced clusters, so we scale the length of the word vectors to be unity. Table 7 shows the number of words in each cluster. We grouped the words into 40 clusters:

cluster 1	338
cluster 2	195
cluster 3	457
cluster 4	386
cluster 5	124
cluster 6	128
cluster 7	68
cluster 8	60
cluster 9	172
cluster 10	6
cluster 11	469
cluster 12	252
cluster 13	194
cluster 14	310
cluster 15	226
cluster 16	396
cluster 17	329
cluster 18	33
cluster 19	68
cluster 20	931
cluster 21	28
cluster 22	602
cluster 23	649
cluster 24	535
cluster 25	319
cluster 26	16
cluster 27	329
cluster 28	35
cluster 29	34
cluster 30	498
cluster 31	202
cluster 32	282
cluster 33	224
cluster 34	155
cluster 35	0
cluster 36	302
cluster 37	284
cluster 38	70
cluster 39	110
cluster 40	185

**Table 7**

After training and testing, we could get a clear view of the word vector distribution defined by GMM. This is illustrated in table 8. For each Guassain mixture, the first two columns shows the mixture number and mixture weight, the second part lists information for the mean word (the word is closest to the mean vector), it contains the name of the word, the likelihood obtained, the cluster id,

and the rank of the word in its original frequency (small number indicates high frequency). The GCONST value gives information for the variance. A small GConst indicates small variance element values. Please notice that table 8 only lists part of the GMM information. The whole GMM contains 50 mixtures.

Mix No.	Weight	Mean	Likelihood	Cluster ID	Freq Rank	GConst
1	3.651464e-03	Quarter	595.462280	29	107	-1.324552e+03
2	2.513627e-03	Fifteen	810.139038	25	220	-7.180728e+02
3	1.117004e-02	Year	786.058960	27	47	-1.624088e+03
4	3.966968e-04	commercials	-1182.369507	41	3189	1.251720e+03
5	1.205066e-02	Sarney	698.691345	50	9194	-1.663337e+03
6	8.877443e-03	Biden	-530.381165	50	8387	-1.675832e+03
7	6.191295e-03	Limited	-140.587250	24	311	-4.670174e+02
8	4.876575e-04	Familiar	-2409.776855	28	1299	3.619238e+03
9	1.016240e-02	Nast	-543.275208	28	9250	-1.675832e+03

**Table 8**

The following table also shows the information for each word, including the likelihood, cluster id and frequency rank. Then words are sorted by their likelihood values, in descending order. After normalized the length of the word vectors, we clearly get a much better likelihood. At least the most frequent word “the” have almost reached its frequency position, rather than been squeezed to number 8000 or 9000.

Also note that except for the first two words, the other words in top 20 all come from the same cluster. This trend is always true for my following experiments.

Word	Likelihood	Cluster ID	Freq Rank
<s>	836.872986	28	3
the	836.698486	33	1
ironically	836.575134	6	6268
industrielle	836.532837	6	6012
similarly	836.528809	6	3246
unfortunately	836.518127	6	4479
nevertheless	836.50769	6	2635
indeed	836.501343	6	1114
separately	836.499329	6	1481
postscripts	836.490356	6	8445
daffynition	836.490356	6	9949
fortunately	836.488403	6	8104
nonetheless	836.479309	6	2690
moreover	836.478882	6	1236
but	836.467224	6	38
pantera	836.460571	6	8260
mr	836.459961	6	27
furthermore	836.453308	6	5511
meanwhile	836.449768	6	789
besides	836.448914	6	2343

**Table 9**

Now we keep the raw count representation, and add two more experiments based on SVD: the log variation and the normalize-by-entropy variation.

Before doing SVD, we replace the raw counts  $w_{ij}$  in the co-occurrence matrix with  $1 + \log(w_{ij})$ , and leave the zero's unchanged. The reason to add 1 is to avoid the ambiguity of a zero entry: the number of zero entries would increase since the original "1" would be replaced by zero, while the original zero's remain zero's.

Semi-Tied transform has also been tried for training, but it fails because of singular covariance matrix at the first stage when we use HCompV to find the full covariance of the training data.

This method give out balanced clustering, even without scaling the length of the vectors, but the perplexity we got is worse than raw count representation. The top 20

words which received highest likelihood are listed below, with other useful information. The likelihood ranking is not reasonable in this case.

Notice that <s> is always ranked first. This is because most of the vector elements are close to zero, so the zero vector is always closest to the global mean of the data set and thus get highest likelihood. This may cause a problem for assigning likelihood. Even when training without <s>, we still get highest probability for <s>. And those words which are close to <s> will also be assigned high probability.

Word	Likelihood	Cluster ID	Freq Rank
<s>	139.914093	27	3
nast	139.914093	27	9250
femina	139.914093	27	6373
poulenc	139.901901	27	6812
ichi	139.84903	27	3394
coaster	139.815918	27	9262
marwick	139.776794	27	9645
wedd	139.774765	27	9056
vegas	139.751053	27	4151
pont	139.733063	27	3253
jaffray	139.714478	27	8334
cetera	139.605301	27	6675
rica	139.595428	27	5890
lufkin	139.496307	27	3875
rican	139.451035	27	7048
agers	138.980179	27	9482
cone	138.967407	27	9833
donuts	138.94397	27	6203
clara	138.748138	27	8182
jolla	138.629669	27	8488

**Table 10**

The normlize-by-entropy matrix setup has been introduced in the former sections. And here we need to evaluate the performance of the word vectors in detail. One thing that is different from our old procedure is that we do not reconstruct the matrix by multiplying the reciprocals of the entropy. We calculate word vectors directly from the output of SVD.

We do not need to go too far for this experiment, since it did not give out good

clustering. After grouping the word vectors into 100 clusters, one dominant cluster contains 9479 words, which will obviously cause the problem we mentioned before.

