Distributional Word Clustering in Parallel

Alan L. Ritter
Computer Science
Western Washington University
Bellingham, WA 98225
ritter.alan@gmail.com

James W. Hearne
Computer Science
Western Washington University
Bellingham, WA 98225
James.Hearne@cs.wwu.edu

Philip A. Nelson
Computer Science
Western Washington University
Bellingham, WA 98225
phil@cs.wwu.edu

Abstract

We discuss various methods which have been applied to grouping words into syntactic and semantic categories, primarily how they deal with the problems of sparsity and computational complexity. We then present a method of distributional clustering, and discuss the parallelization of the most computationally intensive part of this process.

1 Introduction

There are many reasons to group words into syntactic or semantic categories from purely distributional information. For example, tagging words whose part-of-speech properties are not known[12], improving the performance of n-gram models[2] and exploratory data analysis. In addition, distributional clustering is of interest to researchers in the field of Cognitive Science wherein the question of what role distributional information plays in language acquisition is a hotly debated topic[10].

Previous research in this area has avoided using a simple hierarchical complete-link clustering algorithm, due to the time complexity involved. Instead most approaches have used flat clustering algorithms to deal with this problem.

We implement bottom-up hierarchical clustering using a complete-link similarity measure between clusters. For a similarity measure between words the *Kullback Leibler (KL) divergence* is used over context distributions, while utilizing any clustering information we have obtained from previous iterations. This is similar to what is done by Clark[4] and Finch/Chater[10]. We then discuss the parallelization of the most computationally expensive part of this algorithm, namely computing the difference measure between each pair of unique words using the KL divergence.

2 Related Work

There is a small literature sharing the basic presupposition of the present project, namely, that the immediate surroundings of a lexical item provides enough information to meaningfully categorize it in an unsupervised way and without explicit training.

The general approach is motivated both by three considerations, (1) the economic advantage of reducing the effort of categorization (2) the need to understand languages that are not well understood and for which no tagged corpora exist and (3) the desire to engage is clustering without the bias of preexisting lexical categories. Typically, the identification of lexical categories (which may or may not converge with conventional parts of speech) proceeds in two parts. For each pair of words a measure of the similarity of its textual context is computed. Even with rather large corpora, the ratio of vocabulary to the size of the corpus is small enough that the exact repetition of lexical contexts is relatively rare. Some method must be used to compensate for the lack of sufficient evidence for example the resulting similarity clusters can be used, as it is in the current approach, to cluster the words. Contributions to this literature are distinguished primarily by the similarity measures assumed and how they handle the problem of data sparseness.

Clark[4] clusters over the context distribution of immediately left and right words using an iterative flat clustering algorithm. Pereira[9] uses a parser to cluster nouns according to their distribution as direct objects of verbs. Schütze[11] uses the two immediately left and right words as context. Singular value decomposition is then performed on the resulting matrix to reduce the time required to cluster. Finch and Chater[10] extend the idea of distributional word clustering to discover and cluster phrasal categories as well as words.

Importantly, the literature has not addressed the matter of parallelization. Because the learning technique requires large corpora for its meaningful application, decomposing the algorithm into data independent processes that can be executed simultaneously is crucial to its practical application.

3 Hierarchical Clustering

Bottom-up hierarchical clustering algorithms start with one cluster for each item in the data set, and repeatedly merge the two most similar clusters until there is only one left. This produces a tree-like structure in which the relative similarity of pairs of clusters can be determined. For a more detailed description see Manning and Schütze[6] or Frakes and Baeza-Yates[1].

The main advantage of hierarchical clustering is that more information is recorded in the tree structure than in flat clusters alone. Once the tree structure has been created it can easily be converted into flat clusters, so one can experiment with different numbers of flat clusters, cutoff points or prunings.

The main thing that characterizes the various hierarchical clustering algorithms is how the distance between clusters is measured. There are 3 main classes, (1) Single Link, (2) Complete Link and (3) Group Average. Of these complete link produces the best global cluster quality, but is also the most computationally intensive, taking time $O(n^2 \log n)$ in the number of objects to be clustered. Because the complete-link measure produces the best quality clusters, this is what we use.