## Non-negative Matrix Factorization

Although we find a reasonable clustering by using SVD to project the word vectors to lower dimensional space, we were not able to train a desirable GMM on this space. Also, SVD aims at the least square error, while we are seeking a good approximation for the distribution, which requires a least divergence. Moreover, the negative values obtained after SVD is always a matter that we have to deal with. Fortunately Non-negative Matrix Factorization (NMF) could help us with all these problems.

Given a Non-negative matrix  $V$ , the goal of NMF is to find two non-negative matrices  $W$  and  $H$  such that  $V \approx WH$ , and minimize  $D(V || WH)$  with respect to  $W$  and  $H$ .

In Lee and Seung's paper (2001), they proposed multiplicative update rules for searching for the optimal solution of the NMF problem. The proof of convergence is done by defining an auxiliary function and turn the problem into minimizing the auxiliary function in each update. From the update rules and the proof, we could learn that an increment of dimensionality does not simply add new columns or rows to the factor matrices, but affects the entire matrices. The update rules are listed below:

$$H_{ij} \leftarrow H_{ij} \frac{\sum_k W_{ki} V_{kj} / (WH)_{kj}}{\sum_a W_{ai}}$$

$$W_{ij} \leftarrow W_{ij} \frac{\sum_k W_{jk} V_{ik} / (WH)_{ik}}{\sum_a W_{ja}}$$

Like SVD, rows of  $H$  are treated as a set of basis which span the entire vector space. And rows of  $W$  could be viewed as some encoding which define linear combinations of the set of basis. Thus, rows of  $W$  are used as the word vectors. Another nice

property of NMF is that W and H are also sparse matrices, thus may speed up our computations.

Our generic program for full matrix runs on 500-factor NMF for less than 30 minutes each update. Since we use sparse matrix, we can be 50 times faster. So our program for running NMF on sparse matrices runs for less than one minute in each update. Note that our matrix size is 10001\*10001, and the number of nonzero entries is 762323.

The input of our NMF program is a matrix stored in Harwell-Boeing Format. The output is a binary file which stores two matrices: W and H. We also provide one log file tracking the divergence between the original matrix and the approximation (WH).

With the newly developed NMF tool, we could get a set of new word vectors and do experiments on that. Even when using NMF, the clustering for Euclidean distance is not good, it contains a cluster which has around 9000 words. And this cluster gives little semantic or syntactic information. The other clusters contains only a few words, but reveals good semantic or syntactic information.

We now do training and testing both on the raw length vectors given by NMF and on the vectors which have been normalized to have unity length. The results further convinced us good clustering indeed gives out smaller perplexity.

The last step for getting perplexity is to floor all the probability value to be larger than or equal to  $10^{-6}$ . After we get the probability values, we solve for a small value  $\varepsilon$

from  $\frac{\varepsilon}{\sum_{i:q_i>\varepsilon} q_i + k\varepsilon} = 10^{-6}$ , where k is the number of probability values which are

smaller than  $10^{-6}$ . Then we assign  $\varepsilon$  to all the small values which are smaller than  $10^{-6}$ . And finally we could renormalize the numbers to make them sum to one.

After scaling by a scale factor K and flooring to have all the probability values no smaller than 10e-6, we get a perplexity of 1501.87, while K=150. The traditional unigram is 731.984, which is almost half of our result. The table below shows the perplexity we got for different number of mixtures. The K value is the scale factor to obtain the best perplexity.

Number of Mixes	Perplexity	K
16	1769.526714	170
40	1544.852033	150
50	1507.960686	150
60	1556.137086	1030
70	1596.388671	150
100	1601.274826	140

**Table 12**

The following several pages shows the result we get for the GMM at 50 mixes. There are three lines for each mixture component: the first line gives the mixture id and the mixture weight. The second line gives information about the mean word which is closest to the mean vector, including the likelihood of the word, the cluster id and the frequency rank of the word. The third line gives information about variance. A small GConst value indicates very low variance elements, as stated before.

**50 Mixes:**

```

<MIXTURE> 1 1.852973e-02
<MEAN> environmental      2068.968018      cluster 20      freq_rank 938
<GCONST> -3.668207e+03

<MIXTURE> 2 1.277492e-02
<MEAN> should              2252.543701      cluster 12      freq_rank 282
<GCONST> -4.694461e+03

<MIXTURE> 3 2.048583e-02
<MEAN> <unk>                2702.929199      cluster 12      freq_rank 2
<GCONST> -5.412248e+03

<MIXTURE> 4 3.876713e-03

```

<MEAN> out	2619.592041	cluster 12	freq_rank 93
<GCONST> -5.285996e+03			
<MIXTURE> 5 1.973865e-02			
<MEAN> first	2223.209229	cluster 12	freq_rank 80
<GCONST> -4.539771e+03			
<MIXTURE> 6 2.135876e-02			
<MEAN> for	2647.122070	cluster 12	freq_rank 11
<GCONST> -5.317004e+03			
<MIXTURE> 7 2.623928e-02			
<MEAN> clean	1860.998413	cluster 6	freq_rank 1951
<GCONST> -4.049065e+03			
<MIXTURE> 8 1.721405e-02			
<MEAN> i	2443.287354	cluster 39	freq_rank 67
<GCONST> -5.058216e+03			
<MIXTURE> 9 4.986325e-02			
<MEAN> the	2692.822266	cluster 39	freq_rank 1
<GCONST> -5.391776e+03			
<MIXTURE> 10 4.804777e-03			
<MEAN> percent	2695.482422	cluster 12	freq_rank 23
<GCONST> -5.401658e+03			
<MIXTURE> 11 2.329499e-02			
<MEAN> to	2702.364502	cluster 12	freq_rank 6
<GCONST> -5.412248e+03			
<MIXTURE> 12 1.552622e-02			
<MEAN> recommended	1972.976318	cluster 17	freq_rank 2443
<GCONST> -4.424557e+03			
<MIXTURE> 13 2.604488e-02			
<MEAN> environmental	2068.968018	cluster 20	freq_rank 938
<GCONST> -4.295187e+03			
<MIXTURE> 14 2.206335e-02			
<MEAN> hear	2459.100830	cluster 6	freq_rank 2257
<GCONST> -5.001510e+03			
<MIXTURE> 15 7.243808e-03			

<MEAN> cs	2498.681396	cluster 12	freq_rank 4331
<GCONST> -5.188219e+03			
<MIXTURE> 16 2.401846e-02			
<MEAN> environmental	2068.968018	cluster 20	freq_rank 938
<GCONST> -3.952124e+03			
<MIXTURE> 17 2.897298e-02			
<MEAN> mr	2391.810791	cluster 12	freq_rank 27
<GCONST> -4.853081e+03			
<MIXTURE> 18 3.268174e-03			
<MEAN> s	2700.400391	cluster 12	freq_rank 42
<GCONST> -5.412248e+03			
<MIXTURE> 19 2.685182e-02			
<MEAN> that	2599.999268	cluster 12	freq_rank 10
<GCONST> -5.238819e+03			
<MIXTURE> 20 2.973766e-02			
<MEAN> is	2612.602783	cluster 12	freq_rank 13
<GCONST> -5.258417e+03			
<MIXTURE> 21 1.763766e-02			
<MEAN> children	2107.872314	cluster 43	freq_rank 841
<GCONST> -4.436559e+03			
<MIXTURE> 22 2.545489e-02			
<MEAN> present	1883.739258	cluster 20	freq_rank 1517
<GCONST> -4.520100e+03			
<MIXTURE> 23 1.901498e-02			
<MEAN> more	2288.736084	cluster 12	freq_rank 62
<GCONST> -4.631313e+03			
<MIXTURE> 24 2.260327e-02			
<MEAN> environmental	2068.968018	cluster 20	freq_rank 938
<GCONST> -4.552328e+03			
<MIXTURE> 25 2.032490e-02			
<MEAN> </s>	2702.921143	cluster 12	freq_rank 4
<GCONST> -5.412248e+03			
<MIXTURE> 26 2.391045e-02			

<MEAN> allen	1974.205200	cluster 26	freq_rank 2984
<GCONST> -4.181859e+03			
<MIXTURE> 27 2.679343e-02			
<MEAN> it	2529.896729	cluster 12	freq_rank 15
<GCONST> -5.113010e+03			
<MIXTURE> 28 7.744491e-03			
<MEAN> point	2660.734619	cluster 12	freq_rank 14
<GCONST> -5.333421e+03			
<MIXTURE> 29 2.629978e-02			
<MEAN> august	2289.834961	cluster 39	freq_rank 244
<GCONST> -4.701894e+03			
<MIXTURE> 30 2.493159e-02			
<MEAN> eight	2478.658691	cluster 12	freq_rank 37
<GCONST> -5.048692e+03			
<MIXTURE> 31 2.825879e-02			
<MEAN> a	2576.745605	cluster 12	freq_rank 7
<GCONST> -5.171952e+03			
<MIXTURE> 32 7.066310e-03			
<MEAN> would	2558.731201	cluster 12	freq_rank 61
<GCONST> -5.177064e+03			
<MIXTURE> 33 2.816259e-02			
<MEAN> of	2685.784180	cluster 12	freq_rank 5
<GCONST> -5.380110e+03			
<MIXTURE> 34 2.032490e-02			
<MEAN> in	2670.585449	cluster 12	freq_rank 9
<GCONST> -5.412248e+03			
<MIXTURE> 35 2.060356e-02			
<MEAN> and	2702.241699	cluster 12	freq_rank 8
<GCONST> -5.412248e+03			
<MIXTURE> 36 2.048583e-02			
<MEAN> <unk>	2702.929199	cluster 12	freq_rank 2
<GCONST> -5.412248e+03			
<MIXTURE> 37 1.770674e-02			

<MEAN> pact	2187.482178	cluster 12	freq_rank 1703
<GCONST> -4.655471e+03			
<MIXTURE> 38 2.240378e-02			
<MEAN> environmental	2068.968018	cluster 20	freq_rank 938
<GCONST> -4.230617e+03			
<MIXTURE> 39 2.265969e-02			
<MEAN> twenty	2470.836426	cluster 12	freq_rank 53
<GCONST> -5.024889e+03			
<MIXTURE> 40 1.955566e-02			
<MEAN> obviously	1904.978149	cluster 39	freq_rank 3125
<GCONST> -3.967174e+03			
<MIXTURE> 41 2.756151e-03			
<MEAN> all	2565.737305	cluster 12	freq_rank 97
<GCONST> -5.192271e+03			
<MIXTURE> 42 2.032490e-02			
<MEAN> </s>	2702.921143	cluster 12	freq_rank 4
<GCONST> -5.412248e+03			
<MIXTURE> 43 2.032490e-02			
<MEAN> in	2670.585449	cluster 12	freq_rank 9
<GCONST> -5.412248e+03			
<MIXTURE> 44 1.216868e-02			
<MEAN> by	2677.447510	cluster 12	freq_rank 24
<GCONST> -5.370698e+03			
<MIXTURE> 45 1.838858e-02			
<MEAN> in	2670.585449	cluster 12	freq_rank 9
<GCONST> -5.350200e+03			
<MIXTURE> 46 1.179097e-02			
<MEAN> as	2661.403809	cluster 12	freq_rank 22
<GCONST> -5.346886e+03			
<MIXTURE> 47 2.669533e-02			
<MEAN> one	2595.765137	cluster 12	freq_rank 12
<GCONST> -5.233608e+03			
<MIXTURE> 48 2.210815e-02			

<MEAN> discovered 2149.736084 cluster 17 freq\_rank 2447  
 <GCONST> -4.638633e+03  
  
 <MIXTURE> 49 2.522372e-02  
 <MEAN> billion 2650.300537 cluster 12 freq\_rank 79  
 <GCONST> -5.326478e+03  
  
 <MIXTURE> 50 1.636658e-02  
 <MEAN> advance 2001.499512 cluster 39 freq\_rank 1704  
 <GCONST> -4.465987e+03

The table below shows the words sorted in decreasing likelihood order. We are pleased to see that the frequency rank begins to look similar as the likelihood rank, which is why we are getting more reasonable perplexity values now.