4 Distance Measure

Before we can begin hierarchical clustering, we need a distance measure between pairs of words. For this purpose, we choose the Kullback Leibler divergence, which is a measure of the difference between two probability distributions p and q, and is defined as follows:

$$\mathrm{KL}(p,q) = \sum_{x \in X} p(x) \log \frac{p(x)}{q(x)}$$

The probability distribution of each unique context in the text is used to compare pairs of words, and we experimented with several different types of context. For instance, consider the sentence in figure 1. If we

> I do not think, however, that *this* **report** *has* come too late.

Figure 1: A sample sentence from the corpus

are to take the context to be the immediately left and right words, then **report** in the above sentence would have the context $\langle$*this*, *has*$\rangle$.

$$\mathbf{P} = \begin{pmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,m-1} \\ p_{1,0} & \ddots & \cdots & p_{1,m-1} \\ \vdots & \ddots & \cdots & \vdots \\ p_{n-1,0} & p_{n-1,1} & \cdots & p_{n-1,m-1} \end{pmatrix}$$

Figure 2: The probability matrix

4.1 Probability Matrix

Our first goal is to construct an $n \times m$ matrix, $\mathbf{P}$, of the probabilities of each unique context given each unique word as illustrated in figure 2. Here n is the number of unique words in the text, and m the number of occurring unique contexts. Note that if the context consists of only a single word then $n = m$, but if the context includes more than one word, m is typically much larger than n.

For each word in the text, w_i, we count the number of times it occurs in each context, $C(w_j w_i w_k)$ and use these frequencies to estimate the probability of each context given the word:

$$\begin{aligned} p_{i,j} &= P(\overrightarrow{C_j}|w_i) \\ &= P(\langle w_k, w_l \rangle | w_i) \\ &\approx \frac{C(w_k w_i w_l)}{C(w_i)} \end{aligned}$$

Since the calculation of Kullback-Leibler Divergence will not permit zeroes in $\mathbf{P}$, the question arises as to how to estimate this probability for contexts which aren't observed for a specific word. Our solution has been to simply assign an arbitrarily small frequency [1] ($\frac{1}{1000}^{\text{th}}$ seems to work well in practice).

4.2 Word Difference Matrix

Once we have the matrix of probabilities, the next step is to compute the difference between each pair of unique words in the text. The $n \times n$ difference matrix $\mathbf{D}$ can be computed as follows:

$$\mathbf{D} = \mathbf{P} \times \mathbf{P}^T$$

where the multiplication operation of matrix multiplication is replaced with the $p_{i,j} \log \frac{p_{i,j}}{p_{j,k}}$ operation from the KL divergence equation. The time required to compute $\mathbf{D}$ directly using matrix multiplication is $O(n^2 m) > O(n^3)$.

This matrix is computed in advance because the difference between each pair of words will be needed

[1] Another solution to this problem is to use Good-Turing smoothing[4].

many times during the single-link bottom up hierarchical clustering algorithm. In practice computing **D** is the most computationally expensive part of the clustering, for two reasons:

1. The hierarchical clustering algorithm has complexity $O(n^2 \log n)$, while the computation of the difference matrix is $O(n^2 m) > O(n^3)$.

2. The constant factors involved in the computation of the word difference table are much larger than those in the hierarchical clustering algorithm. The computation of **D** involves a great deal of floating point log operations, whereas the clustering algorithm can be implemented efficiently using table look-ups.

5 Using Previous Clustering Information

One of the main problems with direct distributional word clustering is that similar words can have very different distributions for example 'a' and 'an' serve similar grammatical functions, but generally appear in different contexts[4]. Our solution has been to use previous clustering information to deal with this problem. This changes the concept of context to be the left and right clusters, as opposed to the left and right words. For example given the following clusters:

$$\begin{aligned} C_1 &= \{\text{this}, \text{these}, \text{those}\} \\ C_2 &= \{\text{has}, \text{was}, \text{had}\} \end{aligned}$$

the context for **report** in figure 1 would now be $\langle C_1, C_2 \rangle$. In this way words derived from **report** such as the plural form, **reports** are more likely to have the same context.

A set of current global clusters is maintained, which are used in the construction of the probability matrix during each iteration[2]. Note that this means the number of unique contexts gets smaller at each iteration.