Word	Likelihood	Cluster ID	Freq Rank
<unk>	2702.929199	12	2
<s>	2702.921143	12	3
</s>	2702.921143	12	4
to	2702.364502	12	6
and	2702.241699	12	8
s	2700.400391	12	42
percent	2695.482422	12	23
the	2692.822266	39	1
of	2685.78418	12	5
by	2677.44751	12	24
with	2677.020264	12	26
corporation	2672.194092	12	86
incorporated	2670.960449	12	85
in	2670.585449	12	9
from	2664.420166	12	28
at	2661.575195	12	25
as	2661.403809	12	22
point	2660.734619	12	14
million	2655.631348	12	29
thousand	2654.174561	12	57

**Table 13**

Although it does not give better perplexity, 100 mixes shows better likelihood rank when compared with frequency rank:

**100 mixes:**

<b>Word</b>	<b>Likelihood</b>	<b>Cluster ID</b>	<b>Freq Rank</b>
the	2703.120361	29	1
<unk>	2702.929199	29	2
<s>	2702.921143	29	3
</s>	2702.921143	29	4
of	2702.390137	29	5
to	2702.364502	29	6
a	2702.319336	29	7
and	2702.241699	29	8
in	2702.080078	29	9
from	2700.665771	29	28
s	2700.400635	29	42
my	2697.830078	29	462
percent	2695.482422	29	23
but	2681.542969	29	38
is	2680.980957	29	13
or	2679.668701	29	54
by	2677.44751	29	24
with	2677.020264	29	26
hundred	2676.650146	29	20
million	2676.331299	29	29

**Table 14****Initialize Training by K-Means Clustering**

While doing K-Means clustering, we find that our clustering results are different from those obtained by incremental training of the GMM. Thus, we are interested in comparing the performance of the incremental training with direct training from the mixture parameters given by K-Means clustering.

We start from 50 mixture components. We first do K-Means clustering to group words into 50 clusters, initialize mean and variance using all the samples in each cluster, and then initialize weight by the proportion of data assigned to each cluster. Then we do several iterations of training by HERest. And we will compare the results obtained using this method with those obtained by incremental training from the single mixture GMM.

**Result:**

First iteration:

average log prob per frame = -4.596169e+02

Second iteration:

Total 1128 floored variance elements in 46 different mixes

average log prob per frame = 1.313644e+03

Third iteration:

Total 9783 floored variance elements in 50 different mixes

average log prob per frame = 1.754920e+03

Fourth iteration:

Total 11908 floored variance elements in 50 different mixes

average log prob per frame = 2.151400e+03

At the fourth iteration, around 50% of the variance elements have been floored. We could see that the likelihood at the first iteration is very low, which means the clusters we got from K-Means clustering does not necessarily resemble the clusters given by GMM. Below is the top 20 words sorted by likelihood in descending order:

<b>Word</b>	<b>Likelihood</b>	<b>Cluster ID</b>	<b>Freq Rank</b>
<s>	2659.550293	17	3
be	2586.09375	20	33
undertake	2568.412598	7	9404
ichi	2556.538086	7	3394
oust	2539.561279	24	6008
capitalize	2539.538574	7	7696
will	2533.78833	17	43
when	2531.680176	17	95
defraud	2529.80127	24	9274
has	2527.523682	17	40
would	2525.313965	17	61
federal	2509.612061	19	130
forty	2505.638184	30	84
have	2503.423584	17	44
circumvent	2500.597412	24	8625
cooperate	2497.94458	7	4099
discuss	2497.773438	7	1873
get	2497.506104	33	226
say	2496.234375	13	145
disclose	2495.648438	20	1885

**Table 11**

## **Bigram Model**

### **Model Setup**

Unlike the unigram model, we use a single state to represent a history in our bigram model, and let it emit any words in the vocabulary. Since we have 10001 words in our vocabulary, and the end of sentence `</s>` can not be a history, we have 10000 histories in total, which means we have 10000 HMM's in our model.

In order to train a robust system, we need to tie some less frequent history states. One way to do this is to use K-Means clustering to cluster the infrequent words, but this might not be the same as what we really need, since K-Means clustering is somewhat different from what is done by EM algorithm (as we illustrated in the last section). So we choose to use HHed to cluster the infrequent words. When we say infrequent words, we mean those words with frequency less than 1000.

We build a 50 mixture GMM to do the bigram experiments. The reason is that there are only 551 history words which has more than 1000 samples. In order to train each component sufficiently, we will need to have more data or reduce our model complexity. We notice that lots of the histories only have 20-30 samples, which is impossible to train 100 mixes. Even when histories are tied, it is very easy to get over fitting for 100 mixture components model. Also, although from the frequency and likelihood rank comparison, 100 mixes looks better than 50 mixes, we got a better scaled perplexity from 50 mixes.

### **Data Preparation**

#### **Training data:**

There is no need to split data for each history, we just write each word vector in the order of their appearance in training text, but neglect “begin-of-sentence”, since it's never related to any history.

**MLF file:**

Simply copy words from training text, but no end-of-sentence, since it's never a history, and it is not included as an HMM (each history word corresponds to an HMM). With this MLF file, like in speech recognition, each word vector will be translated to its history word, and thus has the corresponding HMM trained.

**Testing:**

First split test data according to the history word, which means each history word would have one single test file. For getting likelihood of words, we still use HVite to test on vocabulary file for each single HMM. This time, since the probability of a word depends on its history, we need to force the network to only include the HMM of the history word. To achieve this, we need to change the wdnet file, or simply rewrite grammar file and let it only build a particular network for the specific history.

**Tying the States**

We use HHED to tie the infrequent states which has less than 1000 samples. The script file is written as:

```
RO 1000  
NC 100 "macro_tie" {*.state[2]}
```

With the “NC” command, we are able to do data-driven clustering. In the initial step, each state forms a single cluster. Then two clusters which when combined form the smallest cluster are merged. The size of a cluster is defined as the largest distance between two states in the cluster. This procedure repeats until the number of clusters specified in NC command has been reached. The RO command is used to remove outliers which have less than 1000 samples. The way to do this is to find those outliers and merge them with a nearest neighbor. With RO command, we are ensured to have all “states” (some are clusters) with no less than 1000 samples. After HHED is done, we get 33 clusters for the infrequent states. When combined with the 551 frequent states, we get 584 logical “states” in total.

In the last iteration of training before the states tying procedure, HERest should be instructed to output a statistics file used by RO command for clustering. Also notice that HHED for clustering runs slow: 10 hours in our case. To see the words in each cluster, we could use “-T 256” option provided by HHED.

### **MAP Adaptation**

MAP Adaptation can be used to accomplish a “Maximum a Posteriori” approach. The training is based on the knowledge of a prior for the distribution. In our case, we use the parameters obtained for unigram model as the prior knowledge. And we hope this prior knowledge for the parameters would help us deal with the limited data problem on bigram model. The new update of the parameters depends proportionally on the prior knowledge by a constant  $\tau$ . For example, the update formula for the mean is:

$$\mu \leftarrow \frac{N}{N + \tau} \bar{\mu} + \frac{\tau}{N + \tau} \mu$$

Where N is the occupation likelihood of the training data.

Our experiment starts with 50 mixture components. The first step is to copy the priors defined by unigram model, and then we do 20 iterations of MAP training. Below are the training results. Notice that the likelihood grows slowly:

### **Training Results**

average log prob per frame = 2.359063e+03

average log prob per frame = 2.409118e+03

average log prob per frame = 2.422081e+03

average log prob per frame = 2.427051e+03

average log prob per frame = 2.429291e+03

average log prob per frame = 2.430460e+03

average log prob per frame = 2.431151e+03

average log prob per frame = 2.431613e+03

average log prob per frame = 2.431920e+03

average log prob per frame = 2.432137e+03

average log prob per frame = 2.432310e+03

average log prob per frame = 2.432452e+03

average log prob per frame = 2.432552e+03

average log prob per frame = 2.432625e+03

average log prob per frame = 2.432684e+03

average log prob per frame = 2.432734e+03

average log prob per frame = 2.432782e+03

average log prob per frame = 2.432818e+03

average log prob per frame = 2.432845e+03

average log prob per frame = 2.432871e+03

**Testing Results (before flooring):**

<b>Tao</b>	<b>Perplexity</b>	<b>K</b>
1	1044.230972	160
5	1167.587111	110
10	946.6906779	140

**Table 15**

Notice that the traditional bigram model gives a perplexity of 160.131. So our result is around 6 times that of the traditional model.

Since our original motivation is to better predict the infrequent words, we expect our model to work better on them. We split the vocabulary by frequent and infrequent

words, and calculate perplexity only on frequent or infrequent words, respectively.

Below are the results:

Frequent words: ppl=369.035, length=102408

Infrequent words: ppl=7930.8697, length=45406

Compare with the baseline:

Kneser-Ney:

Frequent words: ppl=53.9987

Infrequent words: ppl=646.332

Unfortunately we did not see any improvement on the infrequent words.

### **Tied-Mixture System**

The Tied-Mixture System defines a common pool of mixture components. The mixtures are shared by all the states in the model. The only difference between the GMM parameters used for different states is the mixture weights. Thus, we are able to simplify our model. And more importantly, we will train a robust system by using all the training data for the mixture components.

We are using the same clustering result as MAP Adaptation for states tying. We could use HHED to build a Tied-Mixture System. The script file is described below:

```
JO 50 1.0
TI "mix" {*.state[2].mix}
HK TIEDHS
```

The JO command defines the number of mixture components in the pool, which is 50 in our case. But the pool could be defined larger for more flexible usage. In my own case, I wrote HMM definitions for the Tied-Mixture System myself without the aid of HHED. I also did 20 iterations of training after the system has been built.

## Training Results

Total 17089 floored variance elements in 48 different mixes  
average log prob per frame = 2.359063e+03

Total 17091 floored variance elements in 48 different mixes  
average log prob per frame = 2.353511e+03

Total 17093 floored variance elements in 48 different mixes  
average log prob per frame = 2.353903e+03

Total 17103 floored variance elements in 48 different mixes  
average log prob per frame = 2.354081e+03

Total 17101 floored variance elements in 48 different mixes  
average log prob per frame = 2.354219e+03

Total 17103 floored variance elements in 48 different mixes  
average log prob per frame = 2.354342e+03

Total 17113 floored variance elements in 48 different mixes  
average log prob per frame = 2.354426e+03

Total 17114 floored variance elements in 48 different mixes  
average log prob per frame = 2.354477e+03

Total 17114 floored variance elements in 48 different mixes  
average log prob per frame = 2.354523e+03

Total 17112 floored variance elements in 48 different mixes  
average log prob per frame = 2.354564e+03

Total 17113 floored variance elements in 48 different mixes  
average log prob per frame = 2.354594e+03

Total 17111 floored variance elements in 48 different mixes  
average log prob per frame = 2.354638e+03

Total 17112 floored variance elements in 48 different mixes  
average log prob per frame = 2.354682e+03

Total 17113 floored variance elements in 48 different mixes  
average log prob per frame = 2.354742e+03

The likelihood during training also grows slowly. The test result is shown below. It is worse than MAP Adaptation.