6 Parallelization

We focus our attention on calculating the word difference matrix, **D**, in parallel because this is the most computationally intensive part of the process. Although the hierarchical clustering itself has a relatively high complexity ($O(n^2 \log n)$), we avoid parallelizing this part of the computation due to the difficulties involved. While there exist cost optimal algorithms for parallelizing complete-link hierarchical clustering for the CRCW PRAM[8], it is generally believed that optimal algorithms are not possible for the more realistic Message-Passing architectures. It turns out that this isn't a problem in practice because of the small constant factors involved in hierarchical clustering. In addition n in this setting is the number of unique words in the text which does not grow linearly in it's length. Note also that number of unique contexts, m, in the $O(n^2 m)$ time taken to compute **D** has a growth rate closer to linear if more than one word of context is used.

Because **D** can be computed via matrix multiplication using addition and $\log \frac{p_{i,j}}{p_{j,k}}$, it would seem natural to parallelize the computation using one of the many known dense matrix multiplication algorithms such as SUMMA[13], PSP[3] or those of Cannon or Nelson[7]. These general algorithms, however don't meet the needs of this specific application for a number of reasons:

1. The probability matrix **P**, comfortably fits into the memory (1GB) of currently available commodity computers. Algorithms such as Cannon's divide up the matrix into sub-matrices, thus allowing larger matrix sizes to be computed.

2. **P** is in some sense "sparse". Although there are no zeroes in the matrix **P**, there are long runs of the same value.

6.1 Run-Length encoding

Before discussing computing **D** in parallel we first discuss a method of speeding up the sequential computation, which is also used in the parallelization.

Because there are long "runs" of the same value in the rows of the probability matrix, we can both reduce the amount of space required to store **P**, and the time needed to compute **D** by representing **P** using a simple *run-length encoding* as illustrated in figure 3. Each element of $\mathbf{D} = \mathbf{P} \times \mathbf{P}^T$ is computed as follows:

$$d_{i,j} = \sum_{k=0}^{m} \left(p_{i,k} \log \frac{p_{i,k}}{p_{j,k}} \right)$$

So, by storing the rows of **P** using the run-length encoding it is possible to more efficiently calculate common runs between rows i and j. For instance supposing that row i has a run of the same values from $k = 20$ to $k = 100$, and row j has a run from $k = 10$ to $k = 70$,

[2] In practice we did not find that doing more than 2 iterations was beneficial

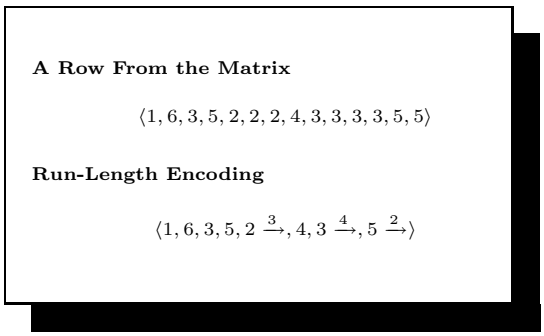

Figure 3: The Run-Length Encoding

we could compute $d_{i,j}$ as follows:

$$\begin{aligned} d_{i,j} &= \sum_{k=0}^{19} \left(p_{i,k} \log \frac{p_{i,k}}{p_{j,k}} \right) \\ &+ 50 \times p_{i,20} \log \frac{p_{i,20}}{p_{j,20}} \\ &+ \sum_{k=70}^{m} \left(p_{i,k} \log p_{i,k} \frac{p_{i,k}}{p_{j,k}} \right) \end{aligned}$$

and thus save 49 floating point divisions and log operations. In practice the matrix is very sparse, so this speeds things up a lot [3].

The speedup gained by using the run-length encoding varies based on the sparseness in a given part of the matrix [4]. In addition the sparseness varies based on the type of context is used.

When using only a single word of context (the immediately right or left word), the the run-length encoding gives us a speedup of approximately 20.

6.2 Data Distribution

For the purposes of hierarchical clustering we only need a lower-triangular distance matrix, $\mathbf{L_D}$, because only one distance measure between each pair of unique words is required. In general $\mathrm{KL}(p,q) \neq \mathrm{KL}(q,p)$, so we define the individual elements of $\mathbf{L_D}$ as follows:

$$l^d_{i,j} = d_{i,j} + d_{j,i} = \mathrm{KL}(p_i, p_j) + \mathrm{KL}(p_j, p_i)$$

This incorporates both $\mathrm{KL}(p,q)$ and $\mathrm{KL}(q,p)$ into the distance measure.