### Testing Results

Ppl = 1641.2475281318857, K: 150

Calculating entropy of the mixture weight vector could give us a sense of how many active mixture components are used by the model. The entropy is averaged among all the histories:

$$H(h) = -\sum_{i=1}^{50} w_i \log w_i, \text{ where } w_i \text{ is the weight for the } i^{\text{th}} \text{ mixture component, and}$$

$H(h)$  is the entropy for a specific history. The overall entropy is:

$$\sum_h \hat{p}(h)H(h), \text{ where } \hat{p}(h) \text{ is the frequency of the history in training data.}$$

There are 20 iterations of training for Tied-Mixture system, so we list the entropy of the weight vector for each iteration below. We could see from the result that on average, there are around 20 to 30 components active.

5.50643912512859  
4.57677585953816  
4.57623825742097  
4.57592957296079  
4.57553734635482  
4.57562425919126  
4.57553957272825  
4.57543915524667  
4.57542797434822  
4.57536868478398  
4.57531558905327  
4.57522038023043  
4.57501745602971  
4.5749567381027  
4.5750344549059  
4.57500764468702  
4.5749999740881  
4.57503385958696

4.5750155567129  
4.57498207479717  
4.57492106670077

The number of mixture components that should be used is determined by the number of types associated with each history. For the entire history words, the average number of word types they have is 76.22. And for the shared system with tied states, the average number of types for the histories is 178.26, which means we should train more than 50 Gaussian Mixtures. This may explain why the performance of Tie-Mixture system is poor.

# NMF Analysis

NMF and its usage in our research have been introduced in the former section. When dealing with sparse matrix of size  $10001 \times 10001$ , with 762323 nonzero entries, our program uses 50M memory, and runs for less than 1 minute for each update. For the raw count co-occurrence matrix, the divergence between the original matrix and its approximation usually converges after 2000 to 3000 iterations of updates. In this section, we discuss whether the initialization step may affect convergence, introduce another method to initialize the two factor matrices which would be better approximation than randomly initialized ones, and then describe a incremental seeding procedure to reach the final dimension we are looking for.

NMF are usually initialized randomly. Results have shown that there are no clear relationship between a better starting point and a better convergence point. In other words, a smaller divergence obtained at the initial step does not guarantee a smaller divergence when the stable point has been reached. Table 17 shows the change of divergence for 2000 iterations of updates. The first column shows the iteration number, from 0 to 2000. The other columns show the divergence. The first row shows different initial divergence values, and each column corresponds to a single process.

The last column uses a smallest initial divergence. This is obtained by a special initialization strategy. As is stated in the former section, we treat rows of H as a set of basis to span the word vector space. So we use the centroid of word clusters to initialize H. We first do K-Means clustering on rows of the co-occurrence matrix. This could be done fast when using cosine distance, since the vector representation is sparse and thus speed up dot product computation. Then we use the centroid of each cluster to initialize the rows of H. We assume H is orthogonal. So W could be initialized as  $W = VH^T$ . Through this method, we indeed get a better starting point.

However, from table 17 we could see that the convergence rate is not better than randomly initialized factor matrices. It may also not reach a better convergence point.

0	17180532736	789565632	2155008512	541490112	405309824
100	1375742.75	1380716.25	1381377.13	1374163.25	1490062.88
200	1339078.5	1338742.38	1344676.25	1333392.63	1430424.25
300	1325419.63	1325142	1331790.38	1317594.63	1409465.75
400	1318484.38	1317656.13	1324218.75	1308942	1398279.75
500	1313755	1312635.63	1319473.38	1304531.38	1391390.25
600	1310519.5	1309560.13	1316093.38	1301735.88	1386560.75
700	1307573.88	1307501.5	1313535.75	1299563.75	1383549
800	1305916.25	1306267.25	1311658.5	1298080.88	1381185.38
900	1304556	1305481.5	1310328	1297047.25	1379744.5
1000	1303403.25	1304818.75	1308878.25	1296061.5	1378460.38
1100	1302572.88	1304281	1307943.75	1295192	1377616.75
1200	1301911.75	1303746.63	1307243.38	1294402.63	1376909.5
1300	1301452	1303425.75	1306493.88	1293797.63	1376406.75
1400	1301088.25	1303143.63	1305802	1293250.38	1375928.63
1500	1300792.38	1302904.25	1305420.38	1292776.25	1375523.5
1600	1300492.25	1302613.63	1305168.88	1292494.13	1375139.13
1700	1300273.13	1302325.5	1304952	1292193.38	1374857.13
1800	1299945.25	1302111.88	1304696.75	1291972.63	1374592.38
1900	1299708.88	1301884.38	1304574.5	1291764.38	1374289.25
2000	1299371.88	1301734.88	1304427.88	1291466.13	1374088.5

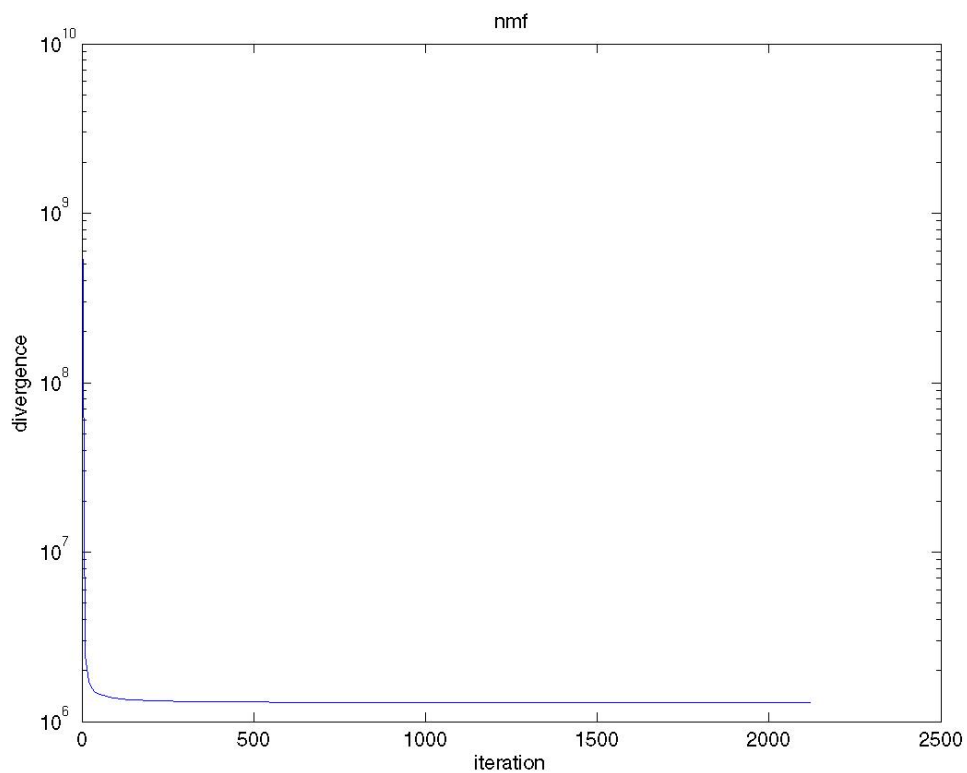
Table 17

Unlike SVD, the first 300 basis in a 500-factor NMF is not the same as the 300 basis in a 300-factor NMF. We had this discussion in the last section. But we still want to see whether we could use a good 300-factor NMF to help initialize a 500-factor NMF. Our experiment for 500-factor NMF goes in stages: we first do 400 iteration of 100-factor NMF, then we use W and H obtained to seed the 200-factor NMF. In order to extend W and H to have 200 columns or rows, we copy the last 100 columns of W and multiply them by  $\frac{1}{\sqrt{2}}$ . Similarly, we copy the last 100 rows of H and multiply

them by  $\frac{1}{\sqrt{2}}$ . Through this way, we have W and H in proper size, while keeping the

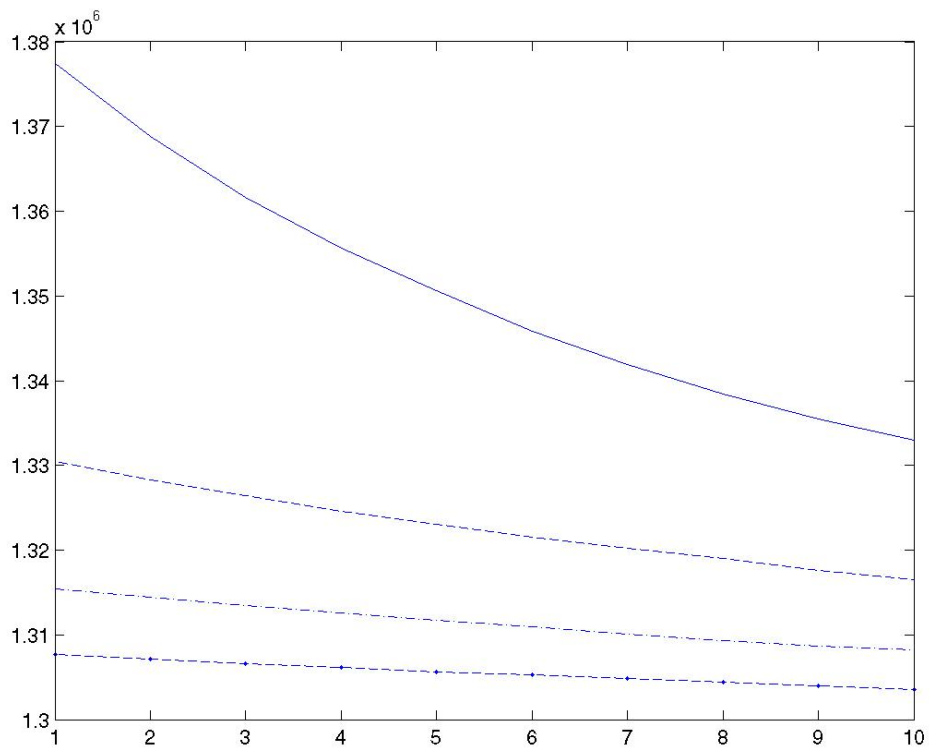
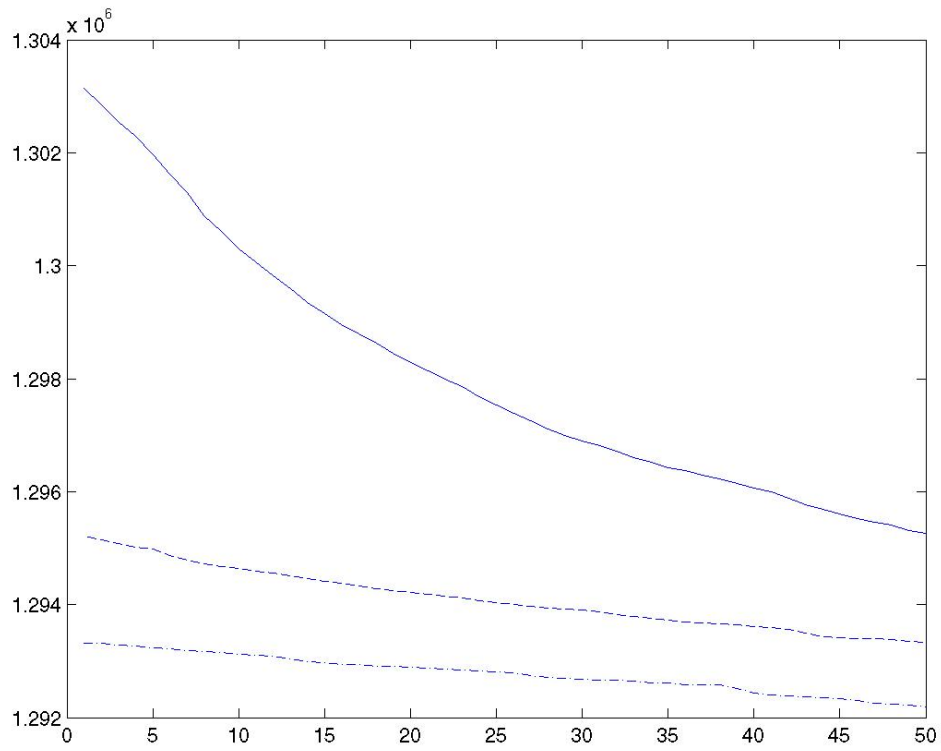
same divergence value obtained at the 400<sup>th</sup> iteration of 100-factor NMF. After that, we do 200-factor NMF for 400 iterations, then use the results to seed a 300-factor NMF. This procedure repeats until we reach 500-factor NMF. Finally we do another 400 iterations of updates for the 500-factor NMF. The total number of iterations we get is still 2000, so we could compare the results with that we got for the traditional method.