In order to evenly balance the load distribution in calculating the lower-triangular matrix $\mathbf{L_D}$ among the P processors we assign processor r rows $r\frac{n}{2P}$ through

[3] Sorting the columns of $\mathbf{P}$ by context frequency makes this computation go even faster. This is because contexts with similar frequency are next to each other, making longer runs more likely.

[4] In general rows of $\mathbf{P}$ corresponding to the more frequent words in the text are less sparse.

$$\begin{pmatrix} \mathcal{P}_0 & & & & & \\ \mathcal{P}_1 & \mathcal{P}_1 & & & & \\ \mathcal{P}_2 & \mathcal{P}_2 & \mathcal{P}_2 & & & \\ \mathcal{P}_2 & \mathcal{P}_2 & \mathcal{P}_2 & \mathcal{P}_2 & & \\ \mathcal{P}_1 & \mathcal{P}_1 & \mathcal{P}_1 & \mathcal{P}_1 & \mathcal{P}_1 & \\ \mathcal{P}_0 & \mathcal{P}_0 & \mathcal{P}_0 & \mathcal{P}_0 & \mathcal{P}_0 & \mathcal{P}_0 \end{pmatrix}$$

Figure 4: Data distribution of $\mathbf{L_D}$ for 3 processors

$(r+1)\frac{n}{2P} - 1$ and $n - (r+1)\frac{n}{2P}$ through $n - r\frac{n}{2P} - 1$. This assigns a roughly equal portion of the individual elements of $\mathbf{L_D}$ to each CPU, and is illustrated in figure 4.

6.3 Computing $\mathbf{L_D}$ in Parallel

The computation of $\mathbf{L_D}$ is summarized in algorithm 1. Each processor computes its assigned rows of

Algorithm 1 Calculate part of $\mathbf{L_D}$ on processor r

```
//Compute the first set of rows assigned to r
for i = r n/2p to (r+1) n/2P − 1 do
  for j = 0 to i − 1 do
    //Exploit the run-length encoding
    l^d_{i,j} = Σ_{k=0}^{m} ( p_ik log p_ik/p_jk ) + Σ_{k=0}^{m} ( p_jk log p_jk/p_ik )
  end for
end for
//Compute the second set of rows assigned to r
for i = n − (r+1) n/2p to n − r n/2P − 1 do
  for j = 0 to i − 1 do
    //Exploit the run-length encoding
    l^d_{i,j} = Σ_{k=0}^{m} ( p_ik log p_ik/p_jk ) + Σ_{k=0}^{m} ( p_jk log p_jk/p_ik )
  end for
end for
```

$\mathbf{L_D}$, which are then collected on processor 0. At this point complete-link bottom up hierarchical clustering can be done in a reasonable amount of time.

The total time to compute $\mathbf{L_D}$ on p processors where n is the number of unique words in the text, and m the number of unique contexts is $O(\frac{n^2 m}{p})$, And the time to communicate the results back to processor 0 is $O(n^2)$ (The size of $\mathbf{L_D}$) regardless of the number of processors used. Finally, the time required for bottom up hierarchical clustering on processor 0 is $O(n^2 \log n)$, thus the total time required is:

$$\underbrace{O(\frac{n^2 m}{p})}_{\text{Compute}\mathbf{L_D}} + \underbrace{O(n^2 \log n)}_{\text{Clustering}} + \underbrace{O(n^2)}_{\text{Communication}} = O(\frac{n^2 m}{p})$$

because n is much smaller than m.

7 Results

We tested our methods on two very different corpora; Golding's translation of Ovid's Metamorphosis, and a collection of proceedings from European Union Parliament[5][5]. The Golding corpus consists of Old English, and has a very narrative tone, whereas in Europarl the English is more modern, and there are more interrogative, declarative and imperative sentences.

Table 1: Corpora

	Length	words	contexts
Golding	163,772	14,310	104,081
Europarl[5]	2,656,126	29,202	687,591

Timings were collected on a cluster of 45 workstations connected via 10Mbit Ethernet with 3.2 GHz Pentium IV processors running NetBSD and using using MPICH for communication.

7.1 Efficiency

The time taken to compute $\mathbf{L_D}$ for the most frequent 8,000 words along with experimental speedup and efficiency is illustrated for both corpora in tables 2 and 3.

Note that for the Golding corpus it took about 567 seconds to do the sequential hierarchical clustering, which is about as long as it took to compute $\mathbf{L_D}$ sequentially. The time needed to compute $\mathbf{L_D}$, however, has a larger growth rate with respect to the length of the corpus than does the time needed by the hierarchical clustering algorithm. Thus the benefits of parallelization are more apparent when working with larger corpora, as illustrated in table 3.

Table 2: Timings from Golding

processors	time (sec)	speedup	efficiency
1	706.1	1	1
2	371.2	1.902	0.9511
4	192.6	3.666	0.9165
8	104.1	6.783	0.8479
12	75.6	9.34	0.7783
16	61.1	11.6	0.7223
32	40.4	17.5	0.5462
45	34.3	20.6	0.4575

8 Summary

When clustering objects such as words it makes sense to use a hierarchical representation, as it is convenient to experiment with cutoff points or prunings. When working with words it is easy to make educated decisions about which clusterings make sense and which do not.

[5]Only a subset of the English translation was used.

Table 3: Timings from Europarl

processors	time (sec)	speedup	efficiency
1	6330.4	1	1
2	3250.1	1.9478	0.97388
4	1618.2	3.9120	0.97800
8	791.7	7.996	0.9995
12	513.8	12.32	1.0267
16	386.5	16.38	1.0236
32	199.2	31.78	0.9931
45	143.7	44.05	0.9790

The main reason cited for avoiding the use of complete-link hierarchical clustering is the computational complexity required ($O(n^2 \log n)$) in the number of objects being clustered), in addition to the fact that single-link clustering is not easily parallelized. In the setting of distributional word clustering however, the objects being clustered are words, and thus n, the number of unique words does not grow linearly in the length of the corpus, so there is little reason to avoid hierarchical clustering in this setting.

The main computational obstacle in hierarchical distributional word clustering is not the clustering algorithm itself. Instead it is the initial calculation of the difference measure between each unique pair of words. This is because the number of unique contexts has a larger growth rate than the number of unique words (if more than one word is used as context). The computation of this difference matrix can easily be parallelized.

Great improvements can be made in the performance of the calculation of $\mathbf{L_D}$ by exploiting a simple run-length encoding. Additionally as the size of the corpus gets larger, parallelization becomes more beneficial.

References

[1] Ricardo A. Baeza-Yates. *Introduction to data structures and algorithms related to information retrieval*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[2] Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. Class-based n-gram models of natural language. *Comput. Linguist.*, 18(4):467–479, 1992.

[3] Bradford L. Chamberlain, E. Christopher Lewis, and Lawrence Snyder. Problem space promotion

and its evaluation as a technique for efficient parallel computation. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 311–318, New York, NY, USA, 1999. ACM Press.

[4] Alexander Clark. Inducing syntactic categories by context distribution clustering. In *Proceedings of the 2nd workshop on Learning language in logic and the 4th conference on Computational natural language learning*, pages 91–94, Morristown, NJ, USA, 2000. Association for Computational Linguistics.

[5] Philipp Koehn. Europarl: A multilingual corpus for evaluation of machine translation. http://people.csail.mit.edu/koehn/publications/europarl/.

[6] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT Press, Cambridge, MA, USA, 1999.

[7] Philip A. Nelson. Hypercube matrix multiplication. *Parallel Computing*, 19(7):777–788, 1993.

[8] Clark F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21(8):1313–1325, 1995.

[9] Fernando C. N. Pereira, Naftali Tishby, and Lillian Lee. Distributional clustering of english words. In *Meeting of the Association for Computational Linguistics*, pages 183–190, 1993.

[10] N. Chater S. Finch and M. Redington. *Acquiring syntactic information from distributional statistics*. UCL Press, 1995.

[11] Hinrich Schütze. Part-of-speech induction from scratch. In *Proceedings of the 31st annual meeting on Association for Computational Linguistics*, pages 251–258, Morristown, NJ, USA, 1993. Association for Computational Linguistics.

[12] Hinrich Schütze. Distributional part-of-speech tagging. In *Proceedings of the seventh conference on European chapter of the Association for Computational Linguistics*, pages 141–148, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[13] Robert A. van de Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, Austin, TX, USA, 1995.