Figure 17 shows the convergence of divergence in NMF, with traditional method.



**Figure 17**

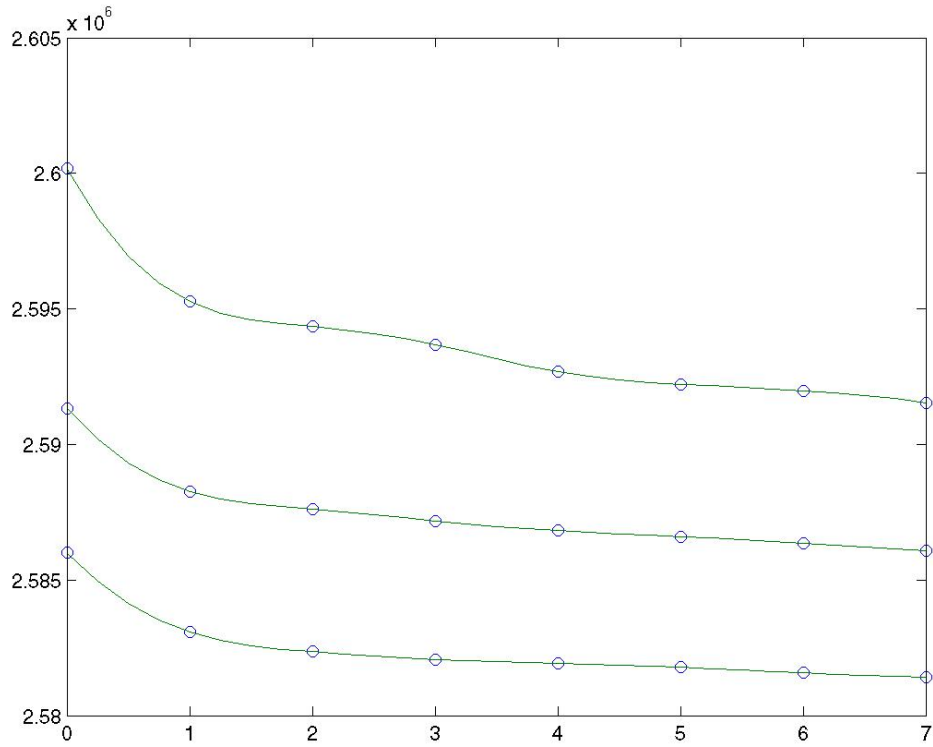
Figure 18 shows the last 1500 iteration of NMF in stages: each line represents 500 iterations. Figure 19 shows the first 500 iteration of NMF in stages: each line represents 100 iterations, except for the very first 100 iterations (the drop is too severe). We could observe the convergence rate from these two plots.



**Figure 19**

Figure 20 shows the results for the “incremental seeding” method. It plots divergence

from iteration 800 to 2000. The upper most line represents iteration 800 to 1200, the middle line represents iteration 1200 to 1600, and the last line represents iteration 1600 to 2000. My own guess is that this seeding method would not help us do either faster NMF or better NMF, since the running time is not obviously faster than the traditional 2000 iteration, and the divergence at 2000 iteration is much higher than what we got from the traditional method.



**Figure 20**

# Bibliography

D. Mrva and P. C. Woodland, "A PLSA-based Language Model for Conversational Telephone Speech", Proc. of Interspeech, 2004.

D. Marva and P. C. Woodland, "Unsupervised Language Model Adaptation for Mandarin Broadcast Conversation Transcription", Proc. of Interspeech, 2006.

J. R. Bellegarda, "Exploiting Latent Semantic Information in Statistical Language Modeling", Proc. of IEEE, 2000.

M. Novak and R. Mammone, "Use of Non-Negative Matrix Factorization for Language Model Adaptation in a Lecture Transcription Task", Proc. of ICASSP 2001

H. Schwenk and J. Gauvain, "Building Continuous Space Language Models for Transcribing European Languages", Proc. of Interspeech, 2005

M. Berry, "Large Scale Sparse Singular Value Computations", International Journal of Supercomputer Applications, 1992

J.K.Cullum, "Lanczos Algorithm for Large Symmetric Eigenvalue Computations", Birkhäuser, 1985

"The HTK Book for HTK Version 3.4", Microsoft Corporation, Cambridge University Engineering Department, 2006

D. D. Lee and H. S. Seung, "Learning the Parts of Objects by Nonnegative Matrix Factorization", Nature 1999.

D. D. Lee and H. S. Seung, "Algorithms for Non-negative Matrix Factorization", In Advances in Neural Information Processing Systems, 2001